Tools for Parallel I/O and Compute Intensive Applications

THÈSE N^{O} 1915 (1998)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

par

VINCENT MESSERLI
Ingénieur-informaticien EPFL
originaire de Rüeggisberg (BE)

Lausanne, EPFL
1998

Acknowledgments

I have many individuals to thank for the successful completion of this research.

First and foremost, I am especially grateful to my research director, Professor Roger-D. Hersch, who helped me to find a challenging and interesting research area, who gave me guidance and support throughout this research, and who spent hours reading and commenting on earlier drafts of this dissertation.

I am particularly indebted to Dr Benoit Gennart for his technical expertise, for his help in correcting bugs at any time of the day, and for the extensive discussions we had, which helped me clarify my ideas, as well as led me to new ideas. This thesis has been possible thanks to the research pursued by Dr. Benoit Gennart who has developed the CAP computer-aided parallelization tool.

I would like to thank the members of the Peripheral System Laboratory, among them Marc Mazzariol, Oscar Figueiredo, Joaquín Tárraga, Oliver Courtois, Samuel Vetsch, for their technical collaboration and support. They were good friends, and made this research enjoyable on a day-to-day basis providing encouragement and motivation.

I would like to thank Professors Claude Petitpierre, David Kotz, John H. Maddocks and Dr Fabrizio Gagliardi, who served on my thesis commitee.

I am grateful to the Department of Computer Science at EPFL and its staff for the excellent facilities.

I would like to thank Prof. Badoux, Président of EPFL for the PhD candidate grant he provided during 3 years.

Finally, I would like to thank my family for all the encouragement they have provided me throughout the 4 years of research at EPFL.

Résumé

La plupart des systèmes de fichiers parallèles actuels offrent une vue séquentielle des fichiers. Pour l'utilisateur, un fichier parallèle est vu comme une suite d'octets adressable. Afin d'augmenter le débit d'entrées-sorties, ces systèmes parallèles distribuent de manière transparente les données d'un fichier sur plusieurs disques magnétiques. La compatibilité avec Unix offre une meilleure portabilité des applications, mais dissimule le parallélisme sous-jacent à l'intérieur des systèmes de fichiers, empêchant toute optimisation des requêtes d'entrées-sorties provenant de différents processus. Bien que les systèmes de fichiers parallèles récents incorporent des interfaces plus élaborées, par exemple des interfaces "collectives" permettant de coordonner les accès d'entrées-sorties de plusieurs processus, ces systèmes manquent de flexibilité et ne fournissent aucun moyen pour exécuter des opérations de traitements spécifiques à des applications sur des noeuds d'entrées-sorties. En effet, afin de réduire la quantité de données voyageant sur le réseau de communication, il serait souhaitable d'exécuter, localement, les opérations de traitements où les données résident. De plus, actuellement, les programmeurs doivent utiliser deux systèmes différents: un système de fichiers parallèles pour le stockage parallèle et, un système de communication pour coordonner les calculs parallèles. Cette séparation entre stockage et traitement permet difficilement aux programmeurs de combiner, d'une manière optimale, accès aux fichiers et calcul, c'est-à-dire, le chevauchement des requêtes d'entrées-sorties asynchrones et des traitements.

Cette thèse propose une nouvelle approche pour développer des applications effectuant en parallèle des opérations de traitements et d'entrées-sorties sur un réseau de PC. En utilisant l'outil d'aide à la parallélisation (CAP), les programmeurs d'applications développent séparément les parties séquentielles et, expriment le comportement parallèle du programme à un niveau d'abstraction élevé. Cette description de haut-niveau est automatiquement transcrite en un programme C++ compilable et exécutable. Grâce à un fichier de configuration spécifiant la répartition des processus légers ("threads") d'une application sur l'architecture parallèle, un même programme peut s'exécuter sans recompilation sur différentes configurations matérielles.

Dans le contexte de cette thèse et, à l'aide de l'outil d'aide à la parallélisation (CAP), un serveur de stockage et de traitement parallèle (PS^2), comprenant une librairie de composants réutilisables pour l'accès à des fichiers parallèles, a été réalisé. Grâce aux formalisme du langage CAP, ces composants pour systèmes de fichiers parallèles peuvent être facilement et efficacement combinés avec des opérations de traitements afin de développer des applications effectuant, en pipeline et en parallèle, des traitements et des entrées-sorties. Ces programmes peuvent s'exécuter sur des serveurs PC offrant leurs services de stockages et de traitements aux clients situés sur le réseau.

Ce travail présente l'outil d'aide à la parallélisation (CAP) et son système de communication sous-jacent accompagné d'une analyse de performance. Le serveur de stockage et de traitement parallèle (PS^2) est présenté, ainsi que son implémentation utilisant l'outil CAP. Ce mémoire décrit également comment les programmeurs peuvent personaliser le serveur PS^2 afin de développer des applications de traitements et d'entrées-sorties parallèles. Plusieurs exemples sont fournis, parmi eux des opérations de traitement d'images parallèles. Finalement, la mise en pratique de l'outil CAP et du serveur PS^2 pour le développement d'applications industrielles est démontré avec une application parallèle d'extraction de plan de coupe dans un volume tomographique 3D. Cette application a été, avec succès, intégrée à un serveur Web et est connue sous le nom de "Visible Human Slice WEB server" accessible à l'adresse "http://visiblehuman.epfl.ch".

Abstract

Most parallel file systems provide applications with conventional sequential views of files that stripe data transparently across multiple disks thus reducing the bottleneck of relatively slow disk access throughput. The Unix-like interface increases the ease of application portability, but conceals the underlying parallelism within the file system precluding any optimization of the I/O access pattern from different processes. Although recent multiprocessor file systems incorporate more sophisticated I/O interfaces, e.g. collective I/O interfaces, enabling the I/O access-pattern information to flow from an application program down to the data management system, they still lack flexibility and they do not enable application-specific codes to run on I/O nodes. Therefore application-specific processing operations cannot be directly executed where data resides. Moreover, with the conventional approach of developing parallel I/O- and compute- intensive applications on distributed memory architectures, programmers are faced with two different systems: a parallel file system for parallel I/O and a message passing system for parallel computation. This separation between storage and processing makes it difficult for application programmers to combine I/O and computation in an efficient manner, i.e. overlap I/O requests with computations.

In this thesis we propose a new approach for developing parallel- I/O and compute- intensive applications on distributed memory PC's. Using the CAP Computer-Aided Parallelization tool, application programmers create separately the serial program parts and express the parallel behaviour of the program at a high level of abstraction. This high-level parallel program description is preprocessed into a compilable and executable C++ source parallel program. Thanks to a configuration file specifying the layout of the application threads onto the parallel architecture, a same program can run without recompilation on different hardware configurations.

In the context of this thesis we have designed the runtime system of CAP incorporating among others a message-passing system, and, using the CAP tool, a Parallel Storage and Processing Server (PS^2) comprising a library of reusable low-level parallel file system components. Thanks to the CAP formalism, these low-level parallel file system components can be combined with processing operations in order to yield efficient pipelined parallel I/O and compute intensive programs. These programs may run on multi-PC servers offering their storage and processing services to clients located over the network.

In this work, we present the CAP computer-aided parallelization tool and its underlying message-passing system along with a performance analysis. We introduce the PS^2 framework and discuss its design and implementation using the CAP tool. We describe the new two-dimensional extent-oriented structure of a parallel file and show how the PS^2 server can be customized by developers in order to yield efficient parallel I/O and compute intensive applications. Several examples are provided, among them parallel imaging operations. Finally, the applicability and the performance of the CAP tool and the PS^2 framework on real applications is demonstrated with a parallel 3D tomographic image server application enabling clients to specify and access parallel slices having any desired position and orientation. This application has been successfully interfaced to a Web server and it is known as the Visible Human Slice WEB server (http://visiblehuman.epfl.ch).

Table of Contents

List of Figures

List of Programs

List of Tables

Chapter 1

Introduction

1.1 The IO/computation gap

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law [Amdahl67]. Amdahl's Law states that the performance improvement to be gained from using a faster mode of execution is limited by the fraction of time the faster mode can be used.

Suppose that an enhancement *E* accelerates a fraction *F* of a task by a factor *S*, and the remainder of the task is unaffected (Figure 1-1). Then, the overall speed-up, i.e. how much faster the task will run using the machine with the enhancement E as opposed to the original machine, is given by Equation 1-1.

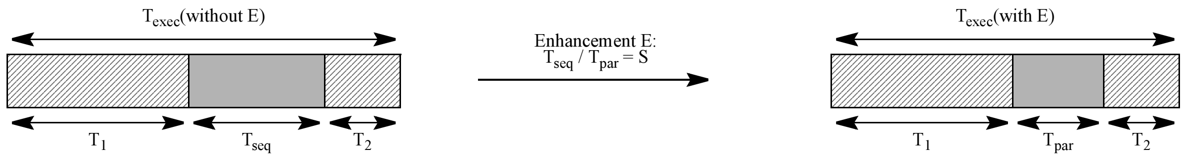


Figure 1-1. Enhancement *E* accelerates a fraction *F* of a task by a factor *S*, and the remainder of the task is unaffected

$$\text{Speedup}_{\text{overall}} = \frac{T_{\text{exec}}(\text{without E})}{T_{\text{exec}}(\text{with E})} = \frac{1}{(1-F)+F/S} \qquad \textbf{(1-1)}$$

Amdahl's Law, i.e. Equation 1-1, expresses the law of diminishing returns: the incremental improvement in speed-up gained by an additional improvement in the performance of just a portion of the computation diminishes as improvements are added. An important corollary of Amdahl's Law is that if an enhancement is only usable for a fraction of a task, we cannot speed up the task by more that the reciprocal of 1 minus that fraction (assuming *S* tends to be infinite).

Amdahl's Law implies that the overall performance of a computer system is limited by the performance of its slowest subsystem or component (processor, memory, disk, or network), i.e. a computer system with faster processors or more processors needs more disk I/O throughput to maintain the system balanced. For example, suppose that a sequential application running on a single-processor single-disk computer spends 90% of its execution time doing computations and 10% of its time doing disk I/O's. Then, after having parallelized the algorithm, the overall application speed-up will be 5.26 on a 10-processor single-disk machine (Equation 1-1). And on a 100-processor single-disk machine, the overall speed-up will be less than 10. This poor speed-up is due to the unbalanced computer system where 100 processors improve the performance of 90% of the execution time of the application but leave 10% unaffected (disk I/O's). Therefore, Amdahl's rule of thumb claims that a balanced computer system needs about 1 MBytes of main memory capacity and 1 Mbits/s of I/O bandwidth per 1 MIPS of CPU performance [Amdahl67].

Contemporary high performance computer systems are becoming increasingly unbalanced. Processor speeds have been improving steadily and substantially for a long time (roughly 35% to 50% growth per year) [Hennessy96]. Memory access times have decreased 30% to 80% each year over the last decade. However, performance gains in disk I/O subsystems have improved only marginally, i.e. less than 10% performance improvement per year. Although the increase in disk storage density has kept pace with gains in memory capacity[1], disk access times, limited by mechanical delays, have improved little in comparison with improvements in memory access times and in processor speeds over the last decades. As a result of these disparities, the disk I/O subsystem has become a potential performance bottleneck in most computer systems. It becomes increasingly difficult to provide sufficient I/O bandwidth to keep applications running at full speed for large problems consuming large amounts of data.

1. Disk density increases by about 50% per year, quadrupling in just over three years; memory density increases by about 60% per year, quadrupling in three years [Hennessy96].

This widening IO/computation gap is known as the I/O gap problem or the I/O crisis [Patterson88, Kotz91, Huber95b]. This problem is further aggravated for parallel applications where the processing power of the applications is increased by using multiple processors.

The most promising solution for balancing the performance of the storage subsystem with other subsystems in a (multiprocessor) computer, i.e. to fill the IO/computation gap, is to extend parallelism to the disk I/O subsystem. However, the I/O crisis is not the only motivation for parallel storage subsystems. Several traditional and emerging I/O- and compute- intensive applications inherently require high-speed and high-volume data transfer to and from secondary storage devices.

1.2 I/O- and compute- intensive applications that need parallel storage-and-processing systems

In addition to the quantitative effect (the I/O gap problem), a second, perhaps more important, qualitative effect is driving the need for parallel storage-and-processing systems. Current (parallel) computers offer tremendous computing power by incorporating (multiple) faster processors. They enable the creation of new applications and greatly expand the scope of existing applications. Besides requiring enormous processing power, these emerging I/O- and compute- intensive applications manipulate huge amounts of persistent data that must be rapidly transferred to and from secondary storage devices. Some of these I/O- and compute- intensive applications are discussed below.

Scientific computing

The area of scientific computing has a profusion of I/O- and compute- intensive applications that need to process large amounts of data [Pool94]. They include biology applications such as neural simulation systems and atomic structure of viruses; chemistry applications such as quantum chemical reaction dynamics, electronic structure of superconductors; earth science applications such as seismic data processing, community climate models, and weather forecasting; astronomy applications [Karpovich93] for example processing data from astronomical instruments; and engineering applications such as Navier-Stokes turbulence simulations. These applications typically use enormous working data sets that are stored as scratch files. They often write large checkpoint files in order to save the computation state in case of a crash and often produce large output files that are used for visualization. These I/O- and compute- intensive applications certainly need parallel storage-and-processing systems that can sustain very high disk I/O throughputs and offer tremendous computing power.

Multimedia

Multimedia applications also have inherently high I/O demands with real-time deadlines. They are usually interactive, and therefore less tolerant to high latency. Video-on-demand, an emerging field in this application area, not only requires high and variable data rates, but also requires synchronization among competing video streams. For this kind of applications, specific multimedia parallel storage systems exist. Besides declustering video streams across several disks, these multimedia parallel storage systems (devised exclusively for multimedia applications) include real-time disk scheduling algorithms, disk access deadlines, synchronization mechanisms among the competing streams, quality of services, resource reservations, admission control, etc. Besides requiring high disk I/O bandwidth, multimedia applications may also require enormous computing power for performing real-time processing operations on streams, e.g. compressing a stream according to the available network throughput, extracting a 3-D stream (x-y-time) from a 4-D stream (x-y-z-time). Therefore, such I/O- and compute- intensive multimedia applications certainly also need powerful parallel storage-and-processing systems.

Wide area information servers

Over the last couple of years, the Web has added a new dimension to wide area information servers. Suddenly, department-scale storage systems are exposed to information access from users all over the world. Web access has not only increased the number of clients that a local system has to sustain, it has fundamentally affected the nature of information access. Not only does a Web server handle many disk I/O requests ranging from very small (e.g. a simple html file) to very large (e.g. a video), but also it processes informations before data is actually transmitted. For example, a search engine Web server has to process huge amounts of data distributed across multiple disks for satisfying one request. Parallel storage-and-processing systems are also suitable as Web server architectures.

1.3 Contribution of this research

With the emergence of fast networks such as Fast Ethernet and Myrinet, running parallel I/O- and compute-intensive applications on a network of PC's are now possible. Making use of a large number of commodity components (PC's, Fast Ethernet, SCSI-2 disks) working in parallel, i.e. parallel processing on several PC's and parallel access to many independent disks, offers the potential of scalable processing power and scalable disk access bandwidth.

The main problem of using parallel distributed memory computers is the creation of a parallel application made of many threads running on different computers. Creating parallel I/O- and compute- intensive programs with completely asynchronous communications and disk I/O accesses is possible, but difficult and error prone. Tiny programming errors in respect to synchronization and information transfer lead to deadlocks which are very hard to debug. The difficulty of building reliable parallel programs on distributed memory computers is one of the reasons why most commercial parallel computers are SMP computers, i.e. computers incorporating shared memory whose processors interact via shared memory and synchronization semaphores (for example SGI Origin 2000 multiprocessor systems). However, compared to SMP computers, PC-based parallel computers can offer cheaper solutions, and have the potential of making parallel processing affordable to medium and small size companies.

In addition, accessing in parallel many independent disks at a time located on different computers requires appropriate parallel storage system support, i.e. means of declustering a parallel file into a set of disks located on different computers and of providing metainformation specifying where the file stripes are located and how the declustering is being done.

This research presents new methods and tools to build parallel I/O- and compute- intensive applications based on commodity components (PC's, SCSI-2 disks, Fast Ethernet, Windows NT). Using a Computer-Aided Parallelization tool called CAP[1], application programmers create separately the serial program parts and express the parallel behaviour of the program at a high level of abstraction. This high-level parallel CAP program description specifies a macro-dataflow, i.e. a flow of data and parameters between operations running on the same or on different processors. This parallel CAP program description is preprocessed into a compilable and executable C++ source parallel program.

In the context of this thesis I have contributed to CAP by developing its portable runtime system, incorporating a CAP-oriented message-passing system (MPS), and by specifying new features to be incorporated into CAP, as a result of the present thesis research work.

This dissertation describes the CAP language, its underlying message-passing system, and the customizable parallel storage and processing server (PS^2), based on CAP, offering a library of low-level parallel file system components. Thanks to the CAP formalism, these low-level parallel file access operations can be combined with application-specific or library-specific processing operations in order to yield efficient pipelined parallel I/O- and compute- intensive programs. These programs may run on multi-PC servers offering their access and processing services to clients located over the network.

1.4 Outline of this thesis

This dissertation is structured as follows. In the next chapter we briefly presents some related parallel file systems. In Chapter 3 we describe the CAP Computer-Aided Parallelization tool facilitating the development of parallel applications running on distributed memory multi-processors, e.g. PC's connected by Fast Ethernet. Chapter 4 experimentally analyses the performances of a multi-SCSI disk array hooked onto a same PC by several SCSI-2 strings and evaluates the potential overheads of the Windows NT operating system in terms of disk access latency time and processor utilization. The second part of Chapter 4 is devoted to the message-passing system (MPS) enabling a distributed CAP application to asynchronously send and receive messages across the network. MPS has been specially developed for the CAP computer-aided parallelization tool. We show how to ensure asynchronous communications over TCP/IP connections by using the Microsoft Windows Socket 2.0 interface and how MPS serializes messages. The end of Chapter 4 gives a detailed performance analysis of MPS. We compare MPS with Windows Socket 2.0 for two different communication patterns found in real CAP

1. The CAP computer-aided parallelization tool has been partly devised by a colleague Dr. Benoit Gennart.

applications: an unpipelined data transfer and a pipelined data transfer. Chapter 5 describes the PS^2 parallel storage and processing server: a framework for developing parallel I/O- and compute- intensive applications using the CAP computer-aided parallelization tool. We show the new extent-oriented two-dimensional structure of a parallel file that enables programmers to develop high-level libraries providing application-specific abstractions of files along with parallel processing operations on parallel files. Chapter 6 describes our new approach for developing parallel I/O- and compute- intensive applications or libraries using the customizable PS^2 parallel file system. We apply the PS^2 methodology to synthesize parallel imaging operations. We demonstrate that, thanks to CAP, the generated operations are very flexible and additional processing steps can be easily added to a pipelined parallel program. The applicability and the performance of the CAP tool and the PS^2 framework on real applications is demonstrated with a parallel 3D tomographic image server application enabling clients to specify and access in parallel image slices having any desired position and orientation. Evaluation of the obtained Visual Human image slice access times shows that performances are close to the best performance obtainable by the underlying hardware. Finally, in Chapter 7, we outline the important conclusions from our work and suggest possible future work.

Chapter 2

Related Research

This chapter presents some parallel file systems.

2.1 The Portable Parallel File System

The Portable Parallel File System (PPFS) [Huber95a, Huber95b] has been developed at the University of Illinois at Urbana-Champaign for quickly experimenting and exploring a wide variety of data placement and data management policies. They claim that the performance of parallel input/output systems is particularly sensitive to access pattern, data distribution, cache size, cache location, cache replacement policy, data prefetching, and write behind. As an answer to these issues, PPFS supports *malleable access* by providing applications with a high degree of control on:

- **Parallel File Access Modes**.
  Under PPFS, files consist of a sequence of variable size *records*, which are numbered starting from zero. The record is the unit of access; only entire records may be read or written by specifying records by number. An application may also specify an access pattern, a sequence of records, which specifies how the records will be accessed. This pattern may be used in two ways: a global access pattern determines how the application as a whole will access the file, and a local access pattern determines how one particular process accesses the file. Besides that, PPFS allows applications to overlap computation with I/O operations by providing both blocking and non-blocking calls.
- **Caching**.
  PPFS employs three levels of caching. 1) Each I/O device has an associated server cache, which only contains data residing on the given device. 2) Each application node has a client cache which holds data accessed by the given user process. 3) Finally an application may use one or more global caches. PPFS may be extended by adding new replacement, prefetching and write behind policies to the system caches. Each cache may potentially have a different set of policies for each file. Furthermore, policies may changed during program execution.
- **Data placement**.
  Under PPFS, a parallel file is divided into *segments*, each of which reside on a particular I/O device. Each segment is simply a collection of records. An application has full control over 1) the parallel file's *clustering* which associates an I/O device with each segment; 2) the parallel file's *distribution* that determines, for each record, in which segment the record resides; 3) and the *indexing* scheme which determines the length and location of the record in that segment.

The organization of PPFS is based on the client/server model. *Clients* are user application processes. There are *Data Servers* which are abstraction of I/O devices. They are built on top of the underlying native file system, e.g. UNIX. These servers cache data, perform physical I/Os and prefetch data as specified by the clients. There is a *Metadata Server* which coordinates the creation, opening and closing of all parallel files and maintains the directory information. In addition to these, there are optional *Caching Agents* acting as coordinators for parallel files which require global caching or access modes.

2.2 The Galley Parallel File System

Thanks to an extensive parallel file system workload characterization on an iPSC/860 multiprocessor machine running Numerical Aerodynamics Simulation (NAS) applications [Kotz94a], the authors have remarked that *strided I/O requests* are common for realistic scientific multiprocessor applications. The authors refer to a set of I/O requests to a file as a *simple-strided* access pattern if each request is the same size and if the offset of the file pointer is incremented by the same amount between each request. A group of requests that appear to be part of a simple-strided pattern is defined as a *strided segment*. And a *nested-strided* access pattern is similar to a simple-strided access pattern but rather than being composed of simple requests separated by regular strides in the file, it is composed of strided segments separated by regular strides in the file.

Based on that observation, the authors of Galley [Nieuwejaar94, Nieuwejaar96] believe that the conventional Unix-like interface, enabling applications to access multiple disks transparently, is inefficient and insufficient for parallel file systems. Indeed this interface conceals the parallelism within the file system, increasing the ease of programmability, but making it difficult or impossible for sophisticated programmers and libraries to use knowledge about their I/O needs to improve the behaviour of their parallel program. Moreover they argue that an interface where an application can explicitly make simple- and nested-strided requests, would reduce the number of I/O requests, and provide additional improvements [Kotz94b, Rosario93, Nitzberg92]. Their analysis suggests the workload for which most multiprocessor file systems were optimized is very different from the workloads they are actually being asked to support.

The authors of Galley are also convinced that designing a general-purpose parallel file system that is intended to directly meet the specific needs of every user will lead to an inefficient system. In response to these motivations, Nils Nieuwejaar at Dartmouth College designed the Galley parallel file system as a more general one that lends itself to supporting a wide variety of libraries, each of which should be designed to meet the needs of a specific community of users [Nieuwejaar96].

In order to provide applications with the ability to fully control the declustering strategy according to their own needs, Galley introduces a new three-dimensional file structure where files are composed of one or more *subfiles*. Each subfile resides entirely on a single disk, and no disk contains more than one subfile from any file. Recursively each subfile is structured as a collection of one or more independent *forks*. A fork is a named, addressable, linear sequence of bytes, similar to a traditional regular Unix file. Unlike the number of subfiles in a file, the number of forks in a subfile is not fixed; libraries and applications may add forks to, or remove forks from a subfile at any time.

Letting applications control both how the data is distributed across the disks and control the degree of parallelism exercised on every subsequent access, Galley only provides data-access requests from a single fork. In addition to the traditional Unix read and write interface, simple-strided and nested-strided interfaces are available.

The organization of Galley is based on the client-server model. There are *Clients* which are simply user applications linked with the Galley run-time library running on compute processors. No caching is done at the client side, the Galley run-time library merely translates file-system requests from the application into lower-level requests and passes them directly to the appropriate *I/O Servers* running on I/O processors.

Each I/O server is composed of a *CacheManager* and a *DiskManager*. The CacheManager maintains a LRU write-back cache of 32KB disk blocks while the DiskManager implements a custom single-disk file system based on the Unix raw device. Galley's DiskManager does not attempt to prefetch data. Indiscriminate prefetching can cause thrashing in the buffer cache. Prefetching is based on the assumption that the system can intelligently guess what an application is going to request next. Using the interfaces described above, there is no need for Galley to make guesses about an application's behaviour; the application is able to explicitly provide that information to each I/O server.

In contrast with most other parallel file systems like Intel's CFS [Pierce89], Galley does not rely on a metadata server to locate files which would create a single point of congestion that could limit the system's scalability. Rather, metadata are distributed across all the I/O servers in the system. To find the I/O server that manages a given file's metadata, a simple hash function is applied to the file name. Vesta uses a similar hashing scheme for their naming system [Corbett96].

2.3 The Vesta Parallel File System

The main innovation in Vesta is the fact that it breaks away from the conventional one-dimensional sequential file structure [Corbett96]. Files is Vesta are two-dimensional, and are composed of one or more *cells*, each of which is a sequence of *basic striping units (*BSUs). BSUs are essentially records, or fixed-sized sequences of bytes. The number of cells and the BSU size are the two parameters that define the structure of a Vesta file. They are defined when the file is created and cannot be changed thereafter. Like Galley's subfiles or PPFS's segments, each cell resides on a single disk. However, if there are more cells than I/O nodes, the cells will be distributed to the I/O nodes in round-robin manner.

Vesta is also unique in terms of its logical partitioning. The data in cells are viewed as a byte sequence defined in groups of BSUs. For files that have more than one cell, we have a two-dimensional matrix of such BSUs. Vesta's interface enables this matrix to be dynamically and logically partitioned, which indicates how the BSUs should be distributed among the compute nodes. Each partition is called a *subfile*. The open call includes parameters that define a logical partitioning scheme and returns a file descriptor that allows access to a single subfile, not to the whole file. Not only does this logical partitioning provide a useful means of specifying data distribution, it allows significant performance gains since it guarantees that each portion of the file will be accessed by only a single processor. This guarantee reduces the need for communication and synchronization between the nodes.

Another unique feature of Vesta is its ability to checkpoint files. Checkpointing is supported by maintaining two versions of each Vesta file: the active version and the checkpoint version. By using a copy-on-write mechanism, only physical blocks that differ are copied, the others are shared by both versions of the file reducing the disk space needed.

Vesta is the basis of the current AIX PIOFS parallel I/O file system available for the IBM SP2. PIOFS has an interface that is very similar to the Vesta interface, and provides a default sequential view of each file compliant with traditional Unix.

Chapter 3

The CAP Computer-Aided Parallelization Tool

3.1 Introduction

This chapter describes the CAP computer-aided parallelization tool. The CAP language is a general-purpose parallel extension of C++ enabling application programmers to create separately the serial program parts and express the parallel behaviour of the program at a high-level of abstraction. This high-level parallel CAP program description specifies a macro-dataflow, i.e. a flow of data and parameters between operations running on the same or on different processors. CAP is designed to implement highly pipelined-parallel programs that are short and efficient. With a configuration map, specifying the layout of threads onto different PC's, these pipelined-parallel programs can be executed on distributed memory PC's.

This chapter is intended to give readers the necessary background of the CAP methodology and programming skills required to understand the remainder of this dissertation. Readers who want to have a more in-depth view of CAP can read its reference manual [Gennart98b].

As CAP is similar to the parallel data-driven (dataflow) computational model, Section 3.2 first presents the four computational models and discusses how CAP differs from the standard dataflow model. Section 3.3 introduces the CAP computer-aided parallelization tool methodology. Sections 3.4 to 3.8 present an in-depth view of the CAP language and describe all the available parallel constructs enabling programmers to specify at a high-level of abstraction the macro-dataflow of a program. A first simple example of a CAP program is described in Section 3.9. Sections 3.10 and 3.11 discuss how CAP addresses the issues of flow-control and load-balancing. Section 3.12 summarizes the chapter.

3.2 The CAP macro dataflow computational model

3.2.1 The four computational models

A computational model describes, at a level removed from technology and hardware details, what a computer or language can do, i.e. which primitive actions can be performed, when these actions are performed, and by which methods can data be accessed and stored. Browne [Browne84a, Browne84b] lists the following five key attributes that must be specified in a computational model that includes parallelism:

1. The *primitive units* of computation or basic actions of the computer or language (the data types and operations defined by the instruction set).
2. The *control mechanism* that selects for execution the primitive units into which the problem has been partitioned.
3. The *data mechanism* (how data in memory are accessed); definition of *address spaces* available to the computation.
4. The modes and patterns of *communication* among computers working in parallel, so that they can exchange needed information.
5. Synchronization mechanisms to ensure that this information arrives at the right time.

Almasi [Almasi94] presents in his book the four computational models, i.e. von Neumann, control–driven, data–driven, and demand–driven computational models. Figure 3-1 depicts A) the von Neumann model, showing sequential execution of instructions and the memory traffic generated by data fetching and storing operations (instruction fetching is not shown); B) the parallel control-driven (shared-memory) computation using the FORK construct to spawn parallel executions and the JOIN construct to synchronize them. FORK acts like a branch command on which both branches are taken. JOIN delays one routine until the other routine arrives. The memory references for data (not shown) are the same as in A); C) the parallel data-driven (dataflow) model where an operation is activated when all its input data (tokens) arrive. Note the absence of "variables" that correspond to cells in global memory; D) the parallel demand-driven (reduction model) model where computation is triggered by a demand for its result, as when some other instruction containing the routine "a" tries to execute; this other instruction might be suspended until its demand for "a" is satisfied. The definition of "a" inside the box is evaluated in a similar series of steps. In string reduction, each demander gets a copy of the definition for a "do-it-yourself" evaluation. In graph reduction, the evaluation is performed at the time of the first demand, and each demander is given a pointer to the result of the evaluation (a=10).

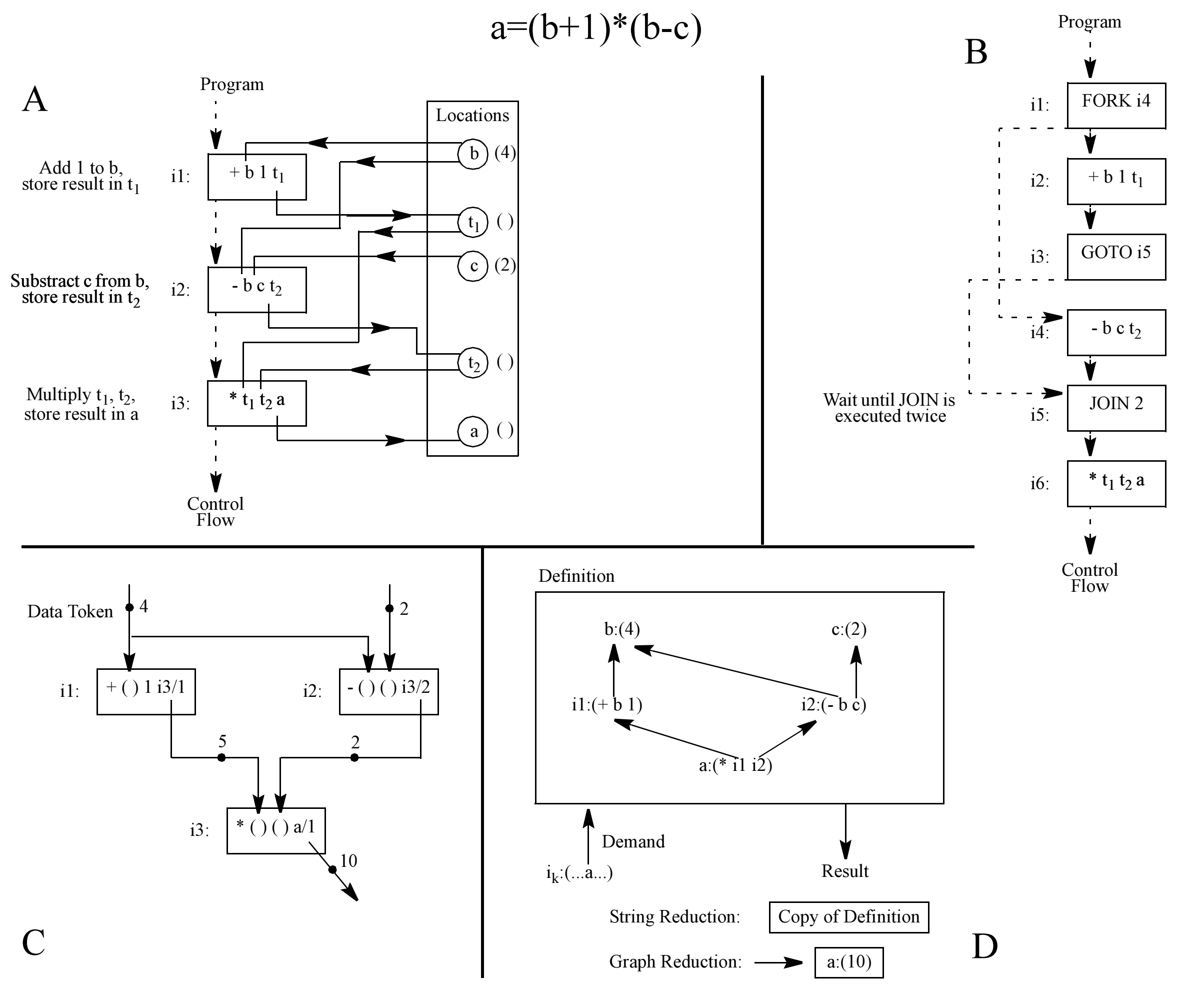


Figure 3-1. A) The von Neumann, B) the control-driven, C) the data-driven, and D) the demand-driven computational models compared on a single example, the evaluation of a=(b+1)*(b-c)

The best-known computational model is the one devised by John von Neumann and his associates years ago. The von Neumann model's features, listed in the same order used above, follow:

1. A processor that performs instructions such as "Add the contents of these two registers and put the result in that register."
2. A control scheme that fetches one instruction after another from the memory for execution by the processor, and shuttles data between memory and processor one word at a time.
3. A memory that stores both the instructions and data of a program in cells having unique addresses.

3.2.2 The CAP primitive units of computation

The primitive unit of computation in CAP is any sequence of C/C++ instructions which is serially executed by a thread to perform a required high-level task[1]. Examples of high-level tasks are multiplying two matrixes, filtering an image, reading some data from a disk, writing some data to a disk, displaying a 24-bit image on the screen. These high-level tasks are defined by a single input data, a single output data, and the sequence of C/C++ instructions that generates the output data from the input data. Of course, these high-level tasks may have side-effects such as modifying shared global variables.

1. A task means here a sequential subprogram usable as a high-level building block for concurrent programming.

The term high-level is used to emphasize the fact that the CAP computational model works at a high-level of abstraction compared with other parallel imperative languages such as High-Performance Fortran [Loveman93], Multilisp [Halstead85], Concurrent Pascal [Hansen75], Occam [Miller88, Jones85, Inmos85] and compared with other parallel programming environments such as Express [Parasoft90], PVM [Beguelin90, Sunderam90], Linda [Carriero89a, Carriero89b], MPI [MPI94]. The history of programming languages shows a discernible trend towards higher levels of abstraction [Watt90]. Machine instructions and most single statements are examples of low-level language constructs. The procedure concept (as in C, for example) allows to treat a larger block of code as an entity that can be easily invoked. The direction in language development has been towards making a program more and more a collection of classes, with their private lives separated from their public lives, i.e object oriented programming [Ghezzi82, Ghezzi85]. We believe that is essential for a parallel programming language such as CAP to offer a high level of abstraction, i.e. to conceive high-level tasks as primitive units of computation and to provide high-level parallel constructs to easily and efficiently combine these high-level tasks. Thanks to its macro-dataflow computational model, CAP alleviates parallel programming efforts and enhances performances. With traditional parallel programming languages (Concurrent Pascal, Occam, etc.) and parallel programming environments (PVM, MPI, Linda, Express, etc.), programmers are burdened with low-level constructs for parallel execution[1]. For example, these include the *fork, join*, *parbegin*, and *parend* primitives for starting and stopping parallel execution, *pvm_send*, *pvm_receive*, *mpi_send*, and *mpi_receive* primitives, and *semaphores*, *barriers*, *rendez-vous*, and *remote procedure calls* for coordinating parallel execution. This point of view is shared with Grimshaw [Grimshaw96] who also believes that low-level abstractions require the programmer to operate at the assembly language level of parallelism. Others [Beguelin92, Hatcher91, Shu91] share our philosophy of providing a higher-level language interface to simplify applications development.

In the remainder of this dissertation, the CAP primitive units of computation are called sequential operations so as to avoid confusion with the so-called parallel CAP operations, which are parallel operations at a higher hierarchical level.

3.2.3 The CAP control mechanism

The control mechanism that selects for execution the primitive units of computation, i.e. sequential operations, is in CAP based on the macro dataflow MDF [Grimshaw93a] model inspired by the dataflow computational model [Agerwala82, Dennis75, Srini86, Veen86]. The CAP macro dataflow computational model is a medium-grain, data-driven model and differs from traditional dataflow in four ways. First, the computation granularity is larger than in traditional dataflow. *Actors*, the basic units of computation, are high-level tasks such as multiplying two matrices specified in a high-level language (C/C++ sequential operation in the CAP terminology) not primitive operations such as addition. Second, some actors may maintain state between invocations, i.e. have side-effects such as modifying shared global variables. Third, actors may only depend on the result of their single previous actor, i.e. single data dependency. Finally, in order to be able to have parallel executions of tasks, CAP has introduced two particular actors called *split* and *merge* routines. A split routine takes as input the token of its previous actor, and splits it into several sub-tokens sent in a pipelined parallel manner to the next actors. A merge routine collects the results and acts as a synchronization means terminating its execution and passing its token to the next actor after the arrival of all sub-results.

An algorithm is described in CAP by its macro dataflow and graphically depicted by a directed acyclic graph DAG. Directed acyclic graphs are completely general, this means that they can describe any sort of algorithm. Furthermore, they ensure that the generated code will be deadlocks free. However at the time of this research, CAP was able to only specify symmetric DAG's, i.e. where all the split and merge points in the graph match pairwise (Figure 3-2).

A recent extension to CAP (not described in this dissertation since not available during the research) enables the programmer to specify asymmetric DAG's as well, i.e. where split and merge points in the graph do not match pairwise (Figure 3-3).

1. In order to build a parallel program, there must be constructs [Ghezzi85] to: (a) *Define* a set of subtasks to be executed in parallel; (b) *Start and stop* their execution; (c) *Coordinate* and specify their interaction while they are executing.

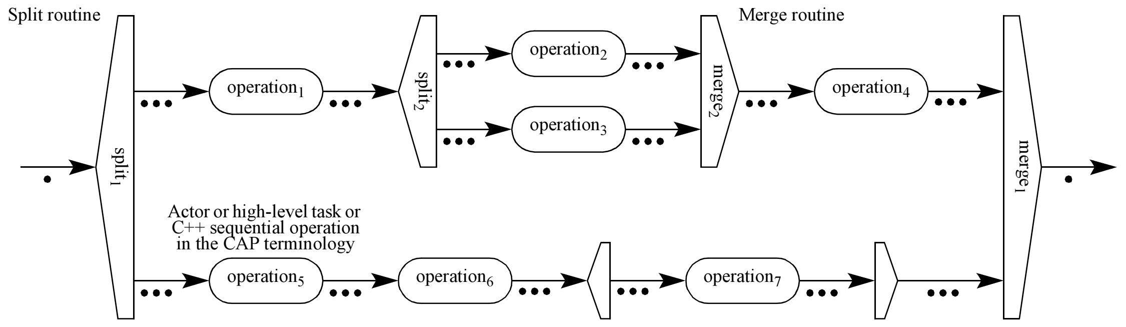


Figure 3-2. An example of a CAP macro dataflow depicted by a symmetric directed acyclic graph. Arcs model data dependencies between actors. Tokens carry data along these arcs. Split routines split input tokens into several sub-tokens sent in a pipelined parallel manner. Merge routines merge input tokens into one output token thus acting as synchronization points.

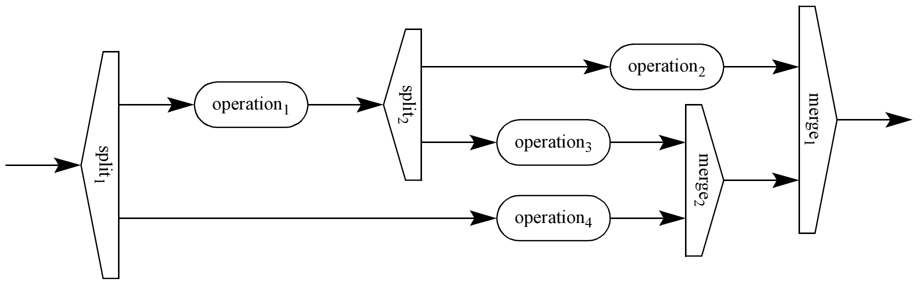


Figure 3-3. An asymmetric directed acyclic graph that can be described with the recent extension to CAP

3.2.4 The CAP communication mechanism and synchronization mechanisms

The CAP language does not explicitly provide data transfer mechanisms such as a *send* or *receive* primitive. Communications, i.e. transfer of data among the address spaces, are automatically deducted by the CAP preprocessor and runtime system based on the CAP specification of the macro dataflow and the mapping of sequential operations onto the threads available for computation. The CAP paradigm ensures that data transfer, i.e. token motion, occurs only at the end of the execution of operations in order to redirect the output token of an operation to the input of the next operation in the macro dataflow. By managing automatically the communications without programmer intervention, the task of writing parallel programs is simplified.

Regarding communications, the CAP language does not explicitly provide synchronization tools such as semaphores, barriers, etc. A CAP program is self-synchronized by the data merging operations. Merge routines act as synchronization points returning their output tokens only when all the sub-tokens have been merged.

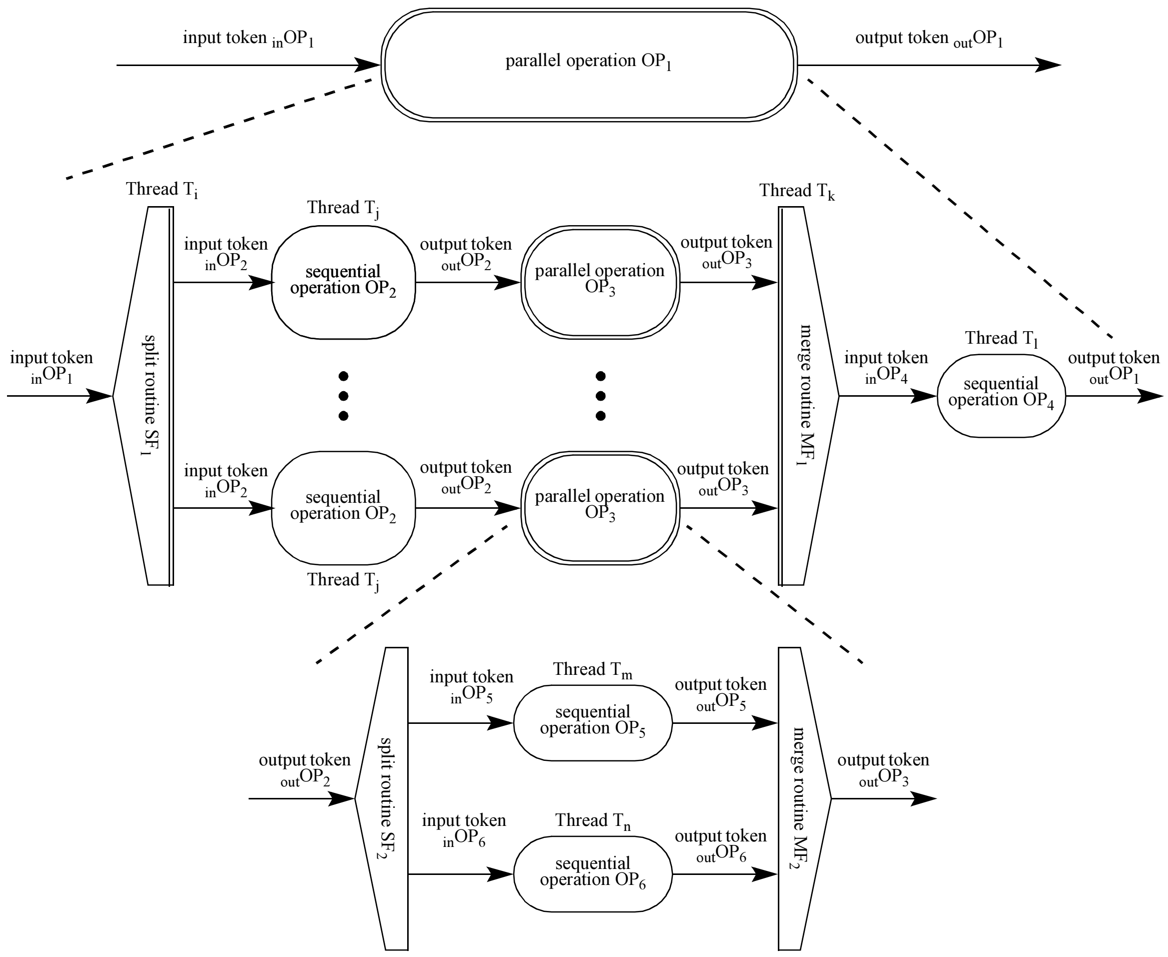
3.3 The CAP computer-aided parallelization tool philosophy

The CAP computer-aided parallelization tool enables application programmers to specify at a high level of abstraction the set of threads, which are present in the application, the processing operations offered by these threads, and the flow of data and parameters between operations. This specification completely defines how operations running on the same or on different PC's are sequenced and what data and parameters each operation receives as input values and produces as output values.

The CAP methodology consists of dividing a complex operation into several suboperations with data dependencies, and to assign each of the suboperations to a thread in the thread hierarchy. The CAP programmer specifies in CAP the data dependencies between the suboperations, and assigns explicitly each suboperation to a thread. The CAP C/C++ preprocessor automatically generates parallel code that implements the required synchronizations and communications to satisfy the data dependencies specified by the user. CAP also handles for a large part memory management and communication protocols, freeing the programmer from low level issues.

CAP operations are defined by a single input, a single output, and the computation that generates the output from the input. Input and output of operations are called *tokens* and are defined as C++ classes with serialization routines that enable the tokens to be packed or serialized, transferred across the network, and unpacked or deserialized. Communication occurs only when the output token of an operation is transferred to the input of another operation. The CAP's runtime system ensures that tokens are transferred from one address space to another in a completely asynchronous manner (socket-based communication over TCP/IP). This ensures that communication takes place at the same time as computation[1].

An operation specified in CAP as a schedule of suboperations is called a *parallel operation.* A parallel operation specifies the assignment of suboperations to threads, and the data dependencies between suboperations. When two consecutive operations are assigned to different threads, the tokens are *redirected* from one thread to the other. As a result, parallel operations also specify communications and synchronizations between s*equential operations*. A sequential operation, specified as a C/C++ routine, computes its output based on its input. A sequential operation cannot incorporate any communication, but it may compute variables which are global to its thread.



input token $_{in}OP_1$
parallel operation OP_1
output token $_{out}OP_1$
Thread T_i
Thread T_j
Thread T_k
input token $_{in}OP_2$
sequential operation OP_2
output token $_{out}OP_2$
parallel operation OP_3
output token $_{out}OP_3$
input token $_{in}OP_1$
split routine SF_1
merge routine MF_1
input token $_{in}OP_4$
Thread T_l
sequential operation OP_4
output token $_{out}OP_1$
input token $_{in}OP_2$
sequential operation OP_2
output token $_{out}OP_2$
parallel operation OP_3
output token $_{out}OP_3$
Thread T_j
input token $_{in}OP_5$
Thread T_m
sequential operation OP_5
output token $_{out}OP_5$
output token $_{out}OP_2$
split routine SF_2
merge routine MF_2
output token $_{out}OP_3$
input token $_{in}OP_6$
Thread T_n
sequential operation OP_6
output token $_{out}OP_6$

Figure 3-4. Graphical CAP specification: parallel operations are displayed as parallel horizontal branches, pipelined operations are operations located in the same horizontal branch

Each parallel CAP construct consists of a split routine splitting an input request into sub-requests sent in a pipelined parallel manner to the operations of the available threads and of a merging function collecting the results. The merging function also acts as a synchronization means terminating the parallel CAP construct's execution and passing its result to the next operation after the arrival of all sub-results (Figure 3-4).

1. In the case of a mono-processor PC communications are only partially hidden, since the TCP/IP protocol stack requires considerable processing power (Section 4.5).

The CAP specification of a parallel program is described in a simple formal language, an extension of C++. This specification is translated into a C++ source program, which, after compilation, runs on multiple processors according to a configuration map specifying the mapping of the threads running the operations onto the set of available processors. The macro data flow model which underlies the CAP approach has also been used successfully by the creators of the MENTAT parallel programming language [Grimshaw93a, Grimshaw93b].

3.4 Tokens

In CAP data that flow through operations are called tokens since a CAP program is self-synchronized by its data motion. A token declaration is similar to the C/C++ struct/class declaration. Along with a token declaration, the programmer must provide serialization routines necessary for moving the token from one address space to another, i.e. packing the token in a structure that is easily and efficiently sent through a TCP/IP connection by the CAP's runtime system and unpacking the same structure in the other address space (Section 4.4.1). Serialization occurs only when a token has to be transferred from one address space to another. Within an address space, CAP's runtime system uses the shared memory to move tokens from one thread to another.

Program 3-1 shows an example of 4 token declarations, *TokenAT* (lines 1-8), *TokenBT* (lines 11-19), *TokenCT* (lines 20-24) and *TokenDT* (lines 30-37). Serialization routines are not shown since Section 4.4.1 is completely devoted to this issue. The declaration of a token consists of the keyword '*token*' (lines 1, 11, 20 and 30) followed by any C/C++ struct/class field declarations (lines 3, 4, 6, 13, etc.). As C++ classes, tokens may contain constructors and a destructor (line 6, 16, 17 and 35). One can even declare MFC objects inside a token (lines 13 and 14) as long as serialization routines are provided. However CAP does not support pointers since it does not know how to serialize them. That is the reason of lines 27 and 28.

```
 1  token TokenAT
 2  {
 3    int Value;
 4    float NumberOfDegree;
 5
 6    TokenAT(int value, float numberOfDegree);
 7    ... // Any C++ field declarations
 8  }; // end token TokenAT
 9
10
11  token TokenBT
12  {
13    CString AStringOfChars;
14    CArray<float, int> AnArrayOfFloat;
15
16    TokenBT(char* initialCountry);
17    ~TokenBT();
18    ... // Any C++ field declarations
19  }; // end token TokenBT
20  token TokenCT
21  {
22    MyOwnClassT AnObject;
23    ... // Any C++ field declarations
24  }; // end token TokenCT
25
26
27  typedef char* PointerToCharT;
28  typedef double* PointerToDoubleT;
29
30  token TokenDT
31  {
32    PointerToCharT APointerToCharP;
33    PointerToDoubleT APointerToDoubleP;
34
35    TokenDT(TokenAT* fromP);
36    ... // Any C++ field declarations
37  }; // end token TokenDT
```

Program 3-1. 4 token declarations in CAP

The pure C/C++ program corresponding to the 4 token declarations in CAP (Program 3-1) is shown in Program 3-2.

```
 1  class TokenAT
 2  {
 3    int Value;
 4    float NumberOfDegree;
 5
 6    TokenAT(int value, float numberOfDegree);
 7    ... // Any C++ field declarations
 8  }; // end class TokenAT
 9
10
11  class TokenBT
12  {
13    CString AStringOfChars;
14    CArray<float, int> AnArrayOfFloat;
15
16    TokenBT(char* initialCountry);
17    ~TokenBT();
18    ... // Any C++ field declarations
19  }; // end class TokenBT
20  class TokenCT
21  {
22    MyOwnClassT AnObject;
23    ... // Any C++ field declarations
24  }; // end token TokenCT
25
26
27  typedef char* PointerToCharT;
28  typedef double* PointerToDoubleT;
29
30  class TokenDT
31  {
32    PointerToCharT APointerToCharP;
33    PointerToDoubleT APointerToDoubleP;
34
35    TokenDT(TokenAT* fromP);
36    ... // Any C++ field declarations
37  }; // end class TokenDT
```

Program 3-2. 4 token declaration in C/C++

3.5 Process hierarchy

As presented in section 3.3, the fundamental CAP methodology consists of specifying at a high-level of abstraction a process hierarchy, the operations offered by the processes in the hierarchy, and for parallel operations the schedule of suboperations described by a macro dataflow depicted as a directed acyclic graph (DAG).

Program 3-3 shows a process hierarchy declaration where 5 types of processes are defined, *ProcessAT* (lines 1-16), *ProcessBT* (lines 19-30), *ThreadAT* (lines 33-42), *ThreadBT* (lines 45-54), *ThreadCT* (lines 57-66), *ThreadDT* (lines 69-78), *ThreadET* (lines 81-90). Processes are declared as C++ classes with the '*process*' keyword (lines 1, 19, 33, etc.). Note that there are processes with subprocess declarations, *ProcessAT* and *ProcessBT*, and processes without subprocess declarations, *ThreadAT*, *ThreadBT*, *ThreadCT*, *ThreadDT* and *ThreadET*. CAP makes a significant distinction between these two types of declaration. Therefore in the continuation of this dissertation the term *hierarchical process* refers to processes with subprocesses defined in a 'subprocesses:' section, the term *leaf process* or *thread* refers to processes without a '*subprocesses:*' section, and the term *process* refers to either hierarchical or leaf processes. In a '*subprocesses:*' section programmers instantiate processes, e.g. the hierarchical process *ProcessAT* has 3 subprocesses: a *ProcessB* hierarchical process of type *ProcessBT* (line 4), a *ThreadA* leaf process of type *ThreadAT* (line 5) and a *ThreadB* leaf process of type *ThreadBT* (line 6). In the same manner *ProcessB* is hierarchically defined.

```
 1 process ProcessAT
 2 {
 3 subprocesses:
 4   ProcessBT ProcessB;
 5   ThreadAT ThreadA;
 6   ThreadBT ThreadB;
 7
 8 operations:
 9   Operation1
10     in TokenAT* InputP
11     out TokenDT* OutputP;
12
13   Operation2
14     in TokenAT* InputP
15     out TokenDT* OutputP;
16 }; // end process ProcessAT
17
18
19 process ProcessBT
20 {
21 subprocesses:
22   ThreadCT ThreadC;
23   ThreadDT ThreadD;
24   ThreadET ThreadE;
25
26 operations:
27   Operation1
28     in TokenCT* InputP
29     out TokenDT* OutputP;
30 } // end process ProcessBT
31
32
33 process ThreadAT
34 {
35 variables:
36   int ThreadLocalStorage;
37
38 operations:
39   Operation1
40     in TokenAT* InputP
41     out TokenBT* OutputP;
42 }; // end process ThreadAT
43
44
45 process ThreadBT
46 {
47 variables:
48   int ThreadLocalStorage;
49
50 operations:
51   Operation1
52     in TokenBT* InputP
53     out TokenCT* OutputP;
54 }; // end process ThreadBT
55
56
57 process ThreadCT
58 {
59 variables:
60   int ThreadLocalStorage;
61
62 operations:
63   Operation1
64     in TokenCT* InputP
65     out TokenCT* OutputP;
66 }; // end process ThreadCT
67
68
69 process ThreadDT
70 {
71 variables:
72   int ThreadLocalStorage;
73
74 operations:
75   Operation1
76     in TokenCT* InputP
77     out TokenCT* OutputP;
78 }; // end process ThreadDT
79
80
81 process ThreadET
82 {
83 variables:
84   int ThreadLocalStorage;
85
86 operations:
87   Operation1
88     in TokenCT* InputP
89     out TokenDT* OutputP;
90 }; // end process ThreadET
91
92
93 ProcessAT MyHierarchy;
```

Program 3-3. CAP specification of a process hierarchy

In a hierarchical process, *high-level operations* also called *parallel operations* are defined as a schedule of suboperations (either parallel or sequential suboperations) offered by its subprocesses and/or offered by himself (lines 9-11, 13-15 and 27-29). The flow of tokens between operations, i.e. macro dataflow, is programmed by combining the 8 high-level CAP constructs described in Section 3.8.

In leaf processes, operations are defined as standard sequential C/C++ subprograms (lines 39-41, 51-53, 63-65, 75-77 and 87-89), henceforth called *leaf operations* or *sequential operations*. The term *operation* is used to refer to either parallel or sequential operations.

Both parallel and sequential operations take a single input token, e.g. lines 10 and 40, and produce a single output token, e.g. lines 11 and 41. The input token is in the parallel case redirected to the first operation met when flowing through the macro dataflow, and in the sequential case it is the input parameter of the function. Alternatively, the output token is in the first case the output of the last executed operation in the dataflow and in the second case the result of the serial execution of the C/C++ function.

At line 93 the process hierarchy is instantiated. At run time and with a configuration file (Section 3.6), *MyHierarchy* process hierarchy is created on a multi-PC environment[1], i.e. only the leaf processes are spawn since they are the threads executing the sequential operations they offer. Hierarchical processes are merely entities for grouping operations in a hierarchical manner. They do not participate as threads during execution since the parallel operations represent exclusively schedules used by leaf processes at the end of a sequential operation to locate the next sequential operation, i.e. *successor*, to fire. In other words, tokens flow from sequential operations to sequential operations guided by parallel operations[2]. Therefore at run time behind the *MyHierarchy* process hierarchy, there are 5 threads of execution, i.e. *ThreadA*, *ThreadB*, *ThreadC*, *ThreadD*, *ThreadE*. These threads are distributed among the PC's according to a configuration file.

In leaf processes it is possible to declare thread local variables[3] in 'variables:' sections (lines 36, 48, 60, 72, 84), i.e. variables that are distinct across different thread instantiations. Hierarchical processes may also contain a 'variables:' section but care must be taken if the leaf subprocesses are mapped by the configuration file onto different address spaces. In that case a local copy of the variables present in the 'variables:' section is available in each address space. CAP's runtime system does not ensure any coherence between address spaces. Each of them gets a local copy of all the variables, i.e. the global variables, the thread local variables and the hierarchical process local variables.

The execution model of *MyHierarchy* CAP process hierarchy is shown in Figure 3-5. Each threads in the process hierarchy executes a loop consisting of 1) removing a token from its input queue; 2) selecting the sequential operation to execute based on the current parallel operation; 3) running the sequential operation to produce an output token; 4) finding out the successor, i.e. the next sequential operation to be executed by a thread, and sending asynchronously the output token to that thread using the message passing system.

The C/C++ program corresponding to the CAP process hierarchy declaration (Program 3-3) is shown in Program 3-4. Processes, either hierarchical or leaf processes, are merely declared as C++ classes (lines 1, 18, 32, 44, 56, 68 and 80), subprocesses are declared as members of their parents (lines 4, 5, 6, 21, 22 and 23), and operations, either parallel or sequential operations, are methods within processes (lines 9-10, 12-13, 26-27, 38-39, 50-51, 62-63, 74-75, 86-87). The operation prototype always features two arguments. The first argument is the pointer to the input token and the second argument is the pointer's reference to the output token so as the output token created inside the operation is returned back to the caller. Process hierarchy instantiation is done in the same manner as with a CAP process hierarchy (line 92).

3.6 Configuration file

In order to be able to run a CAP program on multiple address spaces, i.e. with several Windows NT processes distributed on a multi-PC environment, CAP's runtime system needs a configuration file. A configuration file is a text file specifying: 1) the number of address spaces (or Windows NT processes) participating in the parallel computation; 2) the PC's IP addresses on which these Windows NT processes run; 3) the Windows NT process executable filenames; 4) the mapping of threads to address spaces.

Program 3-5 gives an example of a configuration file for the *MyHierarchy* process hierarchy shown in Program 3-3.

1. In the case where the configuration file is omitted, all the threads are spawned in a single address space.
2. In addition, split and merge sequential functions enable scattering and gathering tokens (Sections 3.8.5, 3.8.6 and 3.8.7).
3. This is equivalent to the thread local storage (TLS) used in Win32 program [Cohen98].

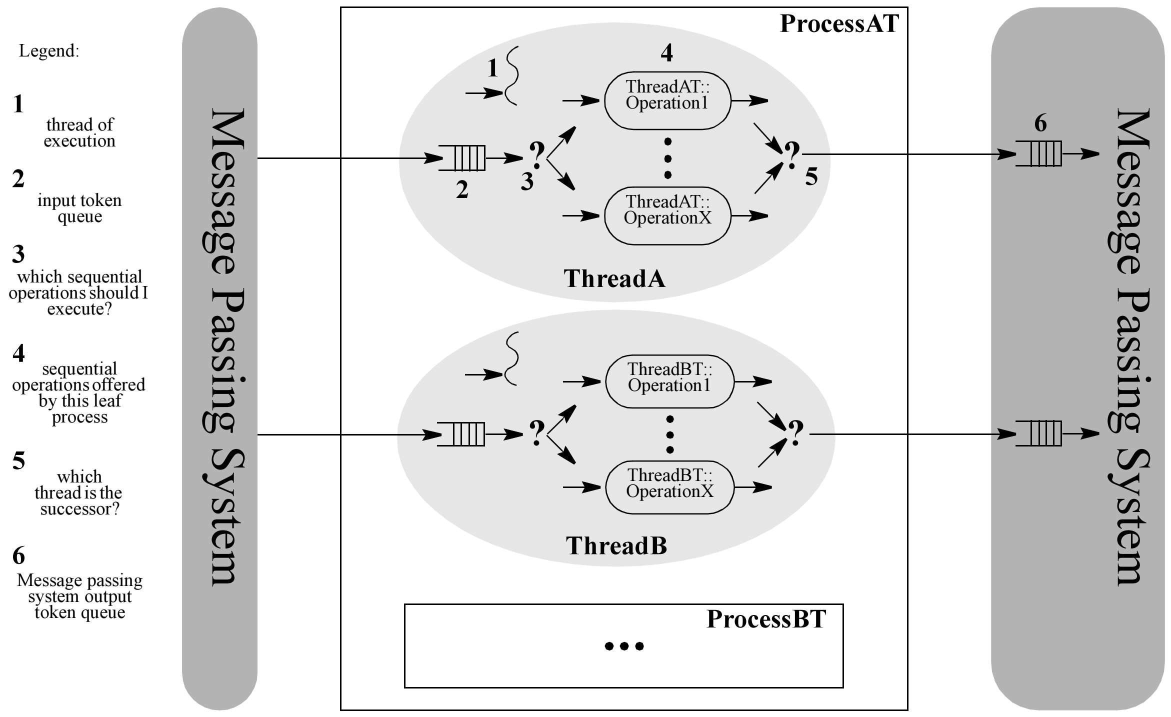


Figure 3-5. Graphical representation of *MyHierarchy* CAP process hierarchy

```
class ProcessAT
{
private:
  ProcessBT ProcessB;
  ThreadAT ThreadA;
  ThreadBT ThreadB;

public:
  void Operation1(TokenAT* InputP,
                  TokenDT* &OutputP);

  void Operation2(TokenAT* InputP,
                  TokenDT* &OutputP);

}; // end class ProcessAT


class ProcessBT
{
private:
  ThreadCT ThreadC;
  ThreadDT ThreadD;
  ThreadET ThreadE;

public:
  void Operation1(TokenCT* InputP,
                  TokenDT* &OutputP);

}; // end class ProcessBT


class ThreadAT
{
private:
  int ThreadLocalStorage;

public:
  void Operation1(TokenAT* InputP,
                  TokenBT* &OutputP);

}; // end class ThreadAT


class ThreadBT
{
private:
  int ThreadLocalStorage;

public:
  void Operation1(TokenBT* InputP,
                  TokenCT* &OutputP);

}; // end class ThreadBT


class ThreadCT
{
private:
  int ThreadLocalStorage;

public:
  void Operation1(TokenCT* InputP,
                  TokenCT* &OutputP);

}; // end class ThreadCT


class ThreadDT
{
private:
  int ThreadLocalStorage;

public:
  void Operation1(TokenCT* InputP,
                  TokenCT* &OutputP);

}; // end class ThreadDT


class ThreadET
{
private:
  int ThreadLocalStorage;

public:
  void Operation1(TokenCT* InputP,
                  TokenDT* &OutputP);

}; // end class ThreadET


ProcessAT MyHierarchy;
```

Program 3-4. C/C++ specification of a process hierarchy

```
 1 configuration {
 2 processes:
 3   A ( "user" ) ;
 4   B ( "128.178.75.65", "\\FileServer\SharedFiles\CapExecutable.exe" ) ;
 5   C ( "128.178.75.66", "\\FileServer\SharedFiles\CapExecutable.exe" ) ;
 6   D ( "128.178.75.67", "\\FileServer\SharedFiles\CapExecutable.exe" ) ;
 7
 8 threads:
 9   "ThreadA" (C) ;
10   "ThreadB" (B) ;
11   "ProcessB.ThreadC" (D) ;
12   "ProcessB.ThreadD" (A) ;
13   "ProcessB.ThreadE" (C) ;
14 }; // end of configuration file
```

Program 3-5. Configuration file declaring 4 address spaces, the PC's addresses where these 4 Windows NT processes run, the 3 executable filenames and the mapping of the 5 threads to the 4 address spaces

A configuration file always contains two sections:

1. A '*processes:*' section (lines 2-6) where all the address spaces or Windows NT processes participating in the parallel computation are listed. For each of them a PC's IP address and an executable filename are given so as the runtime system is able to spawn the Windows NT process on that PC with that executable file (lines 4, 5, 6). Note the special keyword '*user*' (line 3) used to refer the Windows NT process launched by the user at DOS prompt for starting the CAP program (Program 3-6). In that case, the PC's IP address and the executable file name is obviously not mentioned as it is the user who launches it.
2. A '*threads:*' section (lines 8-13) specifying for each threads in *MyHierarchy* process hierarchy the Windows NT process where the thread executes.

Program 3-6 shows how to start a CAP program with a configuration file. At the beginning of the execution the CAP's runtime system parses the configuration file and thanks to a message passing system (Section 4.4) spawns all the Windows NT processes (except the 'user' one, Program 3-5 line 3) on the mentioned PC's. Then the 4 Windows NT processes (A, B, C, and D) parse the configuration file so as to spawn the 5 threads in their respective address space, e.g. the Windows NT process *C* spawns the *ThreadA* thread and the *ProcessB.ThreadE* thread.

```
128.178.75.67> CapExecutable.exe -cnf \\FileServer\SharedFiles\ConfigurationFile.txt ...↵
```

Program 3-6. Starting a CAP program with a configuration file at DOS prompt

If the configuration file is omitted when starting a CAP program (Program 3-6), then CAP's runtime system spawns all the threads in the current Windows NT process.

3.7 CAP operations

After having shown in Section 3.5 how to declare a process hierarchy and the operations offered by the processes in the hierarchy, this section looks at how to implement sequential operations (Section 3.7.1) and parallel operations (Section 3.7.2).

CAP enables the programmer to declare an operation, either a parallel operation or a sequential operation, outside its process interface. This feature is extremely useful for extending the functionalities of existing CAP programs, e.g. the parallel storage and processing server PS^2 (Chapter 6). Instead of declaring the operation inside a given process, the programmer merely declares its interface globally, using the keyword '*operation*' (Program 3-7).

```
operation ProcessBT::Operation2  // Additional parallel operation declaration
  in TokenDT* InputP
  out TokenBT* OutputP;

leaf operation ThreadET::Operation2  // Additional sequential operation declaration
  in TokenBT* InputP
  out TokenAT* OutputP;
```

Program 3-7. Additional operation declarations

CAP allows the programmer to call a CAP operation within a C/C++ program or library using the '*call*' keyword (Program 3-8, line 8). It is the programmer's responsibility to create the input token (line 5) and to delete the output token (line 10). The '*call*' instruction is synchronous, i.e. the thread that performs the '*call*' instruction is blocked until the called operation completes.

```
 1  ProcessAT MyHierarchy;
 2
 3  int main(int argc, char* argv[])
 4  {
 5    TokenAT* InputP = new TokenAT(2, 4.562);
 6    TokenDT* OutputP;
 7
 8    call MyHierarchy.Operation1 in InputP out OutputP;
 9    printf("Result = %s\n"), OutputP->APointerToCharP);
10    delete OutputP;
11
12    return 0;
13  } // end main
```

Program 3-8. Synchronous call of a CAP operation from a sequential C/C++ program

CAP also provides an asynchronous '*start*' instruction where the thread that performs the '*start*' instruction is not blocked and may synchronize itself with the *capWaitTerminate* CAP-library function.

```
ProcessAT MyHierarchy;

int main(int argc, char* argv[])
{
  capCallRequestT* CallRequestP;
  TokenAT* InputP = new TokenAT(2, 4.562);
  TokenDT* OutputP;

  start MyHierarchy.Operation1 in InputP out NothingP return CallRequestP;

  ...

  OutputP = capWaitTerminate(CallRequestP);
  printf("Result = %s\n"), OutputP->APointerToCharP);
  delete OutputP;

  return 0;
} // end main
```

Program 3-9. Asynchronous call of a CAP operation from a sequential C/C++ program

3.7.1 Sequential operations

As mentioned in section 3.2.2, CAP's primitive units of computation are leaf operations or sequential operations, i.e. C++ subprograms usable as building blocks for concurrent programming. A sequential operation is defined by a single input token, a single output token, and the C/C++ function body that generates the output from the input.

In our directed acyclic graph formalism (Figure 3-2), we depict sequential operations as a single rounded rectangle with an input arrow with the input token's type, an output arrow with the output token's type, and the thread which performs this sequential operation (Figure 3-6).

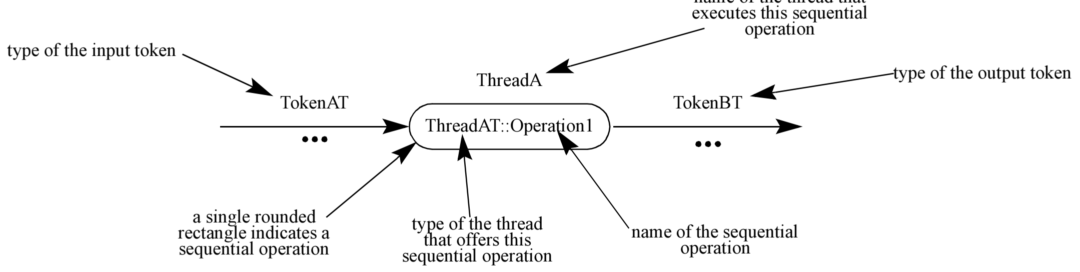

name of the thread that executes this sequential operation
type of the input token
ThreadA
type of the output token
TokenAT
TokenBT
ThreadAT::Operation1
a single rounded rectangle indicates a sequential operation
type of the thread that offers this sequential operation
name of the sequential operation

Figure 3-6. A sequential operation with its input and output token. Single rounded rectangles depict sequential operations.

Program 3-10 shows the implementation of the *ThreadAT::Operation1* sequential operation using the '*leaf operation*' CAP construct.

```
1 leaf operation ThreadAT::Operation1
2   in TokenAT* InputP
3   out TokenBT* OutputP
4 {
5   ... // Any C/C++ statements
6   OutputP = new TokenBT("Switzerland");
7   ... // Any C/C++ statements
8 } // end ThreadAT::Operation1
```

Program 3-10. CAP specification of a sequential operation. Note the '*leaf*' keyword.

It is the responsibility of the sequential operation to create the output token (line 6) using one of the defined constructors (Program 3-1). Once the sequential operation is completed, by default, the CAP's runtime system deletes the input token. A call to the *capDoNotDeleteInputToken* CAP-library function inside a sequential operation tells the CAP runtime system not to delete the input token of the sequential operation.

A sequential operation may have side-effects, i.e. modifying shared global variables or thread local variables, so as to exchange information between threads in a same address space. It's the programmer's responsibility to ensure the coherence of the shared data by using appropriate synchronization mechanisms, e.g. mutexes, semaphores, barriers, provided by the CAP runtime library. Care must be taken when using these synchronization tools in order to avoid deadlocks. Indeed, CAP ensures that parallel programs are deadlock free by specifying macro dataflows as directed acyclic graphs. However, if additional synchronizations outside CAP are used, deadlock free behaviour cannot be guaranteed any more.

By default, when a sequential operation terminates, the CAP runtime system calls the successor (see Section 3.5), i.e. the next sequential operation specified by the DAG. CAP enables the programmer to prevent the CAP runtime system to call the successor, by calling the *capDoNotCallSuccessor* CAP-library function in the body of the sequential operation. The effect of the *capDoNotCallSuccessor* CAP-library function is to suspend the execution of the schedule of this particular token. To resume the execution of a suspended token, CAP supplies the *capCallSuccessor* CAP-library function. It is the programmer's responsibility to keep track of the suspended tokens, e.g. in having a global list of suspended tokens. A typical place to use this feature is when the sequential operation uses asynchronous system calls, e.g. the *ReadFile* Win32 system call. When the sequential operation finishes, the thread is able to execute other sequential operations while the OS is asynchronously doing the system call. When the system call completes, the callback routine resumes the schedule of the suspended token by calling the *capCallSuccessor* CAP-library function (Program 5-18).

3.7.2 Parallel operations

As said in Section 3.5, a parallel operation, i.e. a hierarchically higher-level operation, is defined by an input token, an output token, and a schedule of suboperations that generates the output from the input. Parallel operations are possibly executed in parallel in the case of a parallel hardware environment, i.e. a cluster of PC's.

In a directed acyclic graph (Figure 3-2), parallel operations are depicted as a double rounded rectangle with an input arrow with the input token's type, and an output arrow with the output token's type (Figure 3-7).

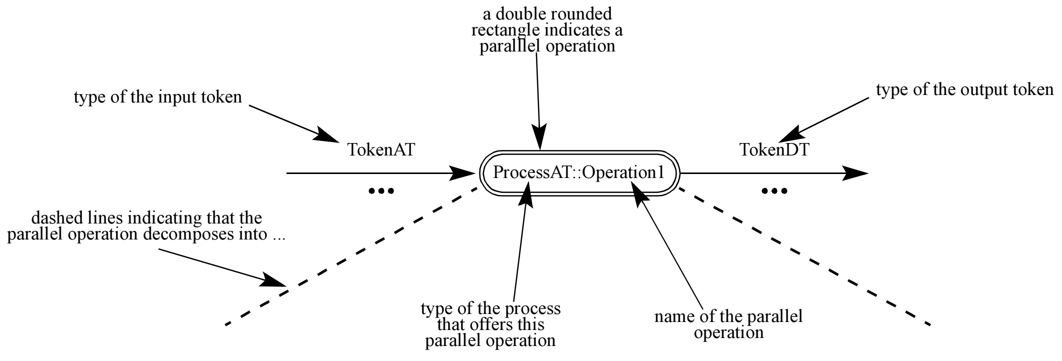

a double rounded rectangle indicates a parallel operation
type of the input token
type of the output token
TokenAT
ProcessAT::Operation1
TokenDT
dashed lines indicating that the parallel operation decomposes into ...
type of the process that offers this parallel operation
name of the parallel operation

Figure 3-7. A parallel operation with its input and output token. Note double rounded rectangle depict parallel operations.

Program 3-11 shows the implementation of the *ProcessAT::Operation1* parallel operation using the '*operation*' CAP construct. In order to specify the content of the parallel operation, i.e. to build the schedule of suboperations (line 5), the programmer may use one or several parallel CAP constructs described in Section 3.8. C/C++ statements are strictly forbidden since a parallel operation only describes the order of execution of suboperations.

```
1 operation ProcessAT::Operation1
2   in TokenAT* InputP
3   out TokenDT* OutputP
4 {
5   ... // One or several parallel CAP constructs
6 } // end ProcessAT::Operation1
```

Program 3-11. CAP specification of a parallel operation

3.8 Parallel CAP constructs

This section looks at the 8 parallel CAP constructs, i.e. *pipeline* (Section 3.8.1), *if* and *ifelse* (Section 3.8.2), *for* (Section 3.8.4), *while* (Section 3.8.3), *parallel* (Section 3.8.5), *parallel while* (Section 3.8.6), and *indexed parallel* (Section 3.8.7) CAP constructs. They are used as building blocks for specifying the macro dataflow of parallel operations, i.e. the schedule of the underlying sequential operations. These 8 high-level parallel CAP constructs are automatically translated into a C/C++ source program, which, after compilation, runs on a multi-PC environment according to a configuration file specifying the mapping of the threads running the sequential operations onto the set of available Windows NT processes.

For each of the 8 parallel CAP constructs, its graphical specification, i.e its DAG, and its CAP specification are shown. In order to clarify the schedule of suboperations specified by a parallel CAP construct, each parallel CAP construct is also given as a pure C/C++ specification corresponding to the serialized schedule.

A question that may arise when reading this section is who executes parallel operations, i.e. who evaluates the boolean expression in a *if* (Section 3.8.2), *ifelse* (Section 3.8.2) and *while* (Section 3.8.3) CAP construct, who executes the three expressions (init expression, boolean expression, and increment expression) in a *for* CAP construct (Section 3.8.4), who executes the split functions in a *parallel* (Section 3.8.5) and *parallel while* (Section 3.8.6) CAP construct, and who executes the three expressions (init expression, boolean expression, and increment expression) and the split function in an *indexed parallel* CAP construct (Section 3.8.7). The question of who executes a sequential operation is simple: the thread specified by the programmer executes the sequential operation.

The question of who executes parallel operation is much more subtle [Gennart98b]. The job of a parallel operation is to redirect tokens from their producing sequential suboperation, i.e. the suboperation that generates it, to the consuming sequential suboperation, i.e. the suboperation that consumes it. The producing suboperation is not necessarily executed by the same thread as the consuming suboperation. If the producing thread is not in the same address space as the consuming thread, the token must be transferred from one address space to the other, a costly operation that should be performed only when explicitly required by the programmer. Therefore in the current implementation of the CAP runtime system, the producing thread performs the parallel operation in order to decide himself where to redirect the token he produced. For example, the split functions in the *parallel*, *parallel while* and *indexed parallel* CAP constructs are always executed by the thread who produced the input token of the parallel construct.

3.8.1 The *pipeline* CAP construct

The *pipeline* CAP construct enables the output of one operation to be redirected to the input of another. It is the basic CAP construct for combining two operations. Figure 3-8 shows the DAG of the *pipeline* construct where 3 operations are connected in pipeline, i.e. *ThreadAT::Operation1*, *ThreadBT::Operation1* and *ProcessBT::Operation1* operations. The output of the *ThreadAT::Operation1* sequential operation is redirected to the input of the *ThreadBT::Operation1* sequential operation whose output is redirected to the first sequential operation[1] met when flowing through the DAG of the *ProcessBT::Operation1* parallel operation.

1. This is called the successor of the sequential operation and it is not necessary a sequential operation as mentioned in Section 3.5.

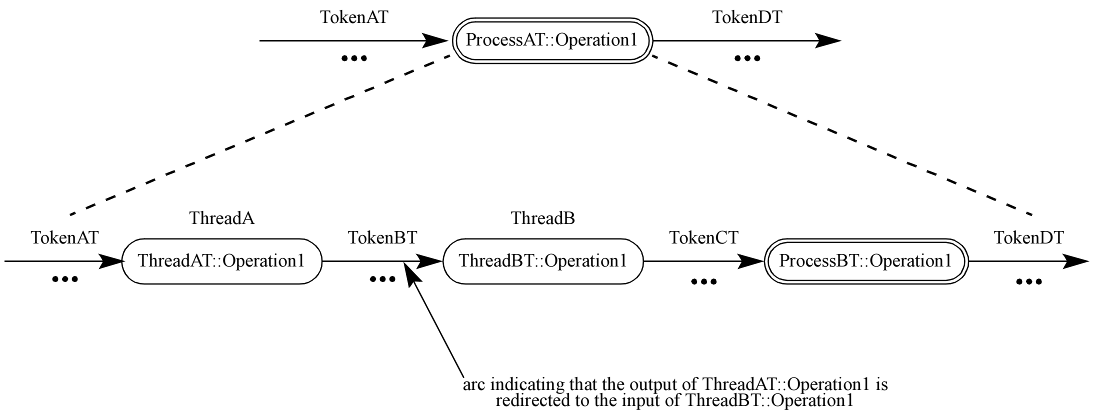


Figure 3-8. Graphical CAP specification of the *pipeline* construct

The CAP specification of the DAG in Figure 3-8 is shown in Program 3-12. The output of one operation is redirected to the input of another operation using the '>->' CAP construct (lines 6 and 8).

```
 1 operation ProcessAT::Operation1
 2   in TokenAT* InputP
 3   out TokenDT* OutputP
 4 {
 5   ThreadA.Operation1
 6   >->
 7   ThreadB.Operation1
 8   >->
 9   ProcessB.Operation1;
10 } // end ProcessAT::Operation1
```

Program 3-12. CAP specification of the *pipeline* construct

The execution of Program 3-12 strongly depends on the configuration file, i.e. whether the threads are mapped onto different processors or not. If the *ThreadA* thread, the *ThreadB* thread and the threads running the *ProcessB::Operation1* parallel operations are mapped onto different processors and if the first operation in the horizontal branch, i.e. *ThreadAT::Operation1* sequential operation, is fed with several tokens, then the 3 operations are executed in a pipelined manner like an assembly line (Figure 3-9).

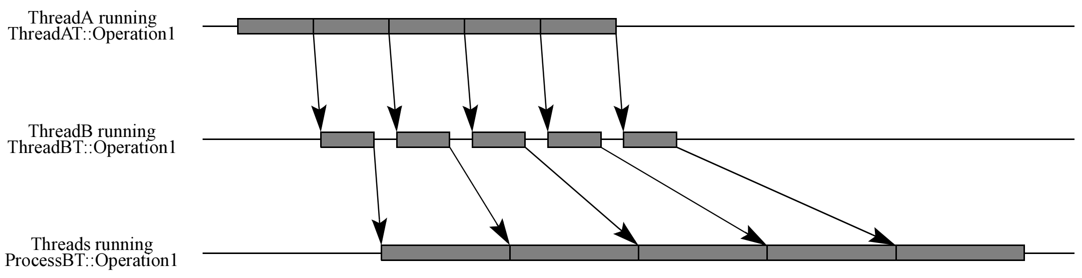

ThreadA running ThreadAT::Operation1
ThreadB running ThreadBT::Operation1
Threads running ProcessBT::Operation1

Figure 3-9. Timing diagram of the execution of the 3 operations in a pipelined manner

Supposing now that the *ProcessAT* hierarchical process declaration contains a pool of *ThreadAT* threads (Program 3-13, line 5) and a pool of *ThreadBT* threads (Program 3-13, line 6) instead of *ThreadA* and *ThreadB* threads (Program 3-3, line 5 and 6), then in the CAP specification of *ProcessAT::Operation1* parallel operation the programmer must select within the 2 pools which threads are running *ThreadAT::Operation1* and *ThreadBT::Operation1* sequential operations (Program 3-14, lines 5 and 7).

The CAP runtime system provides programmers with the *thisTokenP* variable pointing to the token about to enter a CAP construct, e.g. line 5 *thisTokenP* refers to *ProcessAT::Operation1* input token and line 7 refers to *ThreadAT::Operation1* output token. The *thisTokenP* variable enables tokens to be dynamically redirected according to their values. If the configuration file maps all the threads to different processors, then Program 3-14

```
process ProcessAT
{
subprocesses:
  ProcessBT ProcessB;
  ThreadAT ThreadA[];
  ThreadBT ThreadB[];
  ...
```

Program 3-13. Declaring two pools of threads within a hierarchical process

```
 1 operation ProcessAT::Operation1
 2   in TokenAT* InputP
 3   out TokenDT* OutputP
 4 {
 5   ThreadA[thisTokenP->ThreadAIndex].Operation1
 6   >->
 7   ThreadB[thisTokenP->ThreadBIndex].Operation1
 8   >->
 9   ProcessB.Operation1;
10 } // end ProcessAT::Operation1
```

Program 3-14. Selecting a thread within a pool

is executed in a pipelined parallel manner, i.e. *ThreadAT::Operation1*, *ThreadBT::Operation1* and *ProcessBT::Operation1* operations are executed in pipeline while *ThreadAT::Operation1* and *ThreadBT::Operation1* operations are executed in parallel by the threads in the two pools (Figure 3-10).

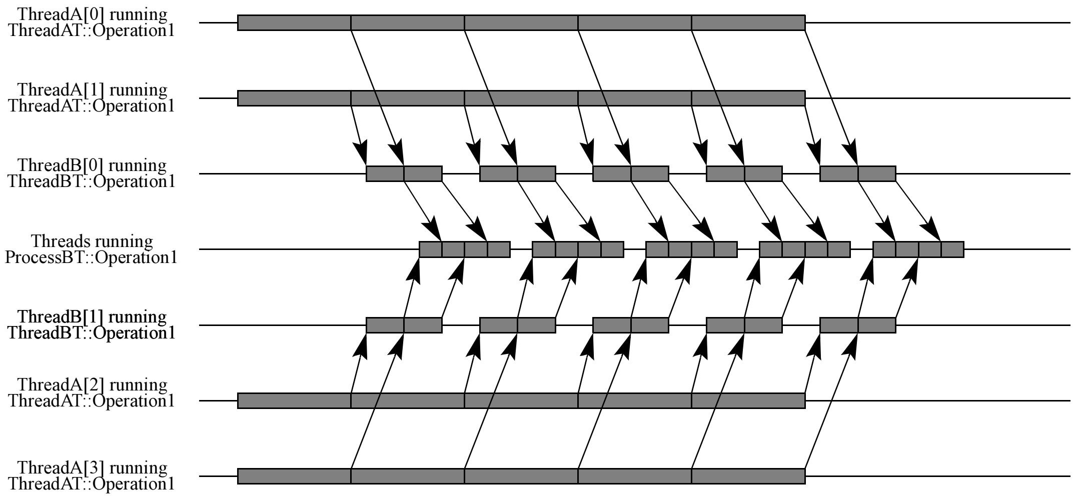

ThreadA[0] running ThreadAT::Operation1
ThreadA[1] running ThreadAT::Operation1
ThreadB[0] running ThreadBT::Operation1
Threads running ProcessBT::Operation1
ThreadB[1] running ThreadBT::Operation1
ThreadA[2] running ThreadAT::Operation1
ThreadA[3] running ThreadAT::Operation1

Figure 3-10. Timing diagram of a pipelined parallel execution

In order to express this pipelined parallel execution, the DAG of *ProcessAT::Operation1* hierarchical operation is modified so as to include two parallel branches (Figure 3-11).

Selecting a thread within a pool raises the issue of load balancing in CAP (Program 3-14, lines 5 and 7). Supposing that the execution time of *ThreadAT::Operation1* depends on the values in its input token and *ThreadA[]* threads are selected in a round-robin fashion (Program 3-14, line 5), then the execution flow may be unbalanced, i.e. the loads of the different *ThreadA[]* threads[1] may be unbalanced, and therefore decrease performances. This issue is further discussed in Section 3.11.

The equivalent serialized version of the *pipeline* CAP construct (Program 3-12) is shown in Program 3-15. This construct consists of (1) declaring the output token – lines 7, 14 and 21; (2) calling the operation – lines 8, 15 and 22; (3) deleting the input token – lines 9, 16 and 23; (4) assigning the output token to *thisTokenP* token– lines 10, 17 and 24.

1. The load of a thread is defined here as the percentage of elapsed time that this thread is executing sequential operations.

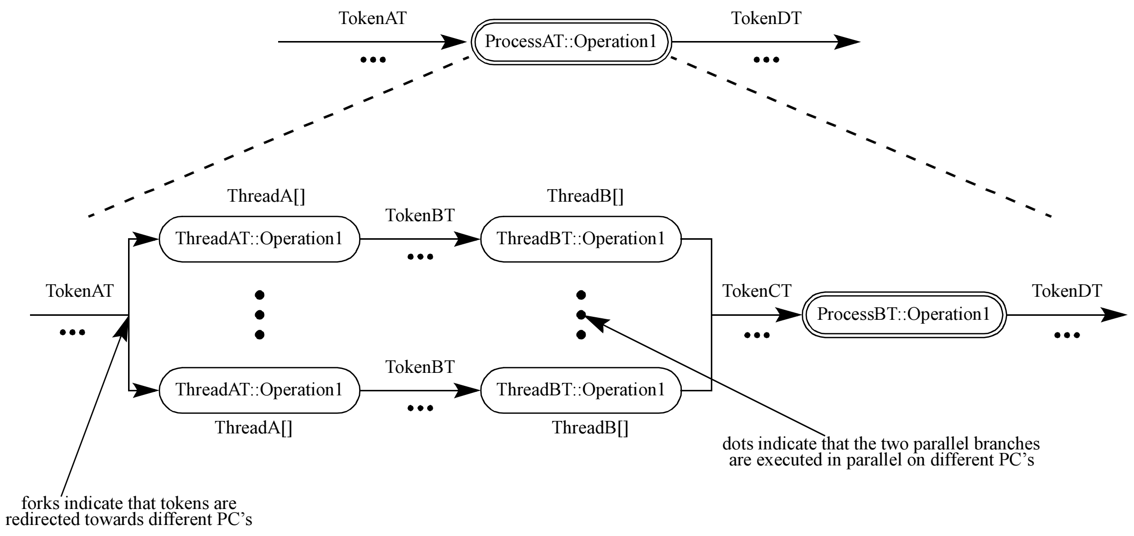


Figure 3-11. Graphical representation of a pipelined parallel execution

```
void ProcessAT::Operation1(TokenAT* InputP,
                           TokenDT* &OutputP)
{
  void* thisTokenP;

  { // begin pipeline construct
    void* OutputP;
    ThreadA.Operation1((TokenAT*) thisTokenP, (TokenBT*&) OutputP);
    delete (TokenAT*) thisTokenP;
    thisTokenP = OutputP;
  } // end pipeline construct

  { // begin pipeline construct
    void* OutputP;
    ThreadB.Operation1((TokenBT*) thisTokenP, (TokenCT*&) OutputP);
    delete (TokenBT*) thisTokenP;
    thisTokenP = OutputP;
  } // end pipeline construct

  { // begin pipeline construct
    void* OutputP;
    ProcessB.Operation1((TokenCT*) thisTokenP, (TokenDT*&) OutputP);
    delete (TokenCT*) thisTokenP;
    thisTokenP = OutputP;
  } // end pipeline construct

  OutputP = (TokenDT*) thisTokenP;
} // end ProcessA::Operation1
```

Program 3-15. Equivalent C/C++ specification of the *pipeline* construct

3.8.2 The *if* and *ifelse* CAP construct

The *if* and *ifelse* CAP constructs enable the output of one operation to be redirected to the input of another according to the result of the evaluation of a C/C++ boolean expression. Figure 3-12 shows the DAG of the *if* construct where *ProcessBT::Operation1* input token is redirected either to *ThreadCT::Operation1* sequential operation or to *ThreadET::Operation1* sequential operation according to the result of the evaluation of the C/C++ boolean expression. Note that the subconstruct output token type must be equivalent to the *if* construct input token type.

The CAP specification corresponding to the DAG in Figure 3-12 is shown in Program 3-16. The *if* CAP construct is similar to the *if* C/C++ construct but instead of braces '{}', the subconstruct must be parenthesized (lines 7-9).

The equivalent serialized version of the *if* CAP construct (Program 3-16) is shown in Program 3-17. The *if* CAP construct is similar to the *if* C/C++ expression.

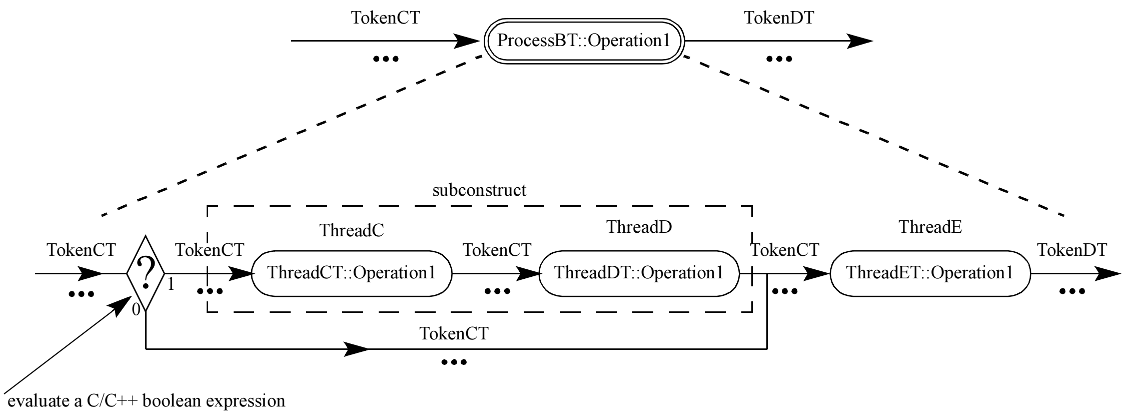


Figure 3-12. Graphical CAP specification of the *if* construct

```
operation ProcessBT::Operation1
  in TokenCT* InputP
  out TokenDT* OutputP
{
  if ( BooleanExpression )
  (                                          subconstruct
    ThreadC.Operation1
    >->
    ThreadD.Operation1
  ) // end if
  >->
  ThreadE.Operation1;
} // end ProcessBT::Operation1
```

Program 3-16. CAP specification of the *if* construct

```
void ProcessBT::Operation1(TokenCT* InputP,
                           TokenDT* &OutputP)
{
  void* thisTokenP;

  { // begin if construct
    if ( BooleanExpression )
    {
      { // begin pipeline construct
        void* OutputP;
        ThreadC.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct

      { // begin pipeline construct
        void* OutputP;
        ThreadD.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct
    } // end if                                                    subconstruct
  } // end if construct

  { // begin pipeline construct
    void* OutputP;
    ThreadE.Operation1((TokenCT*) thisTokenP, (TokenDT*&) OutputP);
    delete (TokenCT*) thisTokenP;
    thisTokenP = OutputP;
  } // end pipeline construct

  OutputP = (TokenDT*) thisTokenP;
} // end ProcessB::Operation1
```

Program 3-17. Equivalent C/C++ specification of the *if* construct

Figure 3-13 shows the DAG of the *ifelse* CAP construct where the *ProcessBT::Operation1* input token is redirected either to the *ThreadCT::Operation1* sequential operation or to the *ThreadDT::Operation1* sequential operation according to the result of the evaluation of the C/C++ boolean expression. Note that the input and output token type of the true-subconstruct and false-subconstruct must be equivalent.

The CAP specification corresponding to the DAG in Figure 3-13 is shown in Program 3-18.

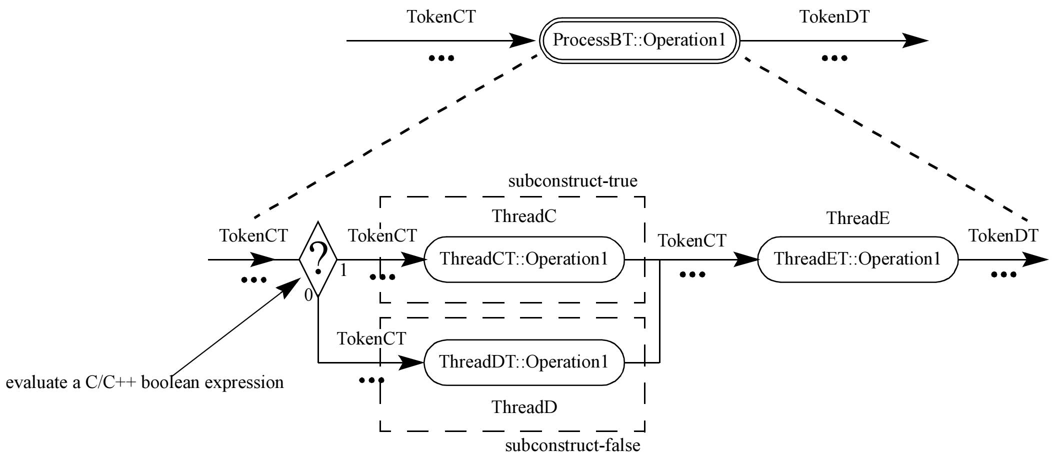


Figure 3-13. Graphical CAP specification of the *ifelse* construct

```
operation ProcessBT::Operation1
  in TokenCT* InputP
  out TokenDT* OutputP
{
  ifelse ( BooleanExpression )
  (                                              subconstruct-true
    ThreadC.Operation1
  )
  (
    ThreadD.Operation1
  ) // end ifelse                                subconstruct-false
  >->
  ThreadE.Operation1;
} // end ProcessBT::Operation1
```

Program 3-18. CAP specification of the *ifelse* construct

The equivalent serialized version of the *ifelse* CAP construct (Program 3-18) is shown in Program 3-19.

```
void ProcessBT::Operation1(TokenCT* InputP,
                           TokenDT* &OutputP)
{
  void* thisTokenP;

  { // begin ifelse construct
    if ( BooleanExpression )
    {                                                                  subconstruct-true
      { // begin pipeline construct
        void* OutputP;
        ThreadC.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct
    } else {
      { // begin pipeline construct
        void* OutputP;
        ThreadD.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct
    } // end ifelse                                                    subconstruct-false
  } // end ifelse construct

  { // begin pipeline construct
    void* OutputP;
    ThreadE.Operation1((TokenCT*) thisTokenP, (TokenDT*&) OutputP);
    delete (TokenCT*) thisTokenP;
    thisTokenP = OutputP;
  } // end pipeline construct

  OutputP = (TokenDT*) thisTokenP;
} // end ProcessB::Operation1
```

Program 3-19. Equivalent C/C++ specification of the *ifelse* construct

3.8.3 The *while* CAP construct

The *while* construct is the first iterative CAP construct. It iterates a CAP subconstruct while the result of the evaluation of a C/C++ boolean expression is true, i.e. at the end of the CAP subconstruct a C/C++ boolean expression is evaluated and if its result is true then the subconstruct output token becomes the subconstruct input token. Figure 3-14 shows the DAG of the *while* CAP construct. Note that the subconstruct input and output token type must be equivalent to the *while* construct input token type.

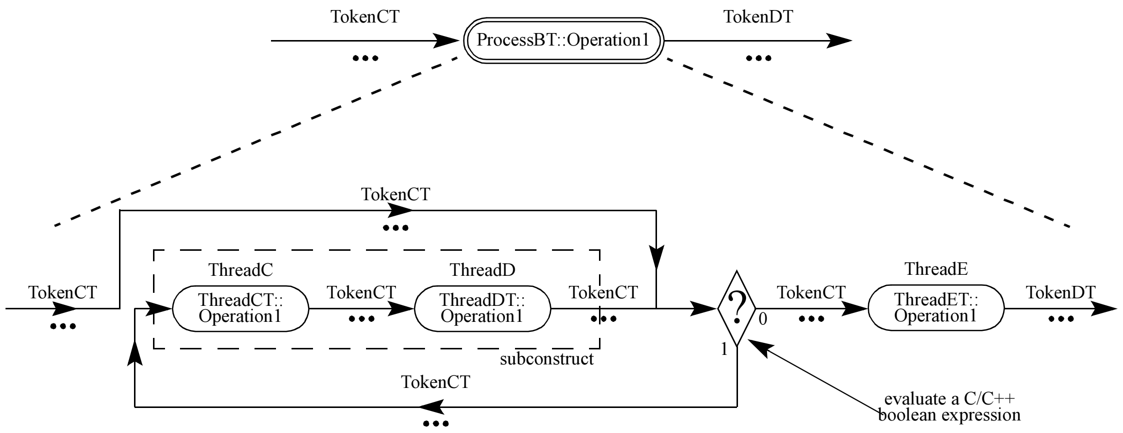


Figure 3-14. Graphical CAP specification of the *while* construct

The CAP specification of the DAG in Figure 3-14 is shown in Program 3-20. The *while* CAP construct is similar to the *while* C/C++ expression but instead of braces '{}', the subconstruct must be parenthesized (lines 7-9).

```
 1 operation ProcessBT::Operation1
 2   in TokenCT* InputP
 3   out TokenDT* OutputP
 4 {
 5   while ( BooleanExpression )
 6   (                                              subconstruct
 7     ThreadC.Operation1
 8     >->
 9     ThreadD.Operation1
10   ) // end while
11   >->
12   ThreadE.Operation1;
13 } // end ProcessBT::Operation1
```

Program 3-20. CAP specification of the *while* construct

The equivalent serialized version of the *while* CAP construct (Program 3-20) is shown in Program 3-21.

3.8.4 The *for* CAP construct

The *for* construct enables a CAP subconstruct to be iterated while the result of the evaluation of a C/C++ boolean expression is true. The difference with the *while* CAP construct resides in the fact that a variable is first initialized at the beginning of the loop, and at the end of the CAP subconstruct this counter modified, a C/C++ boolean expression is evaluated and if its result is true then the subconstruct output token becomes the subconstruct input token. Figure 3-15 shows the DAG of the *for* construct. Note that the subconstruct input and output token type must be equivalent to the *for* construct input token type.

The CAP specification of the DAG in Figure 3-15 is shown in Program 3-22. The *for* CAP construct is similar to the *for* C/C++ construct but instead of braces '{}', the subconstruct must be parenthesized (lines 7-9).

The equivalent serialized version of the *for* CAP construct (Program 3-22) is shown in Program 3-23.

```
void ProcessBT::Operation1(TokenCT* InputP,
                           TokenDT* &OutputP)
{
  void* thisTokenP = InputP;

  { // begin while construct
    while ( BooleanExpression )
    {
      { // begin pipeline construct
        void* OutputP;
        ThreadC.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct

      { // begin pipeline construct
        void* OutputP;
        ThreadD.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct
    } // end while                                                          subconstruct
  } // end while construct

  { // begin pipeline construct
    void* OutputP;
    ThreadE.Operation1((TokenCT*) thisTokenP, (TokenDT*&) OutputP);
    delete (TokenCT*) thisTokenP;
    thisTokenP = OutputP;
  } // end pipeline construct

  OutputP = (TokenDT*) thisTokenP;
} // end ProcessB::Operation1
```

Program 3-21. Equivalent C/C++ specification of the *while* construct

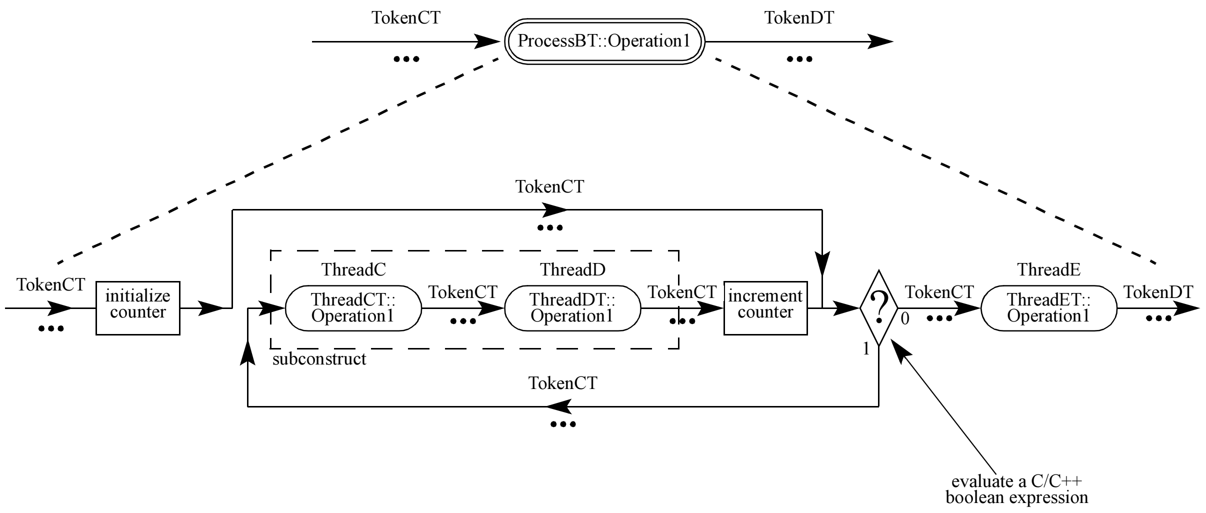


Figure 3-15. Graphical CAP specification of the *for* construct

```
operation ProcessBT::Operation1
  in TokenCT* InputP
  out TokenDT* OutputP
{
  for ( InitExpression; BooleanExpression; IncrementExpression )
  (
    ThreadC.Operation1
    >->
    ThreadD.Operation1
  ) // end for                                                subconstruct
  >->
  ThreadE.Operation1;
} // end ProcessBT::Operation1
```

Program 3-22. CAP specification of the *for* construct

```
void ProcessBT::Operation1(TokenCT* InputP,
                           TokenDT* &OutputP)
{
  void* thisTokenP = InputP;

  { // begin for construct
    for ( InitExpression; BooleanExpression; IncrementExpression )
    {
      { // begin pipeline construct
        void* OutputP;
        ThreadC.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct

      { // begin pipeline construct
        void* OutputP;
        ThreadD.Operation1((TokenCT*) thisTokenP, (TokenCT*&) OutputP);
        delete (TokenCT*) thisTokenP;
        thisTokenP = OutputP;
      } // end pipeline construct
    } // end for
  } // end for construct

  { // begin pipeline construct
    void* OutputP;
    ThreadE.Operation1((TokenCT*) thisTokenP, (TokenDT*&) OutputP);
    delete (TokenCT*) thisTokenP;
    thisTokenP = OutputP;
  } // end pipeline construct

  OutputP = (TokenDT*) thisTokenP;
} // end ProcessB::Operation1
```

subconstruct

Program 3-23. Equivalent C/C++ specification of the *for* construct

3.8.5 The *parallel* CAP construct

The *parallel* CAP construct is the first split-merge construct. It segments an input token into several subtokens, performs different operations in parallel on each of the subtokens, and merges the result of the operations.

Figure 3-16 shows the DAG of the *parallel* CAP construct. The *parallel* construct input token (*TokenAT*) is divided into 3 subtokens by the *Split1*, *Split2* and *Split3* split functions. Each generated subtokens (*TokenAT*, *TokenBT* and *TokenCT*) is then directed to its *parallel* body subconstruct, i.e. subconstruct-1, subconstruct-2 or subconstruct-3. The output of the *parallel* body subconstructs (*TokenBT*, *TokenCT* and *TokenDT*) are merged into the *parallel* construct output token using *Merge1*, *Merge2* and *Merge3* merge functions by *ThreadB* thread. When all the subtokens are merged, the *parallel* construct output token (*TokenDT*) is redirected to its successor. Depending whether different threads on different processors are selected for computation or not, the 3 *parallel* body subconstructs may actually execute in parallel.

The CAP specification of the DAG in Figure 3-16 is shown in Program 3-24. The *parallel* CAP construct consists of the keyword '*parallel*', the two construct initialization parameters and a list of parallel bodies. The two initialization parameters are the name of the thread (*ThreadB*) who merges the output of the *parallel* body subconstructs into the *parallel* construct output token (*Out1*) and the output token declaration (line 41). Note the keyword '*remote*' which indicates that the *parallel* construct input token (*thisTokenP*) is sent to *ThreadB* thread in order to initialize the *parallel* construct output token in *ThreadB* address space. If, instead, the keyword '*local*' would have been used, the *parallel* construct output token would have been initialized in the current address space, i.e. the address space of the thread executing the *parallel* CAP construct, and sent in the *ThreadB* address space. Depending on their sizes, the programmer can choose to transfer either the *thisTokenP* or the *Out1* token from the current address space to *ThreadB* address space. A parallel body consist of a split function (lines 44, 49 and 54), a *parallel* body subconstruct (lines 45, 50 and 55), and a merge function (lines 46, 51, 56). A split function is a sequential C/C++ routine that creates a subtoken from an input token (lines 1-6, 8-13 and 15-20). A merge function is a sequential C/C++ routine that merges a subtoken into an output token (lines 22-25, 27-30 and 32-35).

The equivalent serialized version of the *parallel* CAP construct (Program 3-24) is shown in Program 3-25. It consists of sequentially calling the *parallel* body subconstructs (lines 48-54, 59, 65 and 70-76) parenthesized by the split functions (lines 47, 58and 69) and the merge functions (lines 55, 66 and 77).

The *parallel* CAP construct with a single branch can be used when incorporating an operation which has not the appropriate interface (i.e. the appropriate input and output token types[1]) into the schedule of a parallel operation (Program 3-26). Indeed, this design pattern enables the programmer to adapt the interface of the incorporated

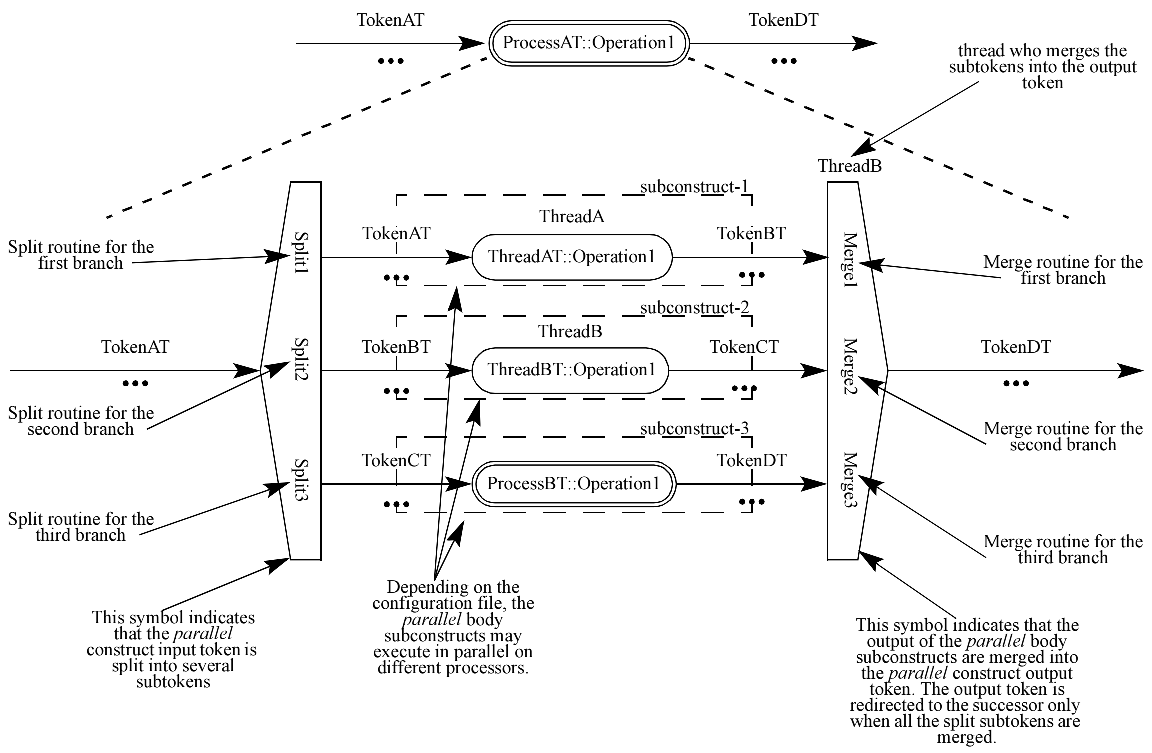

TokenAT
ProcessAT::Operation1
TokenDT
thread who merges the subtokens into the output token
ThreadB
subconstruct-1
ThreadA
TokenAT
ThreadAT::Operation1
TokenBT
Split routine for the first branch
Split1
Merge1
Merge routine for the first branch
subconstruct-2
ThreadB
TokenAT
TokenBT
ThreadBT::Operation1
TokenCT
Split2
Merge2
TokenDT
Split routine for the second branch
Merge routine for the second branch
subconstruct-3
TokenCT
ProcessBT::Operation1
TokenDT
Split3
Merge3
Split routine for the third branch
Merge routine for the third branch
This symbol indicates that the parallel construct input token is split into several subtokens
Depending on the configuration file, the parallel body subconstructs may execute in parallel on different processors.
This symbol indicates that the output of the parallel body subconstructs are merged into the parallel construct output token. The output token is redirected to the successor only when all the split subtokens are merged.

Figure 3-16. Graphical CAP specification of the *parallel* construct

```
void Split1(TokenAT* inputP, TokenAT* &subtokenP)
{
  ... // Any C/C++ statements
  subtokenP = new TokenAT(3, 6.9806);
  ... // Any C/C++ statements
} // end Split1

void Split2(TokenAT* inputP, TokenBT* &subtokenP)
{
  ... // Any C/C++ statements
  subtokenP = new TokenBT("Italy");
  ... // Any C/C++ statements
} // end Split2

void Split3(TokenAT* inputP, TokenCT* &subtokenP)
{
  ... // Any C/C++ statements
  subtokenP = new TokenCT;
  ... // Any C/C++ statements
} // end Split3

void Merge1(TokenDT* outputP, TokenBT* subtokenP)
{
  ... // Any C/C++ statements
} // end Merge1

void Merge2(TokenDT* outputP, TokenCT* subtokenP)
{
  ... // Any C/C++ statements
} // end Merge2

void Merge3(TokenDT* outputP, TokenDT* subtokenP)
{
  ... // Any C/C++ statements
} // end Merge3

operation ProcessAT::Operation1
  in TokenAT* InputP
  out TokenDT* OutputP
{
  parallel (ThreadB, remote TokenDT Out1(thisTokenP))
  (
    (
      Split1,
      ThreadA.Operation1,
      Merge1                                subconstruct-1
    )
    (
      Split2,
      ThreadB.Operation1,
      Merge2                                subconstruct-2
    )
    (
      Split3,
      ProcessB.Operation1,
      Merge3                                subconstruct-3
    )
  ); // end parallel
} // end ProcessAT::Operation1
```

Program 3-24. CAP specification of the *parallel* construct

operation, i.e. to adapt the incorporated operation input type to the previous operation output token type (*AdaptInput* routine, line 10) and to adapt the incorporated operation output token type to the next operation input token type (*AdaptOutput* routine, line 12). Moreover, this design pattern (Program 3-26) enables the programmer to transfer possible data "through" the incorporated operation, i.e. from the *parallel* CAP construct's input token to the *parallel* CAP construct's output token, without having to modify the incorporated operation (line 7, initializes the *parallel* CAP construct's output token with the *parallel* CAP construct's input token).

1. i.e. when the incorporated operation input token type is different from the previous operation output token type or the incorporated operation output token type is different from the next operation input token type.

```
void Split1(TokenAT* inputP, TokenAT* &subtokenP)
{
  ... // Any C/C++ statements
  subtokenP = new TokenAT(3, 6.9806);
  ... // Any C/C++ statements
} // end Split1

void Split2(TokenAT* inputP, TokenBT* &subtokenP)
{
  ... // Any C/C++ statements
  subtokenP = new TokenBT("Italy");
  ... // Any C/C++ statements
} // end Split2

void Split3(TokenAT* inputP, TokenCT* &subtokenP)
{
  ... // Any C/C++ statements
  subtokenP = new TokenCT;
  ... // Any C/C++ statements
} // end Split3

void Merge1(TokenDT* outputP, TokenBT* subtokenP)
{
  ... // Any C/C++ statements
} // end Merge1

void Merge2(TokenDT* outputP, TokenCT* subtokenP)
{
  ... // Any C/C++ statements
} // end Merge2

void Merge3(TokenDT* outputP, TokenDT* subtokenP)
{
  ... // Any C/C++ statements
} // end Merge3

void ProcessAT::Operation1(TokenAT* InputP,
                           TokenDT* &OutputP)
{
  void* thisTokenP = InputP;

  { // begin parallel construct
    TokenAT* InputP = (TokenAT*) thisTokenP;
    TokenDT* OutputP =
      new TokenDT((TokenAT*) thisTokenP);

    Split1(InputP, (TokenAT*&) thisTokenP);
    { // begin pipeline construct
      void* OutputP;
      ThreadA.Operation1((TokenAT*) thisTokenP,
                         (TokenBT*&) OutputP);
      delete (TokenAT*) thisTokenP;
      thisTokenP = OutputP;
    } // end pipeline construct              subconstruct-1
    Merge1(OutputP, (TokenBT*) thisTokenP);
    delete (TokenBT*) thisTokenP;

    Split2(InputP, (TokenBT*&) thisTokenP);
    { // begin pipeline construct
      void* OutputP;
      ThreadB.Operation1((TokenBT*) thisTokenP,
                         (TokenCT*&) OutputP);
      delete (TokenBT*) thisTokenP;
      thisTokenP = OutputP;
    } // end pipeline construct              subconstruct-2
    Merge2(OutputP, (TokenCT*) thisTokenP);
    delete (TokenCT*) thisTokenP;

    Split3(InputP, (TokenCT*&) thisTokenP);
    { // begin pipeline construct
      void* OutputP;
      ProcessB.Operation1((TokenCT*) thisTokenP,
                          (TokenDT*&) OutputP);
      delete (TokenCT*) thisTokenP;
      thisTokenP = OutputP;
    } // end pipeline construct              subconstruct-3
    Merge3(OutputP, (TokenDT*) thisTokenP);
    delete (TokenDT*) thisTokenP;

    delete InputP;
    thisTokenP = OutputP;
  } // end parallel construct

  OutputP = (TokenDT*) thisTokenP;
} // end ProcessAT::Operation1
```

Program 3-25. C/C++ specification of the *parallel* construct

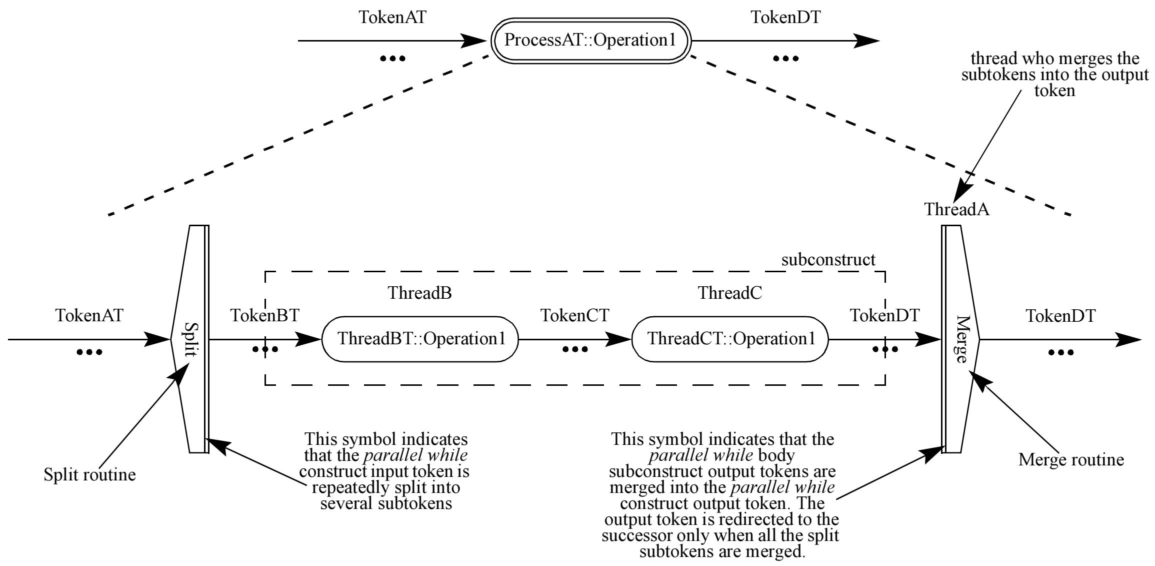
```
operation ProcessAT::Operation1
  in TokenAT* InputP
  out TokenDT* OutputP
{
  ...
  >->
  parallel (ThreadB, local TokenCT Out(thisTokenP))
  (
    (
      AdaptInput, // Adapt input token type
      ProcessB.Operation1, // Incorporated operation
      AdaptOutput // Adapt output token type
    ) // end parallel branch
  ) // end parallel
  >->
  ...
} // end processAT::Operation1
```

Program 3-26. Incorporating an operation which has not the appropriate interface into the schedule of a parallel operation using a *parallel* CAP construct with a single branch

3.8.6 The *parallel while* CAP construct

The *parallel while* CAP construct is the second split-merge construct. It iteratively divides a token into several subtokens, performs in pipeline similar operations on each of the subtokens, and merges the results of the last operation in the pipeline.

Figure 3-17 shows the DAG of the *parallel while* CAP construct. The *parallel while* construct input token (*TokenAT*) is iteratively divided into several subtokens by *Split* split function. Once a subtoken is generated, it is redirected to the *parallel while* body subconstruct which performs the operations in a pipelined manner (*ThreadBT::Operation1* and *ThreadCT::Operation1* operations). The output tokens of the *parallel while* body subconstruct are merged into the *parallel while* construct output token using the *Merge* merge function executed by the *ThreadA* thread. When all the subtokens are merged, the *parallel while* construct output token (*TokenDT*) is redirected to its successor. As already mentioned in Section 3.8.1, the operations contained in the *parallel while* body subconstruct (*ThreadBT::Operation1* and *ThreadCT::Operation1* operations) may execute in a pipelined manner (Figure 3-9, Program 3-12) or in a pipelined parallel manner (Figure 3-10, Program 3-14) depending whether different threads on different processors are selected for computation or not.



TokenAT
ProcessAT::Operation1
TokenDT
thread who merges the subtokens into the output token
ThreadA
subconstruct
ThreadB
ThreadC
TokenAT
TokenBT
ThreadBT::Operation1
TokenCT
ThreadCT::Operation1
TokenDT
TokenDT
Split
Merge
Split routine
This symbol indicates that the *parallel while* construct input token is repeatedly split into several subtokens
This symbol indicates that the *parallel while* body subconstruct output tokens are merged into the *parallel while* construct output token. The output token is redirected to the successor only when all the split subtokens are merged.
Merge routine

Figure 3-17. Graphical CAP specification of the *parallel while* construct

The CAP specification of the DAG in Figure 3-17 is shown in Program 3-27. The *parallel while* CAP construct consists of the keyword ‘parallel while’ (line 20), the four construct initialization parameters (line 21) and a *parallel while* body subconstruct (lines 23-25). The first two initialization parameters are the split function and the merge function. A split function is a sequential C/C++ routine that creates a new subtoken from the *parallel while* construct input token and the previous generated subtoken (lines 1-8). At the first iteration, a null pointer is passed as the previous subtoken (second argument). The split function is called as long as it has returned a 1 in the previous call. In other words, the split function returns 0 together with the last subtoken. A merge function is a sequential C/C++ routine that merges *parallel while* body subconstruct output tokens into the *parallel while* construct output token (lines 10-13). The last two initialization parameters are similar to those of the parallel construct (Section 3.8.5), i.e. the name of the thread (*ThreadA*) who merges the *parallel while* body subconstruct output tokens into the *parallel while* construct output token (*Out1*) and the output token declaration.

```
 1 bool Split(TokenAT* inputP, TokenBT* prevSubtokenP, TokenBT* &subtokenP)
 2 {
 3   ... // Any C/C++ statements
 4   subtokenP = new TokenBT(...);
 5   ... // Any C/C++ statements
 6
 7   return (IsNotLastSubtoken);
 8 } // end Split
 9
10 void Merge(TokenDT* outputP, TokenDT* subtokenP)
11 {
12   ... // Any C/C++ statements
13 } // end Merge
14
15
16 operation ProcessAT::Operation1
17   in TokenAT* InputP
18   out TokenDT* OutputP
19 {
20   parallel while
21     (Split, Merge, ThreadA, local TokenDT Out1(thisTokenP))
22   (
23     ThreadB.Operation1
24     >->
25     ThreadC.Operation1
26   ); // end parallel while                                      subconstruct
27 } // end ProcessAT::Operation1
```

Program 3-27. CAP specification of the *parallel while* construct

The equivalent serialized version of the *parallel while* CAP construct in Program 3-27 is shown in Program 3-28. It iteratively calls the split function (line 29), the *parallel while* body subconstruct (lines 32-44) and the merge function (line 46) while the last subtoken is not merged (line 30). Since the *parallel while* body subconstruct deletes its input subtoken (line 35) and the split function needs the previous subtoken (line 1) to generate the current subtoken, the input of the *parallel while* body subconstruct (line 34, *thisTokenP*) is always the token preceding the current generated subtoken (line 29, *NextSubtokenP*). Once a *parallel while* body subconstruct output token is merged (line 46, *thisTokenP*), it is deleted (line 47), and the current generated subtoken (line 29, *NextSubtokenP*) becomes the next *parallel while* body subconstruct input token (line 48, *thisTokenP*).

```
 1 bool Split(TokenAT* inputP, TokenBT* prevSubtokenP, TokenBT* &subtokenP)
 2 {
 3   ... // Any C/C++ statements
 4   subtokenP = new TokenBT(...);
 5   ... // Any C/C++ statements
 6
 7   return (IsNotLastSubtoken);
 8 } // end Split
 9
10 void Merge(TokenDT* outputP, TokenDT* subtokenP)
11 {
12   ... // Any C/C++ statements
13 } // end Merge
14
15
16 void ProcessAT::Operation1(TokenAT* InputP,
17                            TokenDT* &OutputP)
18 {
19   void* thisTokenP = InputP;
20
21   { // begin parallel while construct
22     TokenAT* InputP = (TokenAT*) thisTokenP;
23     TokenDT* OutputP = new TokenDT((TokenAT*) thisTokenP);
24     bool IsNotLastSubtoken;
25     TokenBT* NextSubtokenP;
26
27     IsNotLastSubtoken = (TokenBT*) Split(InputP, (TokenBT*) NULL, (TokenBT*&) thisTokenP);
28     while( IsNotLastSubtoken &&
29            IsNotLastSubtoken = Split(InputP, (TokenBT*) thisTokenP, NextSubtokenP),
30            (thisTokenP != (void*) NULL) )
31     {
32       { // begin pipeline construct
33         void* OutputP;
34         ThreadB.Operation1((TokenBT*) thisTokenP, (TokenCT*&) OutputP);
35         delete (TokenBT*) thisTokenP;
36         thisTokenP = OutputP;
37       } // end pipeline construct
38
39       { // begin pipeline construct
40         void* OutputP;
41         ThreadC.Operation1((TokenCT*) thisTokenP, (TokenDT*&) OutputP);
42         delete (TokenCT*) thisTokenP;
43         thisTokenP = OutputP;
44       } // end pipeline construct
45                                                                     subconstruct
46       Merge(OutputP, (TokenDT*) thisTokenP);
47       delete (TokenDT*) thisTokenP;
48       thisTokenP = NextSubtokenP;
49     } // end while
50
51     delete InputP;
52     thisTokenP = OutputP;
53   } // end parallel while construct
54
55   OutputP = (TokenDT*) thisTokenP;
56 } // end ProcessAT::Operation1
```

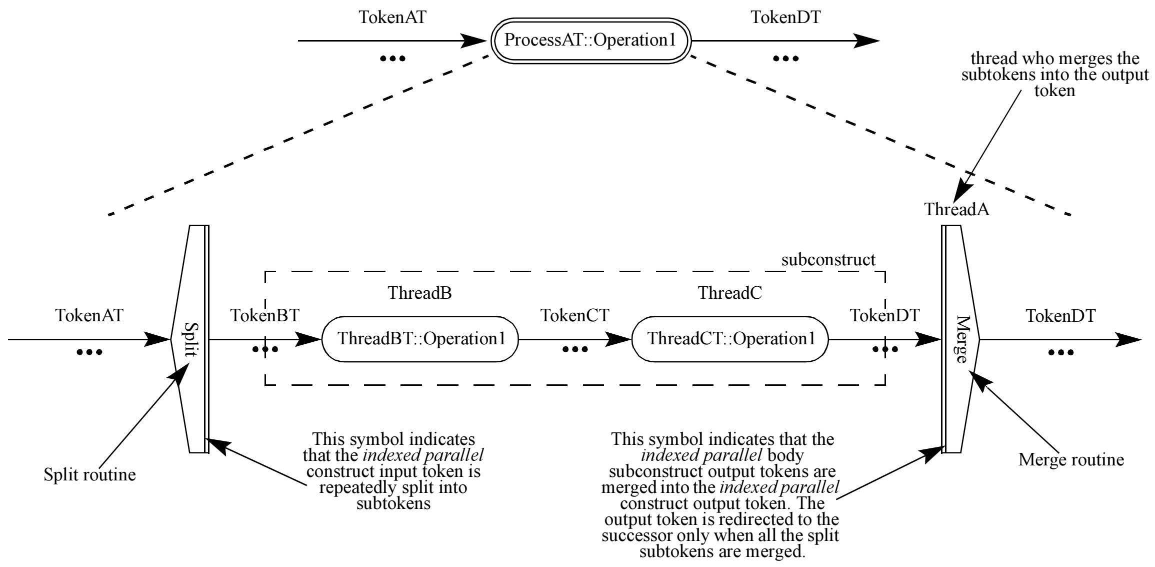
Program 3-28. Equivalent C/C++ specification of the *parallel while* construct

3.8.7 The *indexed parallel* CAP construct

The *indexed parallel* CAP construct is the third and the last split-merge construct. It is similar to the *parallel while* CAP construct (Section 3.8.6) except that the iteration is based on a C/C++ for loop not on a while loop. The *indexed parallel* CAP construct iteratively divides a token into several subtokens, performs in pipeline similar operations on each of the subtokens, and merges the results of the last operation in the pipeline.

Figure 3-18 shows the DAG of the *indexed parallel* CAP construct which is similar to the DAG of the *parallel while* CAP construct (Figure 3-17). The *indexed parallel* construct input token (*TokenAT*) is iteratively divided into several subtokens by *Split* split function. Once a subtoken is generated, it is redirected to the *indexed parallel* body subconstruct which performs the operations in a pipelined manner (*ThreadBT::Operation1* and *ThreadCT::Operation1* operations). The output tokens of the *indexed parallel* body subconstruct are merged into the *indexed parallel* construct output token using *Merge* merge function by *ThreadA* thread. When all the subtokens are merged, the *indexed parallel* construct output token (*TokenDT*) is redirected to its successor. The operations contained in the *indexed parallel* body subconstruct (*ThreadBT::Operation1* and

ThreadCT::Operation1 operations) may execute in a pipelined manner (Figure 3-9, Program 3-12) or in a pipelined parallel manner (Figure 3-10, Program 3-14) depending whether different threads on different processors are selected for computation or not.



TokenAT
ProcessAT::Operation1
TokenDT
thread who merges the subtokens into the output token
ThreadA
subconstruct
ThreadB
ThreadC
TokenAT
TokenBT
ThreadBT::Operation1
TokenCT
ThreadCT::Operation1
TokenDT
TokenDT
Split
Merge
Split routine
This symbol indicates that the indexed parallel construct input token is repeatedly split into subtokens
This symbol indicates that the indexed parallel body subconstruct output tokens are merged into the indexed parallel construct output token. The output token is redirected to the successor only when all the split subtokens are merged.
Merge routine

Figure 3-18. Graphical CAP specification of the *indexed parallel* construct

The CAP specification of the DAG in Figure 3-18 is shown in Program 3-29. The *indexed parallel* CAP construct consists of the keyword '*indexed*' followed by standard C/C++ *for* expressions (line 19), and of the keyword '*parallel*' (line 20) followed by the four construct initialization parameters (line 21) and an *indexed parallel* body subconstruct (lines 23-25). The first two initialization parameters are the split function and the merge function. A split function is a sequential C/C++ routine that creates a new subtoken from the *indexed parallel* construct input token and the current index values (lines 1-6). The split function is called for all the index values specified in the standard C/C++ *for* expressions (line 19) and may return a null subtoken to skip an iteration. A merge function is a sequential C/C++ routine that merges *indexed parallel* body subconstruct output tokens into the *indexed parallel* construct output token (lines 8-11). The last two initialization parameters are similar to those of the *parallel* and *parallel while* constructs (Sections 3.8.5 and 3.8.6), i.e. the name of the thread (*ThreadA*) who merges the *indexed parallel* body subconstruct output tokens into the *indexed parallel* construct output token (*Out1*) and the output token declaration.

```
 1 void Split(TokenAT* inputP, TokenBT* &subtokenP, int index)
 2 {
 3   ... // Any C/C++ statements
 4   subtokenP = new TokenBT(...);
 5   ... // Any C/C++ statements
 6 } // end Split
 7
 8 void Merge(TokenDT* outputP, TokenDT* subtokenP, int index)
 9 {
10   ... // Any C/C++ statements
11 } // end Merge
12
13
14 operation ProcessAT::Operation1
15   in TokenAT* InputP
16   out TokenDT* OutputP
17 {
18   indexed
19     (int Index = 0; Index < 100; Index++)
20   parallel
21     (Split, Merge, ThreadA, remote TokenDT Out1(thisTokenP))
22   (
23     ThreadB.Operation1
24     >->
25     ThreadC.Operation1
26   ); // end indexed parallel                              subconstruct
27 } // end ProcessAT::Operation1
```

Program 3-29. CAP specification of the *indexed parallel* construct

The equivalent serialized version of the *indexed parallel* CAP construct in Program 3-29 is shown in Program 3-30. It consists of C/C++ for expressions (line 23) that iteratively call the split function (line 25), the *indexed parallel* body subconstruct (lines 28-40) and the merge function (line 42). Once an *indexed parallel* body subconstruct output token is merged (line 42, *thisTokenP*), it is deleted (line 43). The test at line 26 is due to the fact that a split function may return a null subtoken to skip an iteration.

```
 1 void Split(TokenAT* inputP, TokenBT* &subtokenP, int index)
 2 {
 3   ... // Any C/C++ statements
 4   subtokenP = new TokenBT(...);
 5   ... // Any C/C++ statements
 6 } // end Split
 7
 8 void Merge(TokenDT* outputP, TokenDT* subtokenP, int index)
 9 {
10   ... // Any C/C++ statements
11 } // end Merge
12
13
14 void ProcessAT::Operation1(TokenAT* InputP,
15                            TokenDT* &OutputP)
16 {
17   void* thisTokenP = InputP;
18
19   { // begin indexed parallel construct
20     TokenAT* InputP = (TokenAT*) thisTokenP;
21     TokenDT* OutputP = new TokenDT((TokenAT*) thisTokenP);
22
23     for ( int Index = 0; Index < 100; Index++ )
24     {
25       Split(InputP, (TokenBT*) thisTokenP, Index);
26       if ( thisTokenP )
27       {
28         { // begin pipeline construct
29           void* OutputP;
30           ThreadB.Operation1((TokenBT*) thisTokenP, (TokenCT*&) OutputP);
31           delete (TokenBT*) thisTokenP;
32           thisTokenP = OutputP;
33         } // end pipeline construct
34
35         { // begin pipeline construct
36           void* OutputP;
37           ThreadC.Operation1((TokenCT*) thisTokenP, (TokenDT*&) OutputP);
38           delete (TokenCT*) thisTokenP;
39           thisTokenP = OutputP;
40         } // end pipeline construct
41                                                                   subconstruct
42         Merge(OutputP, (TokenDT*) thisTokenP, Index);
43         delete (TokenDT*) thisTokenP;
44       } // end if
45     } // end for
46
47     delete InputP;
48     thisTokenP = OutputP;
49   } // end indexed parallel construct
50
51   OutputP = (TokenDT*) thisTokenP;
52 } // end ProcessAT::Operation1
```

Program 3-30. Equivalent C/C++ specification of the *indexed parallel* construct

3.9 The first CAP program: The Sieve of Eratosthenes

Invented by the greek mathematician Eratosthenes in about 200 BC, The Sieve of Eratosthenes is a very simple algorithm to compute prime numbers, i.e. numbers that are only divisible by 1 and by itself. In order to check whether a number is prime or not, this algorithm consists of iteratively dividing this number by the previously computed prime numbers in the growing order, i.e. 2, 3, 5, etc. As soon as one of the prime numbers divides the checked number, it is rejected. If the number is not divisible by all the prime numbers, then it is a new prime number to insert in the list. The algorithm starts by checking the number 2 which is a prime number since the list of previously computed prime numbers is empty, and continues by checking all the numbers in the growing order, i.e. 3, 4, 5, 6, etc.

The parallel algorithm consists of assigning a different thread of execution to each of the previously computed prime numbers so as to form a pipeline of computation. Each threads iteratively (1) waits for a number; (2) divides it with its locally stored prime number; (3) if it divides then rejects it otherwise transfers it to the adjacent thread if it exists. If the thread is the last one in the pipeline, then the number is a new prime number and a new thread is inserted in the pipeline.

In order to avoid dynamically creating many threads, a pipeline is made of a predefined number of threads connected so as the last thread is connected to the first thread. Each thread possesses several computed prime numbers corresponding to different stages in the computation pipeline. Figure 3-19 shows "The Sieve of Eratosthenes" distributed on 3 different threads and the computation pipeline.

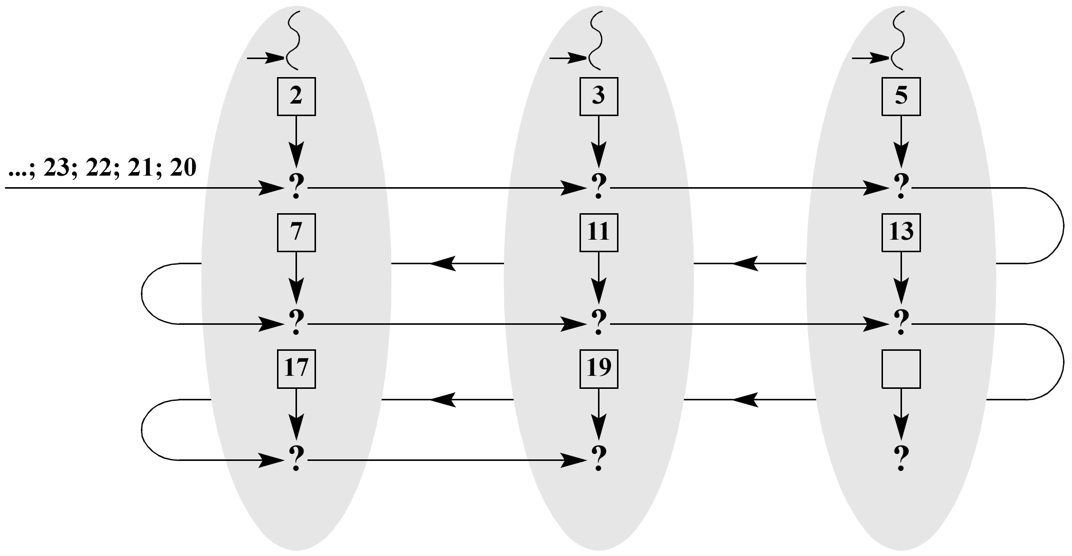


Figure 3-19. The Sieve of Eratosthenes distributed on 3 different threads

This algorithm is not of practical interest for parallel computation since its grain of parallelism is too small, i.e. the ratio between computation and communication is the ratio of division time and the time to send a number across the network. Nevertheless the Sieve of Eratosthenes is a simple program that demonstrates how to use parallel CAP constructs (Sections 3.8) to program the computation pipeline of Figure 3-19.

The macro dataflow of the algorithm (Figure 3-19) is depicted in Figure 3-20. The input of the parallel *ParallelProcessingServerT::ComputePrimeNumbers* operation is a *ComputationRequestT* token comprising the number up to which the algorithm must find the prime numbers. This request is divided by the split *GenerateNumber* function which generates successive numbers from 2 to the specified maximum number. Iteratively a *NumberT* token is redirected through the pipeline formed by the *Slave[]* threads to be filtered by the sequential *ComputeServerT::FilterNumber* operations until the number is either rejected or is a prime number. The *Master* thread gathers the results into the output *PrimeNumbersT* token using the merge *MergeNumber* function. Once all the generated *NumberT* tokens are merged, the output *PrimeNumbersT* token comprising the array of found prime numbers is redirected to its successor.

Figure 3-21 shows the timing diagram of the execution of the DAG in Figure 3-20. The master thread both sends the numbers and collects the results of the pipeline execution on 3 slave threads.

Program 3-31 shows the declaration of the 3 tokens involved in the parallel computation, i.e. the *ComputationRequestT* token (lines 1-4), the *PrimeNumbersT* token (lines 34-40), and the *NumberT* token (lines 47-53). A *ComputationRequestT* token contains the number up to all the prime numbers must be found (line 3). The result of the parallel computation is a *PrimeNumbersT* token comprising an array of prime numbers (line 36) and its size (line 37). In order to be able to serialize the token, the *ArrayOfIntsT* class (lines 6-31) has been defined. Its serialization routines are not shown in Program 3-31 since Section 4.4.1 is entirely devoted to this issue. *NumberT* tokens that flow through the computation pipeline contain 4 fields, namely the number itself (line 49), a first boolean variable specifying if the token has to be redirected to the next thread in the pipeline (line 50), a second boolean variable indicating if this number is a prime number (line 51), and the filter index or stage index in the pipeline (line 52).

Program 3-32 shows the *ParallelProcessingServer* CAP process hierarchy (line 28) where two processes are declared*:* A *ComputeServerT* leaf process (lines 3-12) and a *ParallelProcessingServerT* hierarchical process (lines 15-25). The *ComputeServerT* leaf process offers the *FilterNumber* sequential operation (lines 9-11) that divides the input *NumberT* number with its locally stored prime number and creates the output *NumberT* number according to whether the number is divisible or not. The *PrimeNumbers* thread local storage (line 6) is an array of

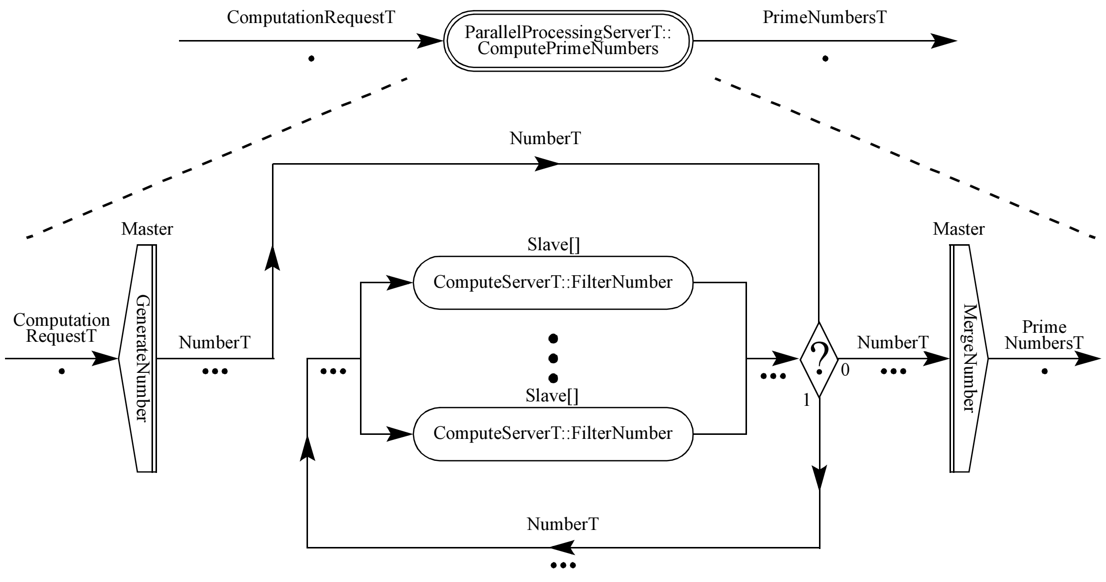


Figure 3-20. Graphical CAP specification of “The Sieve of Eratosthenes”

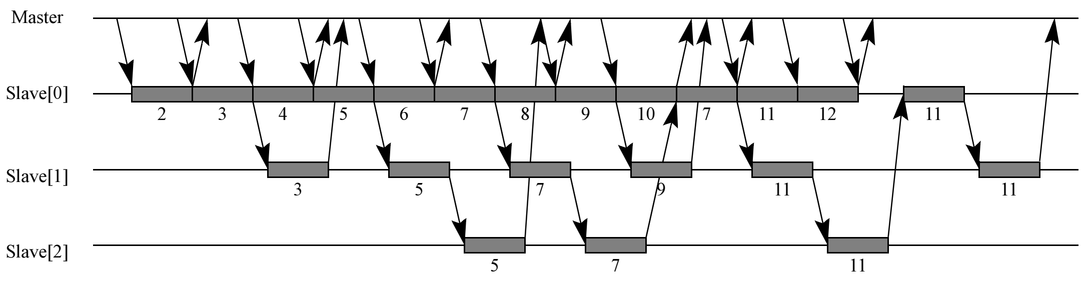


Figure 3-21. Timing diagram of the execution of “The Sieve of Eratosthenes” with a master thread and 3 slave threads

```
token ComputationRequestT
{
  int MaximumNumber;
}; // end token ComputationRequestT

class ArrayOfIntsT
{
public:
  ArrayOfIntsT();
  ArrayOfIntsT(int size);
  ~ArrayOfIntsT();

public:
  int Size;
  int* ArrayP;
}; // end class ArrayOfIntsT

ArrayOfIntsT::ArrayOfIntsT()
  : Size(0), ArrayP(0)
{} // end ArrayOfIntsT::ArrayOfIntsT

ArrayOfIntsT::ArrayOfIntsT(int size)
  : Size(size)
{
  ArrayP = new int[size];
} // end ArrayOfIntsT::ArrayOfIntsT

ArrayOfIntsT::~ArrayOfIntsT()
{
 delete ArrayP;
} // end ArrayOfIntsT::~ArrayOfIntsT()


token PrimeNumbersT
{
  ArrayOfIntsT PrimeNumbers;
  int NumberOfPrimeNumbers;

  PrimeNumbersT(ComputationRequestT* fromP);
}; // end token PrimeNumbers

PrimeNumbersT::PrimeNumbersT(ComputationRequestT* fromP)
  : PrimeNumbers(fromP->MaximumNumber),
    NumberOfPrimeNumbers(0)
{} // end PrimeNumbersT::PrimeNumbersT

token NumberT
{
  int Number;
  bool ContinueToFilterNumber;
  bool IsPrimeNumber;
  int FilterIndex;
}; // end token NumberT
```

Program 3-31. The Sieve of Eratosthenes tokens

integers that can dynamically shrink and grow as necessary (the *CArray* class is part of the Microsoft Foundation Class MFC library). This feature is necessary, since we do not know in advance how many prime numbers each slave must store.

```
 1 const int NUMBER_OF_SLAVES = 5;
 2
 3 process ComputeServerT
 4 {
 5 variables:
 6   CArray<int,int> PrimeNumbers; // 2D array of ints
 7
 8 operations:
 9   FilterNumber
10     in NumberT* InputP
11     out NumberT* OutputP;
12 }; // end process ComputeServerT
13
14
15 process ParallelProcessingServerT
16 {
17 subprocesses:
18   ComputeServerT Master;
19   ComputeServerT Slave[NUMBER_OF_SLAVES];
20
21 operations:
22   ComputePrimeNumbers
23     in ComputationRequestT* InputP
24     out PrimeNumbersT* OutputP;
25 }; // end ParallelProcessingServerT
26
27
28 ParallelProcessingServerT ParallelProcessingServer;
```

Program 3-32. The “Sieve of Eratosthenes” CAP process hierarchy

The hierarchical *ParallelProcessingServerT* process has a *Master* leaf subprocess (line 18) and 5 *Slave[]* leaf subprocesses (line 19) called *Slave[0]* to *Slave[4]* (user-defined). The *NUMBER_OF_SLAVES* constant is defined at line 1. The *Master* thread is responsible for generating the (*ComputationRequestT::MaximumNumber*-1) successive numbers and merging the parallel computation results into the *PrimeNumbersT* token. *Slave[0]* to *Slave[4]* threads form the computation pipeline. The hierarchical *ParallelProcessingServerT* process can perform a parallel *ComputePrimeNumber* operation on a *ComputationRequestT* input token, and produce a *PrimeNumbersT* output token (lines 22-24).

The process hierarchy is instantiated at line 28. At running time and with a configuration file (Program 3-36), the CAP runtime system will automatically spawn the 6 threads, namely *Master* and *Slave[0]* to *Slave[4]*, in order to execute the parallel *ComputePrimeNumbers* operation on these 5 threads.

The implementation of the sequential *ComputeServerT::FilterNumber* operation is shown in Program 3-33.

```
 1 leaf operation ComputeServerT::FilterNumber
 2   in NumberT* InputP
 3   out NumberT* OutputP
 4 {
 5   const int FilterIndex = InputP->FilterIndex/NUMBER_OF_SLAVES; // local index of filter (integer division)
 6
 7   OutputP = new NumberT;
 8   OutputP->Number = InputP->Number;
 9   OutputP->FilterIndex = InputP->FilterIndex + 1; // global index of filter
10
11   if ( PrimeNumbers.GetSize() <= FilterIndex )
12   {
13     // It is a prime number
14     PrimeNumbers.SetAtGrow(FilterIndex, Input->Number);
15     OutputP->ContinueToFilterNumber = false;
16     OutputP->IsPrimeNumber = true;
17   }
18   else
19   {
20     if ( InputP->Number % PrimeNumbers[FilterIndex] == 0 )
21     {
22       // It is not a prime number
23       OutputP->ContinueToFilterNumber = false;
24       OutputP->IsPrimeNumber = false;
25     }
26     else
27     {
28       // Continue to filter the number
29       OutputP->ContinueToFilterNumber = true;
30       OutputP->IsPrimeNumber = false;
31     } // end if
32   } // end if
33 } // end ComputeServerT::FilterNumber
```

Program 3-33. Sequential *ComputeServerT::FilterNumber* operation

Three different cases may arise:

1. There is no locally stored prime number at this stage (line 11). Therefore the input number is a prime number (line 16). It must be stored in the dynamically growing array (line 14). No further filtering is needed for that number (line 15).
2. The input number is divisible by the locally stored prime number (line 20). In that case the number is not a prime number (line 24) and filtering must end (line 23).

3. The input number is not divisible by the locally stored prime number (line 26). In that case the number has to be filtered (line 29) by the next slave in the pipeline (line 9).

Program 3-34 shows the CAP specification of the parallel *ParallelProcessingServerT::ComputePrimeNumbers* operation. As depicted in its directed acyclic graph in Figure 3-20, the parallel *ParallelProcessingServerT::ComputePrimeNumbers* operation consists of an *indexed parallel* CAP construct (Section 3.8.7) that repeatedly calls the *GenerateNumber* split function for *Number* ranges from 2 to *ComputationRequestT::MaximumNumber* (line 26). The *GenerateNumber* routine simply creates a *NumberT* token (line 3) and initializes its 4 fields (lines 4-7).

```
 1 void GenerateNumber(ComputationRequestT* inputP, NumberT* &subtokenP, int number)
 2 {
 3   subtokenP = new NumberT;
 4   subtokenP->Number = number;
 5   subtokenP->ContinueToFilterNumber = true;
 6   subtokenP->IsPrimeNumber = false;
 7   subtokenP->FilterIndex = 0;
 8 } // end GenerateNumber
 9
10 void MergeNumber(PrimeNumbersT* outputP, NumberT* subtokenP, int number)
11 {
12   if ( subtokenP->IsPrimeNumber )
13   {
14     outputP->PrimeNumbers.ArrayP[outputP->NumberOfPrimeNumbers] = subtokenP->Number;
15     outputP->NumberOfPrimeNumbers++;
16   } // end if
17 } // end MergeNumber
18
19 operation ParallelProcessingServerT::ComputePrimeNumbers
20   in ComputationRequestT* InputP
21   out PrimeNumbersT* OutputP
22 {
23   Master.{ } // forces the InputP token to be redirected to the Master thread
24   >->
25   indexed
26     (int Number = 2; Number <= thisTokenP->MaximumNumber; Number++)
27   parallel
28     (GenerateNumber, MergeNumber, Master, local PrimeNumbersT Result(thisTokenP))
29   (
30     while ( thisTokenP->ContinueToFilterNumber )
31     (
32       Slave[thisTokenP->FilterIndex % NUMBER_OF_SLAVES].FilterNumber
33     ) // end while
34   ); // end indexed parallel
35 } // end operation ParallelProcessingServerT::ComputePrimeNumbers
```

Program 3-34. Parallel *ParallelProcessingServerT::ComputePrimeNumbers* operation

The *while* CAP construct (Section 3.8.3) redirects each of the generated *NumberT* tokens to one of the *Slave* threads as long as the number needs to be filtered (lines 30-33). The selection of the *Slave* thread (line 32) is made according to the *NumberT::FilterIndex* value of the token about to enter in the sequential *ComputeServerT::FilterNumber* operation. Since consecutive *FilterIndex* (Program 3-33 line 9) filter indexes are produced, *NumberT* tokens flow through the *Slave* threads in a round-robin fashion as shown in Figure 3-19. Once a *NumberT* token quits the while loop, filtering ends and the token is merged into the *Result* token by the *Master* thread using the *MergeNumber* routine (line 28).

The inline operation at line 23 forces the C/C++ *indexed parallel* indexing expression (line 26) and the *GenerateNumber* split function (lines 1-8) to be executed by the *Master* thread thus ensuring that the split and the merge functions are executed by the same thread independently of who is calling the parallel *ParallelProcessingServerT::ComputePrimeNumbers* operation. Remember from Section 3.8 the discussion about who executes parallel operations: "The producing thread performs the parallel operation, and decides himself where to redirect the token it produced". At line 23 the *Master* thread forwards the *ComputationRequestT* input token, initializes the *PrimeNumbersT* output token (line 28), and executes the *indexed parallel* loop (line 26) and the *GenerateNumber* split function (line 28).

Since the thread who executes the split function is similar to the thread who executes the merge function, i.e. the *Master* thread, one can initialize the *PrimeNumbersT* output token (line 28) either locally with the '*local*' keyword or remotely with the '*remote*' keyword (Section 3.8.5).

Program 3-35 shows the *main* function where the parallel *ParallelProcessingServerT::ComputePrimeNumbers* operation is called (line 15) and the prime numbers displayed (lines 17-21).

```
int main(int argc, char* argv[])
{
  int MaximumNumber, Index;
  ComputationRequestT* InputP;
  PrimeNumbersT* OutputP;

  printf("The Sieve of Eratosthenes\n");
  printf("Compute prime numbers between [2..MaximumNumber]\n");
  printf("MaximumNumber = ");
  fscanf(stdin, "%d", &MaximumNumber);

  InputP = new ComputationRequestT;
  InputP->MaximumNumber = MaximumNumber;

  call ParallelProcessingServer.ComputePrimeNumbers in InputP out OutputP;

  printf("Prime numbers between [2..%d]:\n", MaximumNumber);
  for (Index = 0; Index < OutputP->NumberOfPrimeNumbers; Index++)
  {
    printf("%d\n", OutputP->PrimeNumbers.ArrayP[Index]);
  } // end for
  delete OutputP;

  return 0;
} // end main
```

Program 3-35. "The Sieve of Eratosthenes" C/C++ main program

In order to run the Sieve of Eratosthenes CAP program (Programs 3-31, 3-32, 3-33, 3-34 and 3-35) on a multi-PC environment, the CAP runtime system needs a configuration file (Program 3-36) indicating how the 6 threads are allocated on the various PC's.

```
configuration {
processes:
  A ( "user" ) ;
  B ( "128.178.75.65", "\\FileServer\SharedFiles\Eratosthenes.exe" ) ;
  C ( "128.178.75.66", "\\FileServer\SharedFiles\Eratosthenes.exe" ) ;
  D ( "128.178.75.67", "\\FileServer\SharedFiles\Eratosthenes.exe" ) ;
  E ( "128.178.75.68", "\\FileServer\SharedFiles\Eratosthenes.exe" ) ;
  F ( "128.178.75.69", "\\FileServer\SharedFiles\Eratosthenes.exe" ) ;

threads:
  "Master" (A) ;
  "Slave[0]" (B) ;
  "Slave[1]" (C) ;
  "Slave[2]" (D) ;
  "Slave[3]" (E) ;
  "Slave[4]" (F) ;
}; // end of configuration file
```

Program 3-36. The Sieve of Eratosthenes configuration file

3.10 Issue of flow-control in a pipeline within a split-merge *parallel while* or *indexed parallel* CAP construct

When examining carefully Programs 3-27, 3-29 and 3-34, a common question arises: Who regulates the number of tokens generated by the split routine? Who prevents the split routine to generate all its subtokens without knowing whether the *parallel while* or the *indexed parallel* body subconstruct is consuming them at the same rate they are produced? The answer is nobody. The split routine, in the three programs, actually generates all the subtokens without stopping. If the split routine creates few subtokens this is probably acceptable, but if the split routine generates thousands of large subtokens then the problem is different. The processor will be overloaded executing continuously the split routine, memory will overflow, and the network interface will saturate sending all these subtokens. This will result in a noticeable performance degradation and later in a program crash with a "no more memory left" error message.

The problem with the *parallel while* (Section 3.8.6) and the *indexed parallel* (Section 3.8.7) CAP constructs is that the split routine generates input tokens faster than the merge routine consumes *parallel while* or *indexed parallel* body subconstruct output tokens. Either because the *parallel while* or *indexed parallel* body subconstruct output token rate is too slow or the merging time of a token is too long. Therefore tokens accumulate somewhere in the pipeline between the split and the merge routine in the token queue located in front of the most loaded PC's, thus becoming the bottleneck of the application (Figure 3-22).

In a multi-PC environment potential bottlenecks are: processors, memory interfaces, disks, network interfaces comprising the message passing system interface, the PCI network card adapter and the 100 Mbits/s Fast Ethernet network. If the processor, the memory interface or the disks are the bottleneck, then tokens are accumulated in the

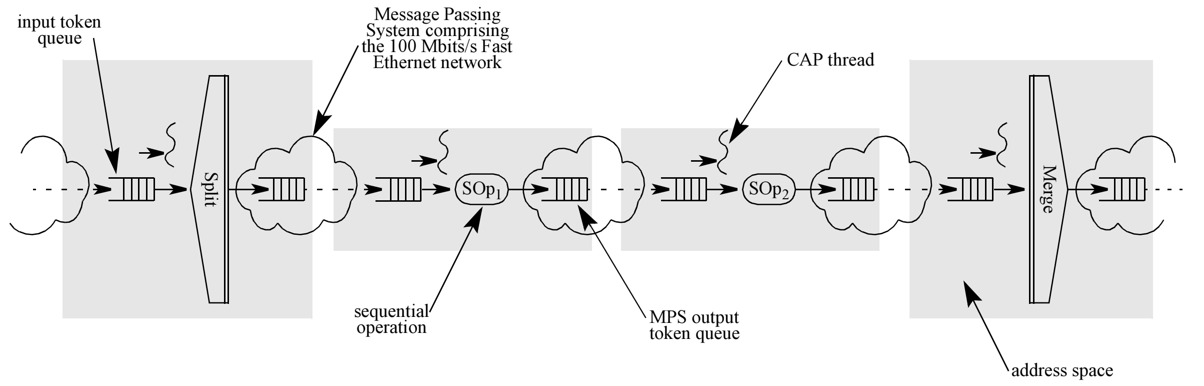


Figure 3-22. Example of a 4-stage pipeline composed of a split routine, two intermediate sequential operations and a merge routine. Note the input token queues in front of each CAP threads and the MPS output token queues at the border of each address spaces.

input token queue (Figure 3-22) of the thread who executes the sequential operation that uses this offending resource, i.e. the resource that forms the bottleneck. Alternately, if the message passing system interface, the PCI network card adapter or the Fast Ethernet network is the bottleneck, then tokens are accumulated in the message passing system output queue (Figure 3-22) of the offending Windows NT process, i.e. the Windows NT process that sends too many tokens. The consequence of such a congestion point is that computing resources are monopolized for handling such a peak in the flow of tokens, e.g. memory space for storing tokens and computing power for sending/receiving tokens, leading to a rapid degradation of performances (virtual memory thrashing).

Best performances are achieved when the split routine generates tokens at the same rate as the merge routine consumes the *parallel while* or *indexed parallel* body subconstruct output tokens. In that situation the most loaded components in the pipeline becomes the bottleneck, i.e. is 100% of the time active and no further improvement is possible.

For flow-control, two approaches are possible. The first approach consists of a low-level flow-control mechanism where CAP's runtime system (message-passing system) itself incorporates at each token queue, i.e. input token queues and MPS output token queues, a local flow-control mechanism (Figure 3-23) similar to Xon-Xoff in RS-232 communications.

1. This mechanism detects when the size of the token queue becomes too important, i.e. either consumes too much memory or contains too many tokens.
2. When the size of the token queue monitored in step 1 is above a certain threshold$_{max}$, the mechanism is able to stop the producer from generating tokens.
3. It detects when the size of the token queue becomes too small, i.e. either consumes not enough memory or does not contain enough tokens.
4. When the size of the token queue monitored in step 3 is below a certain threshold$_{min}$, the mechanism is able to resume the producer to generate tokens.

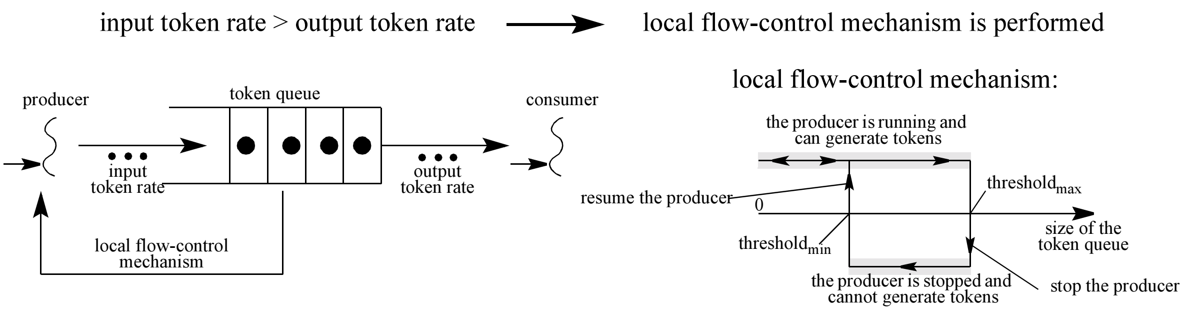

input token rate > output token rate
local flow-control mechanism is performed
producer
token queue
consumer
input token rate
output token rate
local flow-control mechanism
local flow-control mechanism:
the producer is running and can generate tokens
resume the producer
0
threshold$_{max}$
threshold$_{min}$
size of the token queue
the producer is stopped and cannot generate tokens
stop the producer

Figure 3-23. Local flow-control mechanism regulating the input token rate according to the output token rate

The local flow-control mechanism (Figure 3-23) is performed on a token queue basis. But since the producer of one token queue is the consumer of the previous token queue, stopping a producer will have the effect of making the previous token queue grow. After a certain amount of time, the previous local flow-control mechanism will stop its producer. And so on and so forth until the producer is the thread who runs the split routine which will stop generating output tokens. When the original token queue becomes almost empty, the local flow-control mechanism resumes the execution of the producer which will make the previous token queue shrink. After a certain amount of time, the previous local flow-control mechanism will resume the execution of its producer. Finally, the producer who runs the split routine will resume generating output tokens.

The low-level flow-control mechanism adapts easily to variable running conditions, e.g. changes in processor, memory or network utilization, and does not require additional communications. Although with this low-level mechanism we do not need to modify user's CAP programs (in particular the *parallel while* and *indexed parallel* CAP constructs), this solution has not been considered in current releases of CAP. Preventing a thread from sending tokens introduces new issues not investigated yet, e.g. problems of deadlock and circulation of high-priority data (supervision commands, monitoring commands).

The second approach consists in applying a high-level flow-control mechanism. The offending *parallel while* and *indexed parallel* CAP constructs are replaced by a combination of *parallel while*, *indexed parallel* and *for* CAP constructs in order to maintain the difference between the number of split tokens and the number of merged tokens constant by having additional communication between the thread who runs the merge routine and the thread who runs the split routine (Figure 3-24).

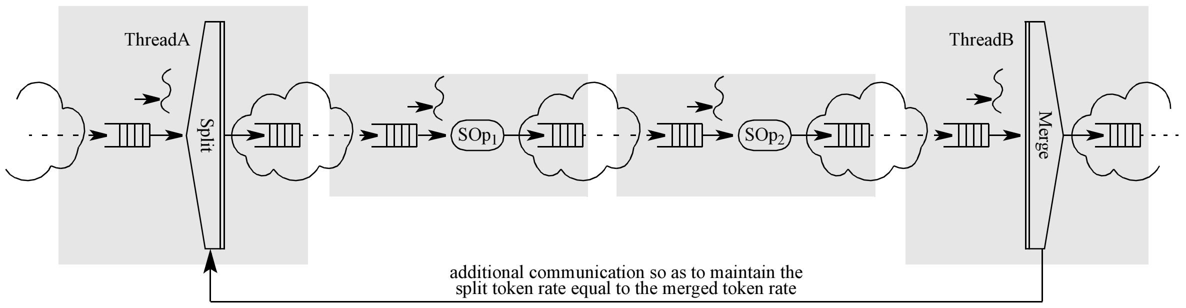

ThreadA
Split
SOp1
SOp2
ThreadB
Merge
additional communication so as to maintain the split token rate equal to the merged token rate

Figure 3-24. High-level flow-control mechanism requiring additional communication between the thread who runs the merge routine (the ThreadB thread) and the thread who runs the split routine (the ThreadA thread) so as to maintain the split token rate equal to the merged token rate

By doing this, the program regulates by itself the split token rate according to the merged token rate, thus preventing tokens to be accumulated somewhere in the pipeline causing a possible degradation of performance.

Program 3-37 shows a CAP program with an *indexed parallel* CAP construct at lines 7-13. The inline operation at line 5 constraints the C/C++ *indexed parallel* indexing expression (line 8) and the *Split* routine (line 10) to be executed by the *ThreadA* thread. The *Merge* routine is executed by the *ThreadB* thread (line 10). Without an adequate flow-control mechanism this parallel CAP construct generates 10'000 *TokenCT* tokens that will certainly fill up the memory and cause a memory overflow condition.

```
 1 operation ProcessAT::Operation1
 2   in TokenAT* InputP
 3   out TokenDT* OutputP
 4 {
 5   ThreadA.{ }
 6   >->
 7   indexed
 8     (int Index = 0; Index < 10000; Index++)
 9   parallel
10     (Split, Merge, ThreadB, remote TokenDT Out(thisTokenP))
11   (
12     ProcessB.Operation1
13   ); // end indexed parallel
14 } // end operation ProcessAT::Operation1
```

Program 3-37. Example of a CAP program requiring a flow-control preventing the split routine from generating 10'000 tokens

In order to maintain the difference between the number of split *TokenCT* tokens and the number of merged *TokenDT* tokens constant, e.g. 20 tokens, the *indexed parallel* CAP construct is rewritten. Program 3-38 shows the first high-level flow-controlled split-merge construct replacing the original *indexed parallel* CAP construct in Program 3-37. The idea consist of generating a small amount of tokens, e.g. 20, with a first *indexed parallel* CAP construct (lines 7-10) and to redirect 500 times, with a second *for* CAP construct (line 12), each of these 20 tokens, through the pipeline formed by the split routine (line 14), the parallel *ProcessBT::Operation1* operation (line 16) and the merge routine (line 18). In other words, the *indexed parallel* CAP construct (lines 7-11) generates 20 tokens (or 20 parallel flows) and each of these 20 tokens circulates, in parallel, 500 times through the pipeline (lines 14-18) thanks to the *for* CAP construct (lines 12-13). The combination of the *indexed parallel* and the *for* CAP constructs ensures that exactly 20 tokens are always flowing in the pipeline, thus preventing a large memory consumption and a possible memory overflow.

```
 1 operation ProcessAT::Operation1
 2   in TokenAT* InputP
 3   out TokenDT* OutputP
 4 {                                                                    filling factor
 5   ThreadA.{ }
 6   >->
 7   indexed
 8     (int IndexFlowControl1 = 0; IndexFlowControl1 < 20; IndexFlowControl1++)
 9   parallel
10     (SplitFlowControl1, MergeFlowControl1, ThreadB, remote FlowControlT Out1(thisTokenP))
11   (
12     for (int IndexFlowControl2 = 0; IndexFlowControl2 < 10000/20; IndexFlowControl2++)
13     (
14       ThreadA.SplitFlowControl2
15       >->
16       ProcessB.Operation1
17       >->
18       ThreadB.MergeFlowControl2
19     ) // end for                                                               pipeline
20   ) // end indexed parallel
21   >->
22   ThreadB.ReturnOutputToken;
23 } // end operation ProcessAT::Operation1
```

Program 3-38. First high-level flow-controlled split-merge construct that requires one communication between the thread who runs the merge routine, i.e. the *ThreadB* thread, and the thread who runs the split routine, i.e. the *ThreadA* thread, for each merged tokens

This first high-level flow-controlled split-merge construct (Program 3-38) requires one additional communication between the thread who runs the merge routine and the thread who runs the split routine. For example, on a 6-stage pipeline (split routine, 4 intermediate pipe stages, and merge routine) this correspond to a communication increase of 20%, i.e. 6 token transfers instead of 5 token transfers.

Henceforth the number of tokens that are simultaneously flowing in the pipeline, i.e. within the *for* CAP construct (lines 14-18) is named the *filling factor*. Performance of a flow-controlled split-merge construct (Program 3-38) highly depends on its filling factor. A too large filling factor creates a congestion point that consumes lots of memory degrading performance and possibly crashing the program. A too small filling factor does not provide enough tokens to saturate one of the PC's components, i.e. the most loaded, on which the pipeline executes. This decreases the parallelism between the different pipeline stages and performance suffers. At the extreme, a filling factor of 1 corresponds to a serial execution of the split-merge construct (Programs 3-28 and 3-30). Best performances are obtained when the filling factor is equal to the minimum factor that saturates the most loaded components on which the pipeline executes.

In order to illustrate how we calculate the optimal value of the filling factor giving the best performance, let us take a simple example where the pipeline is composed of 5 compute-bound stages, each distributed on a different PC. Stages 1 to 5 occupy the processor for 50 ms, 160 ms, 200 ms, 100 ms and 150 ms respectively. The tokens flowing through these 5 stages are made of a single 32 bit number and the time for sending such a 32-bit token from one PC to another is 530 μs (calculated using Equation 3-3 and the measured serial execution time). The experiment consists of flowing through the 5-stage pipeline 1000 times using the high-level flow-controlled split-merge construct shown in Program 3-38 and completing the computation on the same PC it starts, i.e. the 50ms-stage PC. Since for the experiment we use Bi-Pentium Pro PC's, we can assume that sending and receiving tokens happen asynchronously without interfering with the computation, i.e. one processor is dedicated to computation and the second is dedicated to communication. Communication overlaps computation on a given PC[1].

The third processor runs the thread who executes the slowest 200ms stage. It represents the application's bottleneck when the following equation is verified:

1. As mentioned in Section 4.5, on a mono-processor PC, communications are only partially hidden, since the TCP/IP protocol stack requires considerable processing power.

$$(FillingFactor - 1) \cdot 200ms \geq 0.53ms + 100ms + 0.53ms + 150ms + 0.53ms + 50ms + 0.53ms + 160ms + 0.53ms \quad \textbf{(3-1)}$$

Equation 3-1 reflects the case when there are enough tokens in the full pipeline for keeping the third thread active while the first token leaves the 200ms-stage, flows through the next 4 stages and gets back to the 200ms-stage input token queue. Solving equation 3-1 gives a filling factor of at least 3.32 tokens. With an optimal value of 4 tokens, the present high-level flow-controlled split-merge construct (Program 3-38) gives the best performance. Increasing the filling factor beyond 4 tokens will not improve performance since the third processor is already 100% active executing repeatedly the slowest 200ms pipe stage.

More generally, Equation 3-2 gives the formula for calculating the filling factor of any K-stage pipelines running on K different PC's, where the pipeline round trip time is the time for a token to flow through the whole pipeline and to arrive where it starts.

$$(FillingFactor - 1) \cdot LengthOfSlowestPipeStage \geq PipelineRoundTripTime - LengthOfSlowestPipeStage \quad \textbf{(3-2)}$$

Figures 3-25 to 3-28 show how the pipeline execution time decreases while the filling factor increases (PC's are bi-processor PC's).

Figure 3-25 shows the timing diagram of the execution of the 5-stage pipeline with a filling factor of 1.

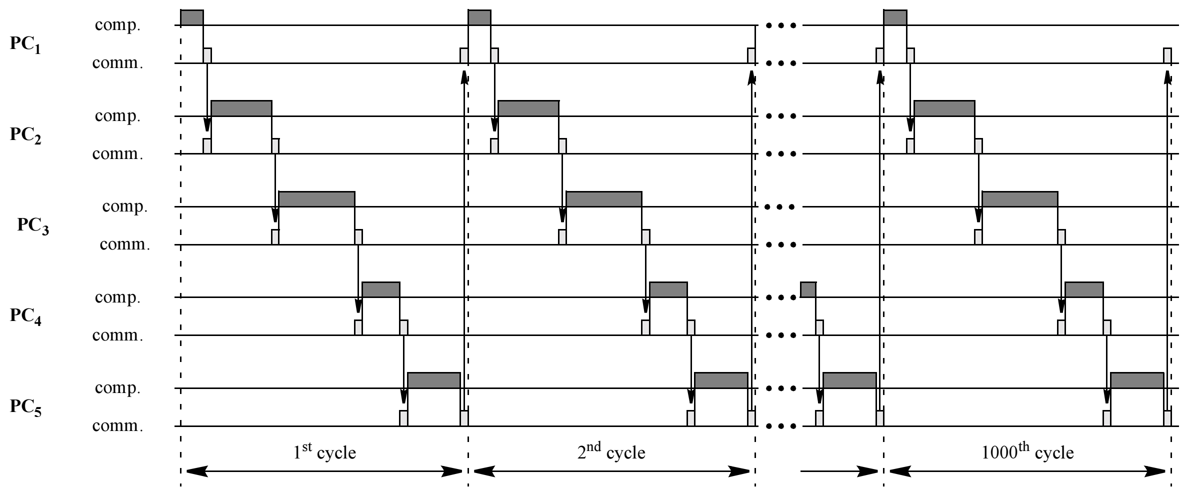


Figure 3-25. Timing diagram of the execution of the 5-stage pipeline with a filling factor of 1

This corresponds to a serial execution where each pipe stages is executed one after the other. In that case the execution time is:

$$1000 \cdot (50ms + 0.53ms + 160ms + 0.53ms + 200ms + 0.53ms + 100ms + 0.53ms + 150ms + 0.53ms) = 662.651s \quad \textbf{(3-3)}$$

Figure 3-26 shows the timing diagram with a filling factor of 2.

In that case the execution time is reduced by a factor of 2:

$$\begin{array}{l} 1 \cdot 200ms + \\ 500 \cdot (50ms + 0.53ms + 160ms + 0.53ms + 200ms + 0.53ms + 100ms + 0.53ms + 150ms + 0.53ms) = 331.525s \end{array} \quad \textbf{(3-4)}$$

With a filling factor of 3, the processor on PC_3 is still not the bottleneck (Figure 3-27).

The execution time is reduced by a factor of 3 compared with the serial execution in Figure 3-25:

$$\begin{array}{l} 2 \cdot 200ms + \\ 334 \cdot (50ms + 0.53ms + 160ms + 0.53ms + 200ms + 0.53ms + 100ms + 0.53ms + 150ms + 0.53ms) = 221.725s \end{array} \quad \textbf{(3-5)}$$

As demonstrated by Equation 3-1, the third processor becomes the application's bottleneck from a filling factor of 4 tokens (Figure 3-28).

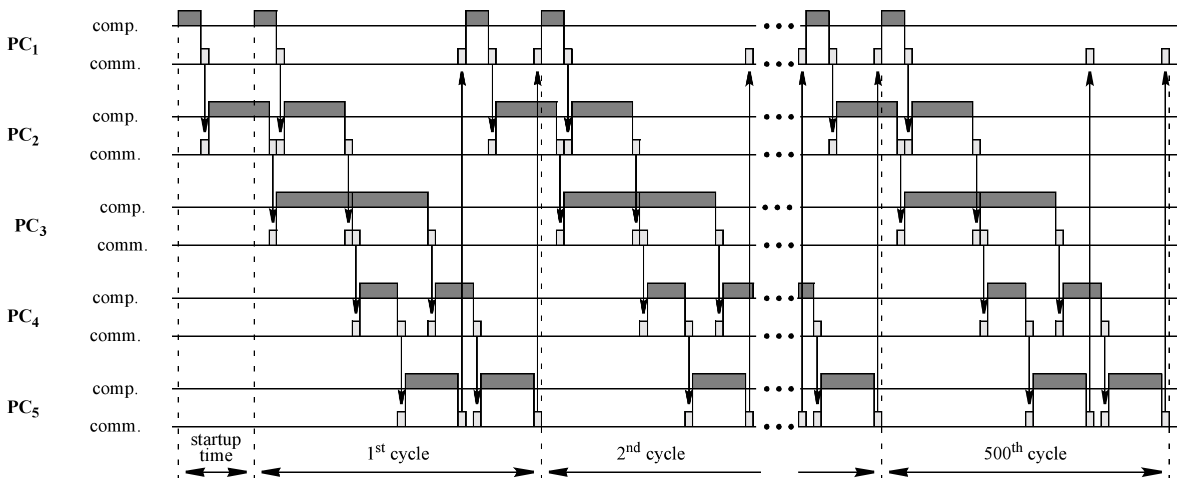


Figure 3-26. Timing diagram of the execution of the 5-stage pipeline with a filling factor of 2

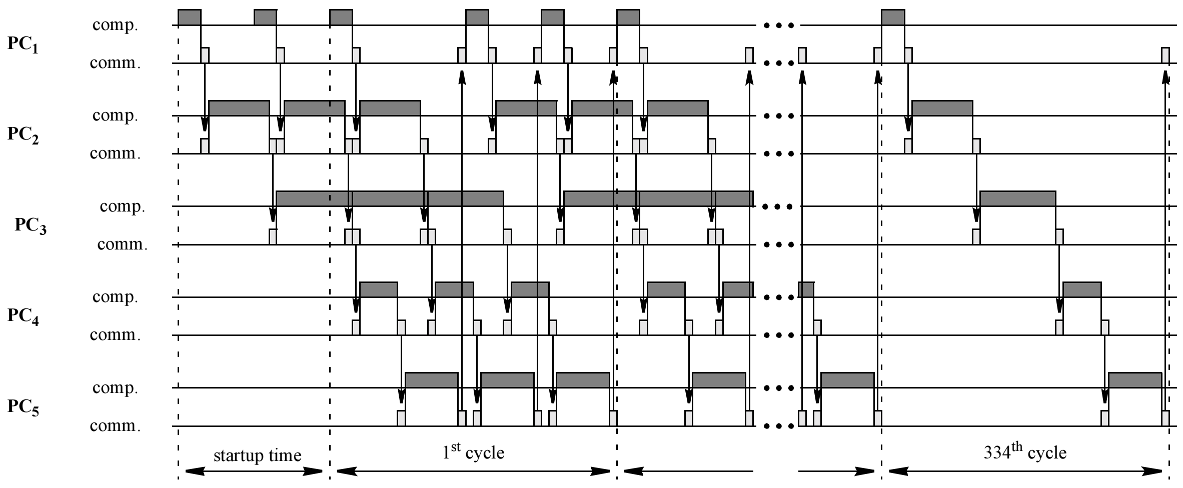


Figure 3-27. Timing diagram of the execution of the 5-stage pipeline with a filling factor of 3

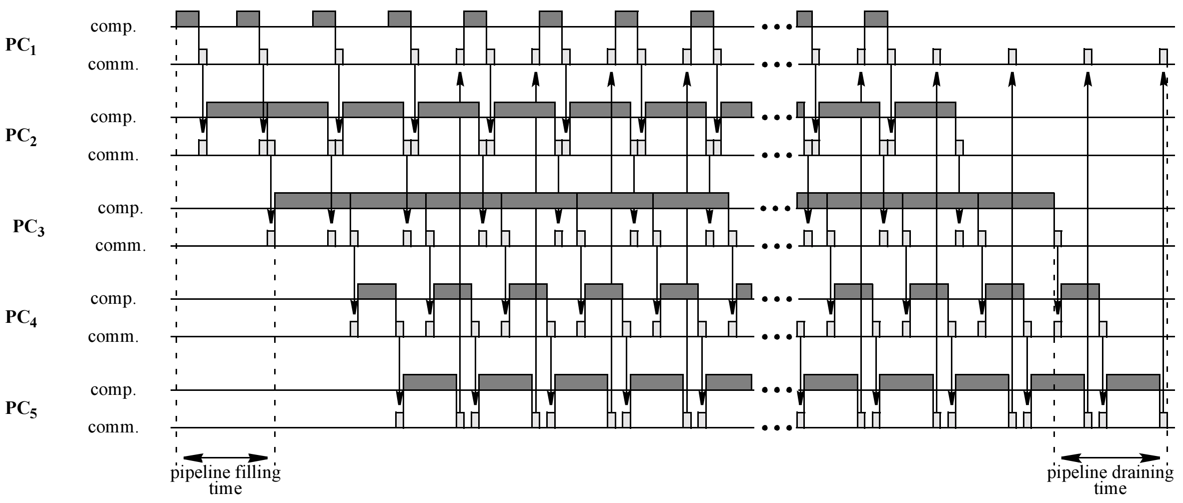


Figure 3-28. Timing diagram of the execution of the 5-stage pipeline with a filling factor of 4

The *pipeline filling time* is the time for a token to reach the slowest pipe stage from the beginning of the pipeline and the *pipeline draining time* is the time for a token to reach the beginning of the pipeline from the end of the slowest pipe stage. The minimum 5-stage pipeline execution time on a 5 PC environment is:

$$\begin{aligned}&(50ms + 0.53ms + 160ms + 0.53ms) + \\ &(1000 \cdot 200ms) + (0.53ms + 100ms + 0.53ms + 150ms + 0.53ms) = 200.463s\end{aligned} \quad \textbf{(3-6)}$$

More generally and only if the filling factor is large enough to make the processor executing the slowest stage the bottleneck (Equation 3-2), the minimum execution time of a K-stage pipeline running on K different PC's (Program 3-38) is given by Equation 3-7 where N is the number of tokens which need to travel through the whole pipeline.

$$PipelineExecutionTime = PipelineFillingTime + N \cdot LengthOfSlowestPipeStage + PipelineDrainingTime \quad \textbf{(3-7)}$$

Or:

$$PipelineExecutionTime = PipelineRoundTripTime + (N - 1) \cdot LengthOfSlowestPipeStage \quad \textbf{(3-8)}$$

A remarkable effect in Equations 3-6, 3-7 and 3-8 is that communication overhead is present only in the pipeline filling and draining time factors. During most of the execution time, i.e. 'N x LengthOfSlowestPipeStage' or in our example 99.77% of the time, communications are overlapped by computations. This is achieved in CAP thanks to the *pipeline* CAP construct (Section 3.8.1) and to the message passing system which asynchronously sends and receives tokens (Section 4.3.3).

Table 3-1 compares the 3 predicted 5-stage pipeline execution times with those measured on 5 Bi-Pentium Pro 200 MHz PC's interconnected by a 100 Mbits/s Fast Ethernet network. As correctly predicted, from a filling factor of 4 tokens the slowest 200ms pipe stage becomes the bottleneck and communications are completely overlapped by computations.

Filling factor	Predicted execution time [secs]	Measured execution time [secs]
1	not predicted; measured	662.651
2	331.525	331.725
3	221.725	221.869
4	200.463	200.711
5	200.463	200.711
10	200.463	200.686
20	200.463	200.688
50	200.463	200.672
100	200.463	200.666
200	200.463	200.705

Table 3-1. Comparison of the predicted and measured execution times of the 5-stage pipeline for various filling factors

Figure 3-29 depicts the evolution of the measured 5-stage pipeline execution times for 10 different filling factors.

From Equation 3-6, we can calculate the maximum 5-stage pipeline application's speed-up:

$$\frac{1000 \cdot (50ms + 160ms + 200ms + 100ms + 150ms)}{200.463s} = 3.29 \quad \textbf{(3-9)}$$

This speedup gives an efficiency of only 3.29 / 5 = 65.8%. This raises a new difficulty when designing a CAP program featuring a pipeline such as the one in Programs 3-27, 3-29 and 3-38, i.e. how to balance the length of each sequential operation composing the pipeline. The same load-balancing problem is found in processor pipelines, e.g. instruction fetch cycle (IF), instruction decode/register fetch cycle (ID), execution/effective address cycle (EX), memory access/branch completion cycle (MEM) and write-back cycle (WB) [Hennessy96, Chapter 3].

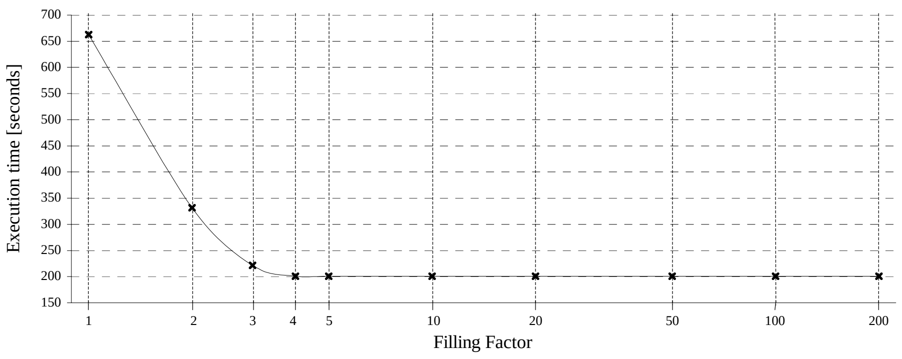


Figure 3-29. Measured execution times of the 5-stage pipeline for various filling factors

In a K-stage pipeline, where all the K stages are perfectly balanced and are mapped onto different PC's, the filling factor is given by Equation 3-2 customized in Equation 3-10 where S is the execution time of a stage and T is the transmission time of a token from one PC to another.

$$(FillingFactor - 1) \times S \geq (K-1) \cdot S + K \cdot T \qquad \textbf{(3-10)}$$

Which after reduction gives:

$$FillingFactor \geq K + \frac{K \cdot T}{S} \qquad \textbf{(3-11)}$$

Equation 3-11 ensures that the K PC's are 100% active throughout the execution of the high-level flow-controlled split-merge construct of Program 3-38 except during the pipeline fills and drains. In that condition, the execution time is given by Equation 3-7 customized in Equation 3-12.

$$PipelineExecutionTime = (N-1) \cdot S + K \cdot (S+T) \qquad \textbf{(3-12)}$$

The factor 'K x (S+T) - S' in Equation 3-12 represents the time that the pipeline fills and drains. The speed-up is given in Equation 3-13.

$$Speedup = \frac{N \cdot K \cdot S}{(N-1) \cdot S + K \cdot (S+T)} \qquad \textbf{(3-13)}$$

Which after reduction gives:

$$Speedup = K \times \left(1 - \frac{1}{1 + \frac{N \cdot S}{K \cdot (S+T) - S}}\right) \qquad \textbf{(3-14)}$$

Equation 3-14 clearly shows that the speed-up is less than K, since during the pipeline filling and draining time not all the K PC's are 100% active. Equation 3-15 gives the corresponding efficiency.

$$Efficiency = 1 - \frac{1}{1 + \frac{N \cdot S}{K \cdot (S+T) - S}} \qquad \textbf{(3-15)}$$

Our first attempt of a high-level flow-controlled split-merge construct (Program 3-38) requires one communication between the thread who runs the last sequential operation of the pipeline (or merge routine, line 18) and the thread who runs the first sequential operation of the pipeline (or split routine, line 14) for each iterations (provided that *ThreadA* thread and *ThreadB* thread are mapped onto two different PC's). Remember that the two original split-merge CAP constructs (Sections 3.8.6 and 3.8.7, Program 3-37) do not require any additional communications but they do not offer any flow-control mechanism. Although communications are mostly overlapped by computations (Equations 3-6, 3-7, 3-8 and 3-12), sending and receiving many small tokens such as those required for the high-level flow-control (Program 3-38) add an additional load on processors and network interfaces. Often, network interfaces and the processing power required for the TCP/IP protocol represent the bottleneck (Section 6.3.3). Therefore, even if flow-control tokens are small (between 50 and 100 Bytes), it is essential to reduce communications so as to alleviate these two scarce resources.

The second high-level flow-controlled split-merge construct in Program 3-39 requires much less communication between the thread who runs the merge routine, i.e. the *ThreadB* thread, and the thread who runs the split routine, i.e. the *ThreadA* thread. This is achieved by having a second *indexed parallel* CAP construct (lines 16-19) within the *for* CAP construct (line 12). Consequently only every each 10 tokens merged by the *ThreadB* thread using the *MergeFlowControl2* routine a *FlowControlT* token (line 19, *Out2* token) is sent back to the *ThreadA* thread (line 14) which uses the *SplitFlowControl2* routine to split the next 10 tokens feeding the pipeline. The empty inline sequential operation at line 14 forces the *ThreadB* thread to send the flow-control *Out2* token to the *ThreadA* thread. Otherwise the *SplitFlowControl2* routine would have been executed by the *ThreadB* thread which is different from the original program (Program 3-37, line 10) where the *Split* routine is executed by the *ThreadA* thread. The outer *indexed parallel* CAP construct (lines 7-10) with an appropriate filling factor, allow to saturate the most loaded PC's components in the pipeline. With an outer filling factor of 1, the pipeline is empty while the flow-control *Out2* token is being sent from the *ThreadB* thread to the *ThreadA* thread thus decreasing performance.

```
 1 operation ProcessAT::Operation1
 2   in TokenAT* InputP
 3   out TokenDT* OutputP
 4 {                                                                   filling factor1
 5   ThreadA.{ }
 6   >->
 7   indexed
 8     (int IndexFlowControl1 = 0; IndexFlowControl1 < 2; IndexFlowControl1++)
 9   parallel
10     (SplitFlowControl1, MergeFlowControl1, ThreadB, remote FlowControlT Out1(thisTokenP))
11   (
12     for (int IndexFlowControl2 = 0; IndexFlowControl2 < 10000/(2*10); IndexFlowControl2++)
13     (
14       ThreadA.{ }                                                      filling factor2
15       >->
16       indexed
17         (int IndexFlowControl3 = 0; IndexFlowControl3 < 10; IndexFlowControl3++)
18       parallel
19         (SplitFlowControl2, MergeFlowControl2, ThreadB, remote FlowControlT Out2(thisTokenP))
20       (
21         ProcessB.Operation1
22       ) // end indexed parallel                                                         pipeline
23     ) // end for
24   ) // end indexed parallel
25   >->
26   ThreadB.ReturnOutputToken;
27 } // end operation ProcessAT::Operation1
```

Program 3-39. High-level flow-controlled split-merge construct requiring much less communication between the thread who runs the merge routine, i.e. the *ThreadB* thread, and the thread who runs the split routine, i.e. the *ThreadA* thread

Contrary to the first high-level flow-controlled split-merge construct (Program 3-38), in the improved construct (Program 3-39) the filling factor is composed of 2 values. A first value named *filling factor$_1$* specifying the number of bunches of tokens (line 8), and a second value named *filling factor$_2$* specifying the number of tokens per bunch (line 17). Additional flow-control communications occur only every each filling factor$_2$ merged tokens. The maximum number of tokens that may simultaneously be in the pipeline is filling factor$_1$ multiplied by filling factor$_2$ tokens. With a filling factor$_2$ of 1, the improved high-level flow-controlled split-merge construct (Program 3-39) is equivalent to the first construct (Program 3-38).

Similarly to Equation 3-2, Equation 3-16 must be verified so that the thread who executes the slowest pipe stage of length S_{max} is 100% active during the execution of the flow-controlled split-merge construct except while the pipeline fills and drains.

$$(FillingFactor_1 - 1) \cdot FillingFactor_2 \cdot S_{max} \geq PipelineRoundTripTime - S_{max} \qquad \textbf{(3-16)}$$

Equations 3-7 and 3-8 giving the pipeline execution time are still valid with the second flow-controlled split-merge construct (Program 3-39), provided that Equation 3-16 is verified.

The CAP preprocessor is able to generate itself the two high-level flow-controlled split-merge constructs of Programs 3-38 and 3-39 from the original *parallel while* and *indexed parallel* CAP constructs (Sections 3.8.6 and 3.8.7) thus simplifying the task of writing parallel CAP programs. Program 3-40 shows how with the keyword '*flow_control*' the programmer can specify a split-merge CAP construct with a high-level flow control mechanism. The first flow-controlled construct in Program 3-38 is generated if the '*flow_control*' keyword contains a single argument, i.e. the filling factor. The second flow-controlled construct in Program 3-39 is generated if the '*flow_control*' keyword contains two arguments, i.e. the filling factor$_1$ and filling factor$_2$. Programmers must be aware that the two high-level flow-controlled *parallel while* and *indexed parallel* CAP constructs (Program 3-40) require additional communications. Performance may suffer if the pipeline empties, either due to the additional communication cost or due to too small filling factors.

```
operation ProcessAT::Operation1
  in TokenAT* InputP
  out TokenDT* OutputP
{
  ThreadA.{ }
  >->
  flow_control(20) // or flow_control(2, 10)
  indexed
    (int Index = 0; Index < 1000; Index++)
  parallel
    (Split, Merge, ThreadB, remote TokenDT Out1(thisTokenP))
  (
    ProcessB.Operation1
  ); // end indexed parallel
} // end operation ProcessAT::Operation1
```

Program 3-40. Automatic generation of the 2 flow-controlled split-merge constructs using the CAP preprocessor

Although the two flow-controlled *parallel while* and *indexed parallel* CAP constructs prevent a program from crashing, the high-level flow-control mechanism has several disadvantages:

- It requires additional communication that may slightly reduce performances.
- Experience has shown that it is quite difficult to adjust the filling factors so as to make the most loaded PC the bottleneck thus giving the best performance.
- Moreover, there are cases where the filling factors must be adapted to the split-merge construct input token making it even more difficult to set these factors. For example in the case of the pipelined parallel slice translation operation (Section 6.3.2), the filling factor giving the number of prefetched slices should depend on the split-merge construct input token, i.e. the image slice size and the zoom factor.
- The high-level flow-control mechanism does not solve the problem when running conditions change during the execution of the split-merge construct, i.e. when processor, memory, disk or network utilization change. For example this happens when several operations are concurrently executed on a same multi-PC environment or in other words when the computing resources must be shared among several users.
- Program 3-41 shows a CAP program where the high-level flow-control mechanism might fail. This is due to the fact that an inner split-merge CAP construct (lines 7-9) is called within an outer split-merge CAP construct (23-27). The filling factors of the inner *parallel while* CAP construct are computed using Equation 3-16 which assumes that the construct is executed one at a time. In Program 3-41 several *parallel while* instances, i.e. 4 in our program (line 23), are concurrently running, therefore too many tokens might accumulate in the pipeline. and memory might overflow. Although the programmer may prevent the application from crashing by adapting the filling factors of the inner split-merge construct (line 7) according to the filling factor of the outer split-merge construct (line 23), the program does not behave as desired. Supposing that *ProcessBT::ExtractAndVisualizeSlice* operation is an image slice extraction and visualization operation (Section 6.3.2) and *ProcessAT::VisualizeConsecutiveSlices* operation is an image slice translation operation extracting consecutive image slices using the *ProcessBT::ExtractAndVisualizeSlice* operation. Program 3-41 starts extracting the second, the third and even the fourth image slice while the first requested image slice is being extracted, requiring lots of memory and an important seek-time overhead. Instead of maximum 20, up to 80 tokens may be in circulation. Therefore due to the imbrication of two levels of flow-control mechanisms, Program 3-41 gives poor performance. The desired execution schedule, i.e. the execution schedule giving the best performance, would start extracting the second image slice as soon as the first image slice is about to be fully extracted so as to maintain the difference between the number of split tokens and the number of merged tokens (line 9) constant (2 x 10 tokens in Program 3-41), i.e. to maintain the pipeline filled or the most loaded PC 100% active. Unfortunately, such a schedule is not specifiable with the present high-level flow-control mechanism. However with standard inter-thread synchronization mechanisms, i.e. semaphores, the above optimal schedule is specifiable (for the sake of simplicity, the solution is not shown in this dissertation).

- Equations 3-2 and 3-16 assume that the execution model of a sequential operation follows the rule “first token in first token out”. However there are cases where a sequential operation may mix up the output token order using the *capDoNotCallSuccessor* and the *capCallSuccessor* CAP-library functions (Section 3.7.1), e.g. the *ExtentServerT::ReadExtent* sequential operation (Program 5-18) taken from the library of reusable parallel file system components of PS^2. In these cases, although the high-level flow-control filling factors are well sized, the pipeline may empty, degrading the performance. Remember that with the second high-level flow-controlled split-merge construct (Program 3-39), a new bunch of 10 tokens is generated (line 17) only when all the 10 tokens belonging to a same bunch are merged. If the pipeline (line 21) mixes up the token order and 9 tokens of each bunch are merged, i.e. a total of 2 x 9 = 18 tokens (line 8: 2 bunches of tokens), then, while the last 2 tokens are being merged, the pipeline is empty.

```
operation ProcessBT::ExtractAndVisualizeSlice
  in SliceExtractionRequestT* InputP
  out VoidTokenT* OutputP
{
  ThreadC.{ }
  >->
  flow_control(2, 10)
  parallel while
    (Split2, Merge2, ThreadC, local TokenDT Out1(thisTokenP))
  (
    ThreadD.ExtractSlice
    >->
    ThreadE.VisualizeSlice
  ); // end parallel while
} // end operation ProcessBT::ExtractAndVisualizeSlice

operation ProcessAT::VisualizeConsecutiveSlices
  in ConsecutiveSliceExtractionRequestT* InputP
  out VoidTokenT* OutputP
{
  ThreadA.{ }
  >->
  flow_control(4)
  indexed
    (int Index = 0; Index < 1000; Index++)
  parallel
    (Split1, Merge1, ThreadB, remote TokenDT Out1(thisTokenP))
  (
    ProcessB.ExtractAndVisualizeSlice
  ); // end indexed parallel
} // end operation ProcessAT::VisualizeConsecutiveSlices
```

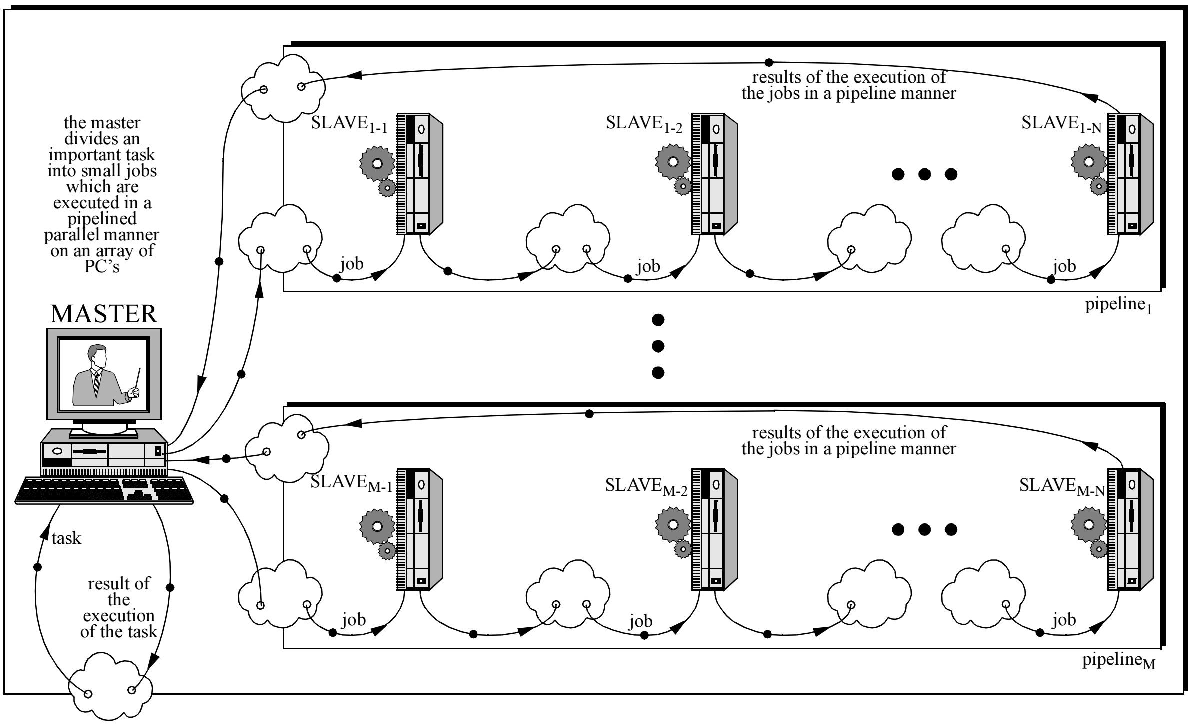
Program 3-41. CAP program where the high-level flow-controlled *parallel while* and *indexed parallel* constructs fail to give good performances

For these above reasons, the high-level flow-control mechanism is at the present time not completely satisfactory. Further investigation and experiences with the CAP tool will address these issues.

3.11 Issue of load balancing in a pipelined parallel execution

Figure 3-30 shows a pipelined parallel execution on a multi-PC environment. The master PC divides a large task into many small jobs which are executed in parallel on M different pipelines of execution. In this example, the pipeline of execution is composed of N different PC’s each performing a particular stage. After a job has performed the N stages of a pipeline, the result is sent back to the master PC where all the job results are merged into the result buffer. When all the jobs making up the task are completed, the result of the execution of the task is sent back to the client PC who requested the execution of the task. The master PC is responsible for distributing the jobs making up a task and for balancing the load amongst the M pipelines.

The graphical representation (DAG) of the pipelined parallel schedule depicted in Figure 3-30 is shown in Figure 3-31. The input of the parallel *ParallelProcessingServerT::PerformTask* operation is a task request, i.e. a *TaskT* token. Using a split-merge CAP construct, this task request is divided into many small jobs by the *Master* thread using the *SplitTaskIntoJobs* split routine. Then, each job is routed through one of the M pipelines. At the end of the M pipelines, the job results, i.e. the *JobResultT* tokens, are merged into a *TaskResultT* token using the *MergeJobResults* merge routine. When all the job results are merged, the result of the execution of the task, i.e. the *TaskResultT* token, is passed to the next operation.



results of the execution of the jobs in a pipeline manner
the master divides an important task into small jobs which are executed in a pipelined parallel manner on an array of PC's
MASTER
SLAVE$_{1-1}$
SLAVE$_{1-2}$
SLAVE$_{1-N}$
job
pipeline$_1$
SLAVE$_{M-1}$
SLAVE$_{M-2}$
SLAVE$_{M-N}$
task
result of the execution of the task
pipeline$_M$

Figure 3-30. Execution in a pipelined parallel manner on a multi-PC environment

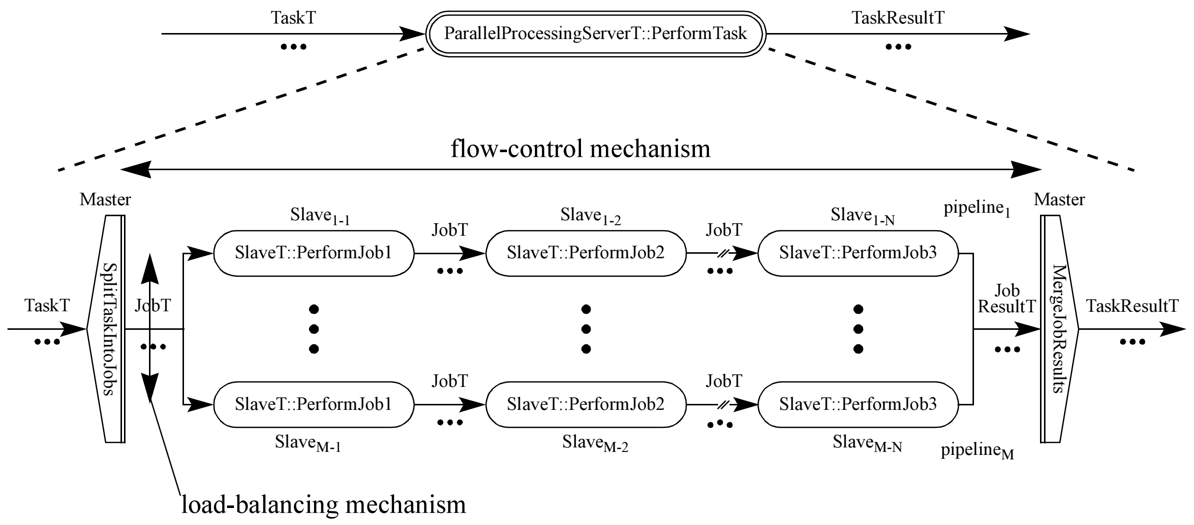


Figure 3-31. Graphical representation of a pipelined parallel execution. The horizontal arrow represents the flow-control mechanism and the vertical arrow represents the load-balancing mechanism.

A pipelined parallel execution such as the one depicted in the schedule of Figure 3-31 raises two orthogonal problems, i.e. flow-control and load-balancing. The mechanism of flow-control, presented in Section 3.10, attempts to regulate the flow of tokens through a single pipeline so that the split token rate is both high enough for maintaining the most loaded PC 100% of the time active and low enough for preventing tokens from accumulating in front of the most loaded PC and causing problems to the application.

A mechanism of load-balancing balances the load among the parallel pipelines (pipeline$_1$ to pipeline$_M$ in Figure 3-30 or 3-31), i.e. prevents the most loaded thread[1] in each of the parallel pipelines from being more loaded than the most loaded threads in another pipelines. A pipelined parallel execution on M pipelines each made of N threads is well balanced if Equation 3-17 is verified.

1. The load of a thread is defined as the percentage of elapsed time that this thread is executing a pipe stage.

$$\underset{i=1}{\overset{N}{\text{Max}}}\,(\text{load of thread}_{1\text{-}i}) = \underset{i=1}{\overset{N}{\text{Max}}}\,(\text{load of thread}_{2\text{-}i}) = \ldots = \underset{i=1}{\overset{N}{\text{Max}}}\,(\text{load of thread}_{M\text{-}i}) \qquad \textbf{(3-17)}$$

Such a mechanism is necessary when the utilization of the computing resources, e.g. processors, memory, disks, network, etc, is unequal between the PC's on which the parallel pipelines execute, i.e. when:

- the multi-PC environment is shared among several users, e.g. on a network of multi-user workstations.
- the multi-PC environment is made of different machines, e.g. Pentium Pro 200 MHz PC's, Pentium II 333MHz PC's, Pentium 90 MHz PC's, etc.
- the execution time of a pipe stage depends on the input token data, e.g. the number of computations in a Mandelbrot computation depends on the current value of the free variable.

Program 3-42 shows a CAP program where the master PC divides an important task into small jobs and statically distributes them in a round-robin fashion (line 8) amongst the worker slaves. The *GetJob* routine (lines 4-12) splits the input *TaskT* task into *JobT* jobs and computes the index of the slave (line 8) that will perform this job (line 29). The *MergeJobResult* routine (lines 14-17) merges the results of the jobs into a single *TaskResultT* token. The parallel *ParallelProcessingServerT::PerformTask* operation consist of iteratively getting a job, performing the job by a slave and merging its result into the output *TaskResultT* token using the high-level flow-controlled *parallel while* CAP construct (lines 25-30). Note that the selection of the slave (line 29) is done statically without balancing the loads of the slaves.

```
 1 const int NUMBER_OF_SLAVES = 5;
 2 const int FILLING_FACTOR_PER_SLAVE = 3;
 3
 4 bool GetJob(TaskT* inputP, JobT* prevSubtokenP, JobT* &subtokenP)
 5 {
 6   ... // Any C/C++ statements
 7   subtokenP = new JobT(...);
 8   subtokenP->SlaveIndex = prevSubtokenP ? (prevSubtokenP->SlaveIndex + 1) % NUMBER_OF_SLAVES : 0;
 9   ... // Any C/C++ statements
10
11   return (IsNotLastJob);
12 } // end GetJob
13
14 void MergeJobResult(TaskResultT* outputP, JobResultT* subtokenP)
15 {
16   // Any C/C++ statements
17 } // end MergeJobResult
18
19 operation ParallelProcessingServerT::PerformTask
20   in TaskT* InputP
21   out TaskResultT* OutputP
22 {
23   Master.{ }
24   >->
25   flow_control(NUMBER_OF_SLAVES * FILLING_FACTOR_PER_SLAVE)
26   parallel while
27     (GetJob, MergeJobResult, Master, local TaskResultT Out(thisTokenP))
28   (
29     Slave[thisTokenP->SlaveIndex].PerformJob
30   ); // end parallel while
31 } // end operation ParallelProcessingServerT::PerformTask
```

Program 3-42. Example of a CAP program where the jobs are statically distributed in round-robin manner among the slaves

The principle of a load-balancing mechanism consist of having a distinct high-level flow-controlled split-merge construct for each of the parallel pipelines. As soon as a slave completes a job, a new job is redirected to him. In that way the load is automatically balanced, since jobs are dynamically redirected to slaves completing their computation.

Program 3-43 shows the CAP program equivalent to Program 3-42 with a load-balancing mechanism. The first *indexed parallel* CAP construct (lines 10-13) generates *NUMBER_OF_SLAVES* parallel flows of execution consisting of a high-level flow-controlled *parallel while* CAP construct (line 15-17) distributing jobs to its slave. As soon as a slave completes a job, a new job is redirected to him thanks to the flow-controlled *parallel while* CAP construct (lines 15-18). The filling factor (line 15) computed using Equation 3-2 ensures that slaves are 100% of the time active and computations overlap communications.

To summarize, at the present time CAP does not provide any automatic tool for balancing the load between worker threads, i.e. there is no specific CAP construct. However, the design pattern shown in Program 3-43 provides application programmers with an excellent load-balancing mechanism.

```
const int NUMBER_OF_SLAVES = 5;
const int FILLING_FACTOR_PER_SLAVE = 3;

operation ParallelProcessingServerT::PerformTask
  in TaskT* InputP
  out TaskResultT* OutputP
{
  Master.{ }
  >->
  indexed
    (int SlaveIndex = 0; SlaveIndex < NUMBER_OF_SLAVES; SlaveIndex++)
  parallel
    (DuplicateTask, MergeJobResult1, Master, local TaskResultT Out1(thisTokenP))
  (
    flow_control(FILLING_FACTOR_PER_SLAVE)
    parallel while
      (GetJob, MergeJobResult2, Master, local TaskResultT Out2(thisTokenP))
    (
      Slave[SlaveIndex].PerformJob
    ) // end parallel while
  ); // end indexed parallel
} // end operation ParallelProcessingServerT::PerformTask
```

Program 3-43. Example of a CAP program where the jobs are dynamically distributed amongst the slaves according to their loads, i.e. the master balances the load amongst the slaves

3.12 Summary

This chapter presents the CAP computer-aided parallelization tool to simplify the creation of pipelined parallel distributed memory applications. Application programmers create separately the serial program parts and express the parallel behaviour of the program with CAP constructs. Thanks to the automatic compilation of parallel applications, application programmers do not need to explicitly program the protocols to exchange data between parallel threads and to ensure their synchronizations. The 8 predefined parallel CAP structures ensure that the resulting parallel program executes as an acyclic directed graph and is therefore deadlock free. Furthermore, due to its macro-dataflow nature, it generates highly pipelined applications where communication operations run in parallel with processing operations.

A CAP program can be easily modified by changing the schedule of operations or by building hierarchical CAP structures. CAP facilitates the maintenance of parallel programs by enforcing a clean separation between the serial and the parallel program parts. Moreover, thanks to a configuration text file, generated CAP applications can run on different hardware configurations without recompilation.

The CAP environment has been ported to a number of operating systems, including Microsoft Windows NT, Sun Solaris and Digital OSF Unix. Its underlying MPS communication library is portable but requires a socket interface for providing asynchronous *SendToken* and *ReceiveToken* routines.

The CAP approach works at a higher abstraction level than the commonly used parallel programming systems based on message passing (for example MPI [MPI94] and MPI-2 [MPI97]). CAP enables programmers to express explicitly the desired high-level parallel constructs. Due to the clean separation between parallel construct specification and the remaining sequential program parts, CAP programs are easier to debug and to maintain than programs which mix sequential instructions and message-passing function calls. Wilson et al. [Wilson96] provide a thorough overview of existing approaches using object oriented programming for supporting parallel program development.

The CAP computer-aided parallelization tool has been conceived by a colleague Dr. Benoit Gennart. However, in the context of this thesis, I have contributed to CAP by developing its portable runtime system incorporating among others a CAP-oriented message-passing system (MPS). Throughout the development of the parallel storage and processing server (PS^2), I tested the functionality of the CAP preprocessor and of the parallel CAP constructs. Based on my experience and the problems that I have encountered, I suggested new parallel constructs and new design patterns, i.e. combinations of parallel CAP constructs. These include solutions for the flow-control problem and a solution for incorporating an operation which has not an appropriate interface within the schedule of a parallel operation.

Chapter 4

Runtime System for CAP and PS^2

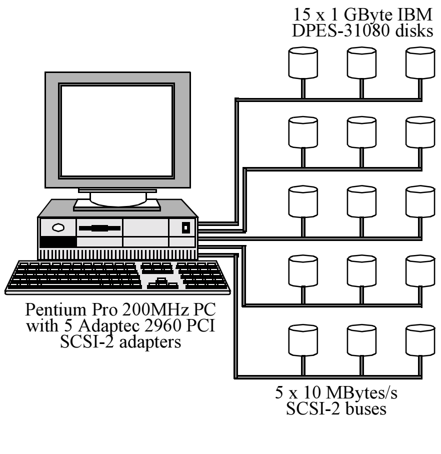
4.1 Introduction

This chapter is divided into two separate parts. The first part (Section 4.2) is devoted to analysing the performance of a multi-SCSI disk array hooked onto a same PC by several 10 MBytes/s SCSI-2 strings. The results of this section will enable performances of PS^2 applications to be predicted, in terms of I/O throughput and to reveal possible bottlenecks when accessing in parallel numerous disks. The second part of this chapter (Sections 4.3 to 4.5) is devoted to the token-oriented Message-Passing System MPS that implements the *SendToken* and *ReceiveToken* functions enabling distributed CAP applications to asynchronously transmit tokens from one address space to another through TCP/IP connections. Section 4.3 first introduces asynchronous socket programming using the Microsoft Windows Sockets 2.0 API. Thanks to this asynchronous network interface, token transmissions over TCP/IP connections occur entirely within pre-emptive contexts, i.e. asynchronously by calling completion routines, thus enabling a single thread to handle several TCP/IP connections simultaneously. Thanks to this crucial feature, CAP applications overlap communications with computations. After having introduced socket programming, Section 4.4 gives an overview of the token-oriented Message-Passing System and addresses the issue of serialization of tokens. Finally, Section 4.5 evaluates the performances of MPS for two types of communication patterns found in real CAP programs: an unpipelined token transfer (Section 4.5.3) and a pipelined token transfer (Section 4.5.4).

4.2 Performance evaluation of a multi-SCSI disk array

This section analyses the performance of a multi-SCSI disk array hooked onto the same PC by several 10 MByte/s SCSI-2 buses. The goal of these experiments is twofold. One is to measure the I/O raw performance in terms of throughput and latency and to analyse the scalability when increasing the number of disks per SCSI string and the number of SCSI strings. Since the low-level parallel file system components (PS^2) (Section 5.10) are based on top of the native Windows NT file system, the second goal of this section is to evaluate the overhead of the Windows NT file system in terms of processor utilization and to examine whether the Windows NT operating system introduces a bottleneck when accessing many disks in parallel.

For the experiments, we used a Mono-Pentium Pro 200 MHz PC with 5 Adaptec 2940 SCSI-2 PCI adapters and an array of 15 IBM DPES-31080 disk drives equally distributed amongst the 5 SCSI-2 buses, i.e. 3 disks per SCSI-2 bus (Figure 4-1). The PC runs the Microsoft Windows NT 4.0 operating system. A short specification of the IBM DPES-31080 is given in Figure 4-1.


15 x 1 GByte IBM DPES-31080 disks
Pentium Pro 200MHz PC with 5 Adaptec 2960 PCI SCSI-2 adapters
5 x 10 MBytes/s SCSI-2 buses

IBM DPES-31080 disk drive specification:
from http://www1.ibmlink.ibm.com/HTML/SPEC/goem1055.html

Formatted capacity: 1054 MBytes
Disks: 2
Data heads: 4:
Rotational speed: 5400 RPM:
Sector size: 512 Bytes
Average number of sectors per track: 95
Head switching time: 0.7 ms
Average seek time (typical read): 10.5 ms
Track to track seek time: 2.3 ms
512 KBytes segmented buffer with write cache and read lookahead

Figure 4-1. An array of 15 disks hooked on a Pentium Pro 200MHz PC with 5 SCSI-2 buses

Disks exhibit a linear behaviour, i.e. the time to read or write randomly-distributed blocks depends linearly on the I/O request size [Hennessy96, Chapter 6 and Patterson98, Chapter 8]. Therefore two parameters, i.e. *latency* (or response time) and *throughput*, are sufficient to model the linear behaviour of a disk or an array of disks using Equation 4-1.

$$\text{Disk Read Time} = \text{Response Time} + \text{Transfer Time} = \text{Latency} + \frac{\text{Request Size}}{\text{Throughput}} \qquad \textbf{(4-1)}$$

To access data, the disk must first position its head over the proper track. This operation is called a *seek*, and the time to move the head to the desired track is called the *seek time*. Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the *rotational delay*. Then, the average disk response time (Equation 4-1), is the sum of the seek time and the rotational delay (Equation 4-2). From the vendor specification (Figure 4-1), we can calculate the IBM DPES-31080 latency where the average rotational delay is half of the time of one rotation.

$$\text{Latency} = \text{Average Seek Time} + \text{Average Rotational Delay} = 10.5\text{ ms} + \frac{60\text{ s}}{\text{minute}} \times \frac{\text{minute}}{5400\text{ rot}} \times 0.5\text{ rot} = 16.1\text{ ms} \qquad \textbf{(4-2)}$$

We calculate the disk data transfer rate based on a complete disk scan policy as follows [Chen95, Chapter2] (Equation 4-3): once the head reaches and retrieves the first block, it retrieves all the adjacent data blocks in the same track. If a whole track has been retrieved, the head switches to the next platter but remains in the same cylinder. If the whole cylinder has been retrieved, it switches to the adjacent cylinder.

$$\text{Throughput} = \frac{\text{NHeads} \times \text{NSectorsPerTrack} \times \text{NBytesPerSector}}{\text{NHeads} \times \text{RotationTime} + \text{NHeads} \times \text{HeadSwitchTime} + \text{TrackToTrackSeekTime}} \qquad \textbf{(4-3)}$$

Given the vendor specification (Figure 4-1), the IBM DPES-31080 throughput is given by Equation 4-4.

$$\text{Throughput} = \frac{4 \times 95 \times 512}{4 \times 11.11\text{ ms} + 4 \times 0.7\text{ ms} + 2.3\text{ ms}} = 3.75\text{ MBytes/s} \qquad \textbf{(4-4)}$$

To evaluate latency and throughput of a given disk or disk array, we measure the delay when accessing in parallel randomly-distributed blocks distributed over *K* disks for increasing block sizes and we linearize the delay using a least-square fit. The slope of the linearized curve gives the throughput. The intersection with the vertical axis gives a measure of the latency.

Figure 4-2 reports the measured delay when reading in parallel 3 disks using the Windows NT native file system, i.e. NTFS. Up to a request size of 64 KBytes, the disk read time increases linearly. This is due to a historic limit of the x86 processors whose memory segments have a maximum size of 64 KBytes.

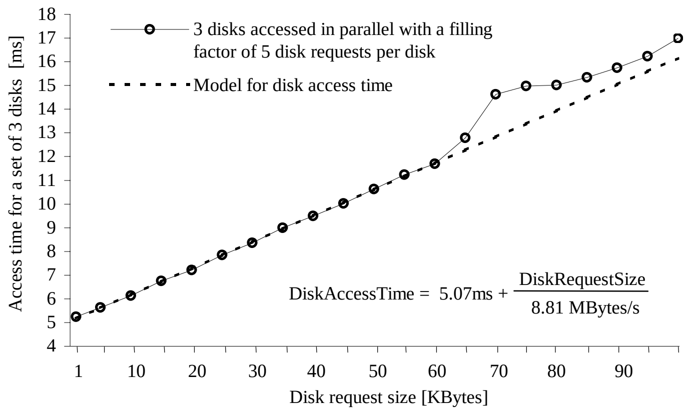


Figure 4-2. A disk array can be described using two numbers: latency and throughput

To measure the performance of the NT file system when accessing in parallel *K* files distributed over *K* disks, two types of experiments are conducted. The first experiment consists in accessing in parallel *K* disks using a low-level SCSI-2 block interface without the overhead of the NT file system. The second experiment consists of accessing the same *K* disks in parallel but through the NT file system. Comparing the results of these two tests enables the overhead induced by the NT file system to be evaluated.

For both experiments, asynchronous calls are used thus enabling a single thread to read or write in parallel from multiple disks, i.e. to issue several I/O requests to different disks. Moreover, to enhance performance, i.e. to benefit from I/O pipelining, several I/O requests are simultaneously issued to a same disk. The *filling factor* parameter corresponds to the number of outstanding I/O requests for a specific disk (Figure 4-2).

4.2.1 Performance measurement using a low-level SCSI-2 block interface

Figure 4-3 shows the measured throughputs and latencies when accessing (reading) a disk array of 1 to 15 disks hooked onto the same multi-SCSI PC using a low-level SCSI-2 block interface. Figure 4-3 reports a single disk throughput of 3.5 MBytes/s and a single disk latency of 13.77 ms confirming the vendor specification (Equations 4-2 and 4-4). With a single disk and a block size of 50 KBytes, an effective I/O throughput of 1.76 MBytes/s is reached. Due to the contention on a single SCSI-2 bus, the throughput is not linearly scalable up to 3 disks. This effect is particularly visible with a filling factor of a single I/O request. With a filling factor of 2 or more requests per disk, the throughput, thanks to I/O pipelining, can be increased by a factor of 1.25. Throughputs scale linearly when increasing the number of SCSI strings. With 5 SCSI strings, i.e. 15 disks, throughput increases by a factor of 5. With a 15 disk configuration and a block size of 50 KBytes, an effective I/O throughput of 23.51 MBytes/s is reached, giving a speedup of 13.35 and an efficiency of 88.98%.

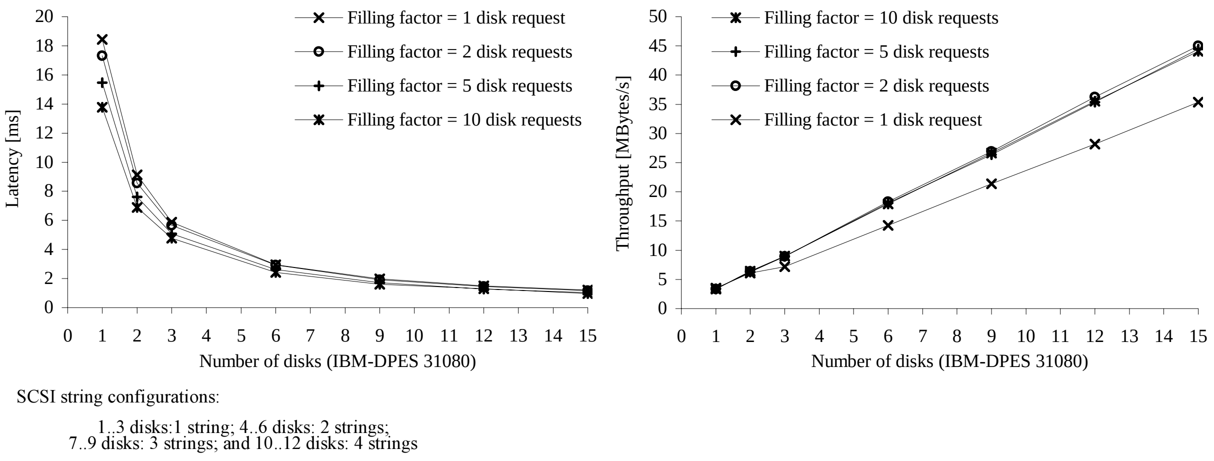


Figure 4-3. Measured disk array throughputs and latencies when accessing in parallel blocks randomly distributed over *K* disks hooked onto the same multi-SCSI PC using a low-level SCSI-2 block interface

According to the PS^2 methodology, disk accesses are overlapped with computations. Measuring the processor activity while accessing disks enables us to evaluate how well a PC can concurrently handle I/O requests and computation tasks. Figure 4-4 reports the measured processor utilization (mainly system and interrupt activities executed in a privileged processor mode) when accessing a disk array of 1 to 15 disks hooked onto the same multi-SCSI PC using a low-level SCSI-2 block interface. As expected, disk accesses do not consume much computing power (less than 14% with 15 disks). The SCSI-2 protocol is handled by the 5 PCI cards which transfer data to or from the computer's main memory using a DMA mechanism.

4.2.2 Performance measurement using the native NT file system

This second experiment consists of accessing in parallel *K* files distributed on *K* disks using the native NT file system with its cache disabled.

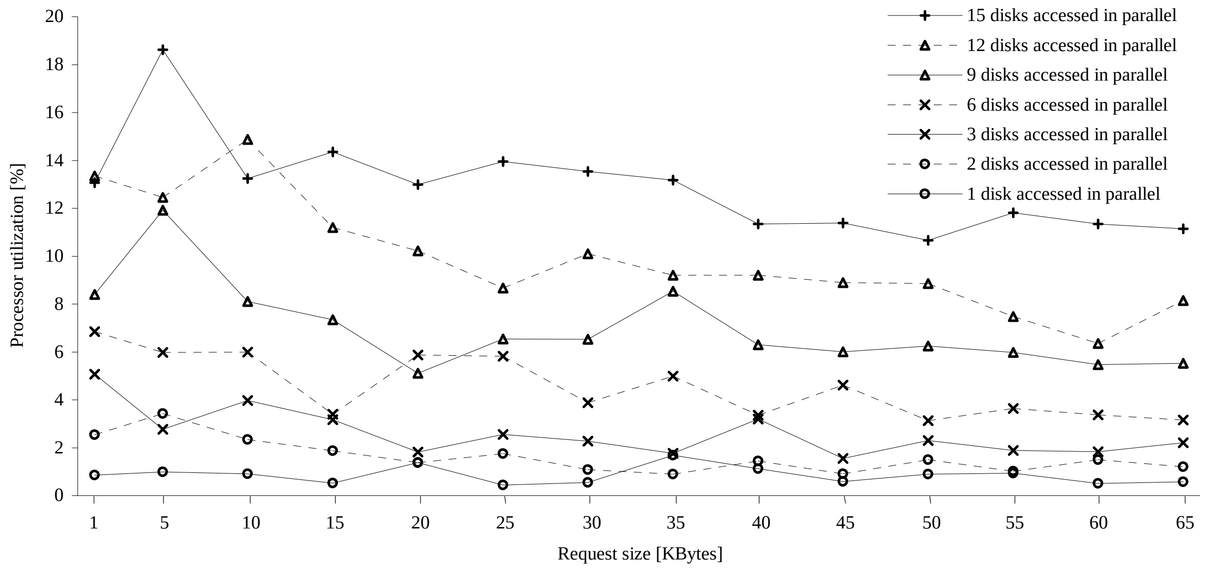


Figure 4-4. Measured processor utilization when accessing in parallel blocks randomly distributed over *K* disks hooked onto a same multi-SCSI PC using a low-level SCSI-2 block interface

Figure 4-5 shows the measured throughputs and latencies when accessing (reading) a disk array of 1 to 15 disks hooked onto the same multi-SCSI PC using the NT file system. By comparing Figures 4-5 and 4-3, we can conclude that the NT file system does not affect I/O performance. The NT file system is perfectly scalable up to 15 disks.

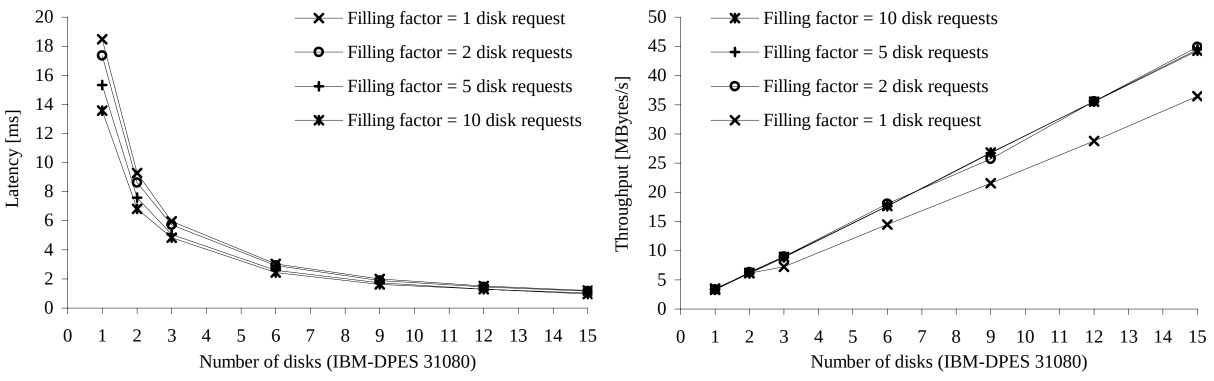


Figure 4-5. Measured disk array throughputs and latencies when accessing in parallel blocks randomly distributed over k disks hooked onto a same multi-SCSI PC using the NT file system

Moreover, the NT file system does not consume much processing power. Figure 4-6 reports the measured processor utilization (mainly system and interrupt activities executed in a privileged processor mode) when accessing a disk array of 1 to 15 disks hooked onto the same multi-SCSI PC using the NT file system. For a request size of 50 KBytes, less than 15% of the processing power is used for retrieving in parallel 23.51 MBytes/s from 15 NT files stored on 15 different disks.

4.3 Microsoft Windows Sockets 2.0 application programming interface

The sockets abstraction was first introduced in 1983 in the 4.2 Berkeley Software Distribution (BSD) Unix [Wright95, Chapter 15] to provide a generic and uniform application programming interface (API) to interprocess and network communication protocols such as the Internet protocols [Postel94], e.g. the User Datagram Protocol (UDP) [Postel80], the Transfer Control Protocol (TCP) [Postel81b] and the Internet Protocol (IP) [Postel81a]. Since then, all Unix versions, e.g. Solaris from Sun Microsystems, have adopted the sockets abstraction as their standard API for network computing. Besides the most popular Berkeley sockets API, there is the Transport

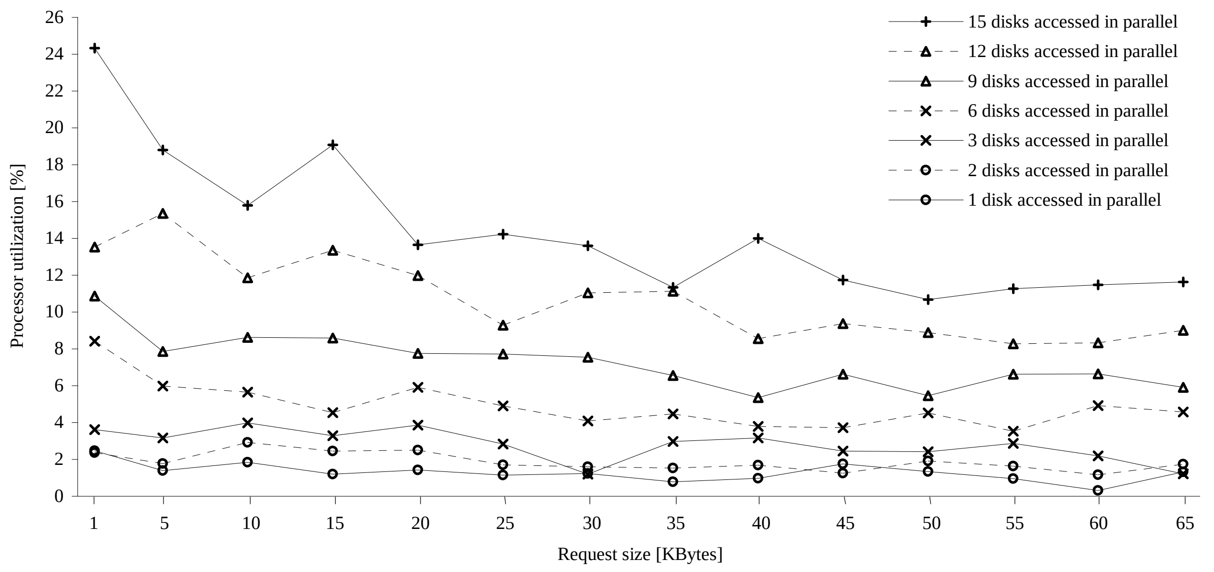


Figure 4-6. Measured processor utilization when accessing in parallel blocks randomly distributed over *K* disks hooked onto a same multi-SCSI PC using the NT file system

Layer Interface (TLI) originally developed by AT&T and sometimes called X/Open Transport Interface (XTI) recognizing the work done by X/Open, an international group of computer vendors that produce their own set of standards. All the programming details for both sockets and TLI are available in [Stevens90].

Microsoft has also adopted and adapted the sockets paradigm as a standard on its various operating systems, i.e. Windows 3.11, Windows 95/98 and Windows NT. The first Windows Sockets 1.1 application program interface released is centred around the UDP/IP and TCP/IP protocol stacks. With the emerging networking capabilities such as multimedia communications, Microsoft released in 1996 the Windows Sockets 2 API now available on Windows 95/98 and Windows NT 4.0. One of the primary goals of the Windows Sockets 2 API has been to provide a protocol-independent interface fully capable of supporting any number of underlying transport protocols. Comparing the Windows Sockets 1.1 API, the Windows Sockets 2 API extends functionality in a number of areas:

- Access to protocols other than UDP/IP and TCP/IP – The Windows Sockets 2 API enables the application to use the familiar socket interface to achieve simultaneous access to a number of installed transport protocols, e.g. DECNet or OSI TP4.
- Overlapped I/O with scatter/gather – The Windows Sockets 2 API incorporates scatter/gather capabilities and the overlapped paradigm for socket I/O, according to the model established in the Win32 environment.
- Protocol-independent name resolution facilities – The Windows Socket 2 API includes a standardized set of functions for querying and working with the myriad of name resolution domains that exist today, e.g. the Domain Name System (DNS) [Mockapetris87a, Mockapetris87b], SAP, NIS and the OSI directory service X.500 [Weider92].
- Quality of service – The Windows Sockets 2 API establishes conventions that applications use to negotiate required service levels for parameters such as bandwidth and latency.

To describe some important mechanisms of the message passing system, the remainder of this section presents the Windows Sockets 2 application programming interface and its advanced features used for improving performance. For the sake of simplicity, the interfaces have been simplified and only essential parameters are shown.

4.3.1 Definition of a socket

The term *socket* first appeared in the original TCP specification [Postel81b] referring to the combination of an IP address and a TCP port number. Later it became used as the name of the Berkeley-derived programming interface.

A socket is a communication endpoint, i.e. an object through which a Windows Sockets application sends or receives packets of data across a network. A socket has a type and is associated with a running process, and it may have a name. Sockets exchange data with other communication endpoints, e.g. sockets or TLI endpoints, in the same "communication domain" which uses for example the Internet protocol suite.

In the case of the Internet address family, two types of sockets are available:

- Stream sockets providing sequenced[1], reliable, flow-controlled, two-way, connection-based data flows without record boundaries, i.e. byte streams. Stream sockets use the underlying TCP/IP protocol stack and may have a name composed of a 32-bit IP address and a 16-bit TCP port number.
- Datagram sockets supporting record-oriented data flows, i.e. datagrams, which are connectionless, unreliable buffers of a fixed (typically 8 KBytes) maximum length. Datagrams are not guaranteed to be delivered and may not be sequenced as sent or unduplicated[2]. Datagram sockets use the UDP/IP protocol stack and may have a name composed of a 32-bit IP address and a 16-bit UDP port number.

Both kinds of sockets are bi-directional: they are data flows that can be communicated in both directions simultaneously, i.e. full-duplex.

To implement a portable message passing system for transmitting CAP tokens (Section 3.4) from one address space to another, two different possibilities exist: either using datagram sockets or stream sockets. Since CAP needs a reliable means of communication, stream sockets were preferred thus avoiding designing a mechanism that provides sequenced, reliable, unduplicated data flows on top of unreliable datagram sockets. The remainder of this section assumes that sockets are always stream sockets.

4.3.2 Parameters of a socket

Three parameters enable programmers to significantly modify the sending and receiving behaviour of stream sockets affecting the TCP/IP performance:

- SO_SNDBUF: Specify the size in Bytes of the transmit buffer.
- SO_RCVBUF: Specify the size in Bytes of the receive buffer. Indirectly this parameter changes the window size of the TCP sliding window protocol [Stevens96, Section 20.4].
- TCP_NODELAY: This option enables or disables the Nagle algorithm [Nagle84] [Stevens96, Section 19.4] which reduces the number of small TCP/IP packets, called *tinygrams*, congestioning wide area networks (WAN). This algorithm says that a TCP connection can only have one outstanding small TCP segment that has not yet been acknowledge. No additional small segments can be sent until the acknowledgment is received. Instead, small amounts of data are collected by TCP and sent in a single segment when the acknowledgment arrives. The beauty of this algorithm is that it is self-clocking: the faster ACKs come back, the faster the data is sent. But on a slow WAN, where it is desired to reduce the number of tinygrams, fewer segments are sent. However there are situations where the Nagle algorithm needs to be turned off. The classic example is the X Window System server: small messages (mouse movements) must be delivered without delay to provide real-time feedback for interactive users doing certain operations.

[Mogul93] shows some results for file transfer between two workstations on an Ethernet, with varying sizes for the transmit buffer and receive buffer. For a one-way flow of data such as a file transfer, it is the transmit buffer on the sending side and the size of the receive buffer on the receiving side that matters. The common default of 4 KBytes for both is not optimal for an Ethernet. An approximate 40% increase in throughput is seen by just increasing both buffers to 16 KBytes. Similar results are shown in [Papadopoulos93].

4.3.3 Asynchronous gather send and asynchronous scatter receive

The *scatter/gather* send and receive follow the overlapped paradigm established in the Win32 environment and enable data to be asynchronously sent or received through connected stream sockets. Thanks to these two new functions not found in the first Windows Socket 1.1 API, a single thread may simultaneously execute several overlapped I/O requests thus alleviating the task of writing networking applications and improving performance. This feature is extensively used in the token-oriented message-passing system for handling several connections simultaneously.

Program 4-1 shows the two overlapped gather/scatter send and receive routines.

The first argument (lines 11 and 18) is a connected stream socket through which data is sent/received. The second argument (line 12 and 19) is an array of *WSABUF* buffers enabling to send/receive data from/into several disconnected buffers, i.e. scatter/gather or vectored I/O. The third argument (lines 13 and 20) gives the number of

1. Sequenced means that packets are delivered in the order sent.
2. Unduplicated means that you get a particular packet only once.

```
 1 struct WSABUF {
 2   int Len;
 3   char* BufP;
 4 }; // end struct WSABUF
 5
 6
 7 typedef void (*CompletionRoutineT) (int bTransferred,
 8                                     void* argP);
 9
10
11 void WSASend(SOCKET socket,
12              WSABUF* BufferP,
13              int BufferCount,
14              void* argP,
15              CompletionRoutineT writeCompletion);
16
17
18 void WSARecv(SOCKET socket,
19              WSABUF* BufferP,
20              int BufferCount,
21              void* argP,
22              CompletionRoutineT readCompletion);
```

Program 4-1. Overlapped gather send and overlapped scatter receive (simplified interface)

WSABUF buffers in the *BufferP* array. The thread performing one of the two operations is not blocked, the I/O operation is started and returns immediately. It overlaps with the thread computation. Once the I/O completes and the thread who initiated the I/O operation is in an alertable wait state, the completion routine (lines 15 and 22) is called with the provided argument (lines 8, 14 and 21) and the number of bytes transferred (line 7). In the case of a *WSASend*, the I/O completion occurs only when all the data that are in the *WSABUF* buffers are sent or the connection is closed. In the case of a *WSARecv*, the I/O completion occurs when the *WSABUF* buffers are filled, the connection is closed or internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the *WSABUF* buffers, the completion indication occurs.

As explained in Section 4.4.1, the scatter and gather interfaces (Figure 4-7) enable the number of memory to memory copies to be reduced when packing and unpacking CAP tokens.

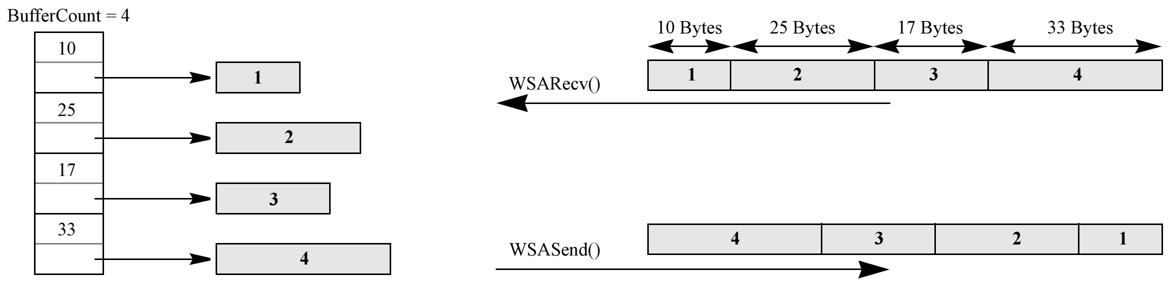


Figure 4-7. Gathering and scattering data

Contrary to the UDP protocol (datagram sockets), the TCP protocol (stream sockets) has no notion of packets with boundaries. A TCP/IP connection is a two-way reliable flow-controlled stream of bytes. The receiver has no information about the number of bytes that the sender is transmitting and how its *WSABUF* buffers are scattered, i.e. the number of *WSABUF* buffers and their sizes. Figure 4-8 depicts a TCP/IP transmission example where the sender with a single call to the *WSASend* routine transmits a whole group of scattered *WSABUF* buffers, while the receiver has to implement a mechanism that repeatedly calls the *WSARecv* routine in order to fill the destination scattered *WSABUF* buffers with the incoming data. Remember that a read completion may occur regardless of whether or not the incoming data fills all the destination *WSABUF* buffers.

To transfer CAP tokens through a TCP/IP connection, a 3-step mechanism indicating to the receiver the number of scattered buffers and their sizes has been implemented (Figure 4-9). For the remainder of this section, the term *packet* (or socket packet) is used to refer to the array of *WSABUF* buffers sent with a single *WSASend* call and received using the appropriate 3-step mechanism that repeatedly calls the *WSARecv* routine.

Thanks to the asynchronous *WSASend* and *WSARecv* routines, data transmissions occur entirely within pre-emptive contexts, i.e. by calling completion routines (Programs 4-3 and 4-4), thus enabling a single thread to handle several connections simultaneously.

4.4 The token-oriented message-passing system

This section describes the token-oriented message-passing system named MPS developed within the context of the CAP computer aided parallelization tool (Chapter 3). It must provide the CAP's runtime system with:

- A portable communication environment. The problem of portability has been addressed by isolating platform-dependent code within a few number of files. A high-level platform-independent stream socket kernel has been devised providing simple robust efficient functions for creating passive and active sockets

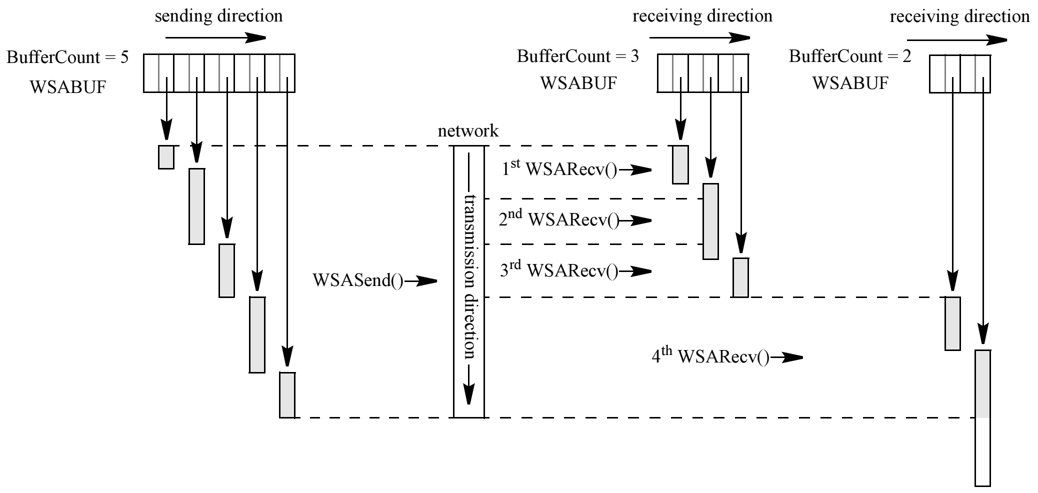


Figure 4-8. The TCP protocol has no notion of packets. There is no correspondence between the *WSASend* and *WSARecv* calls.

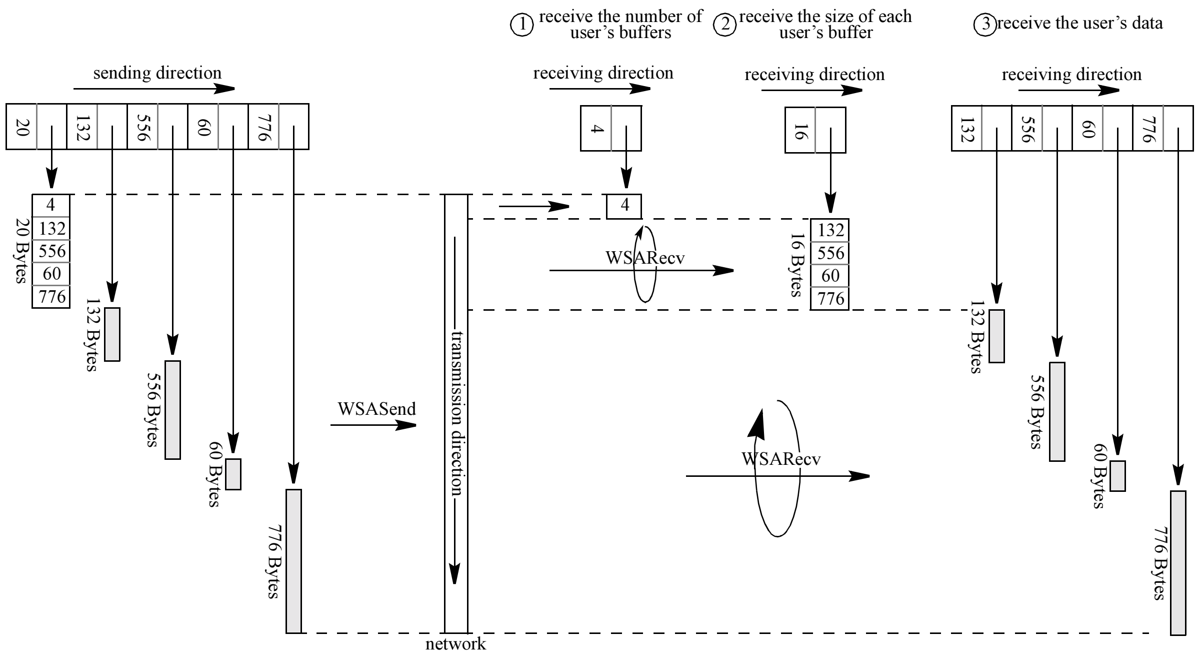


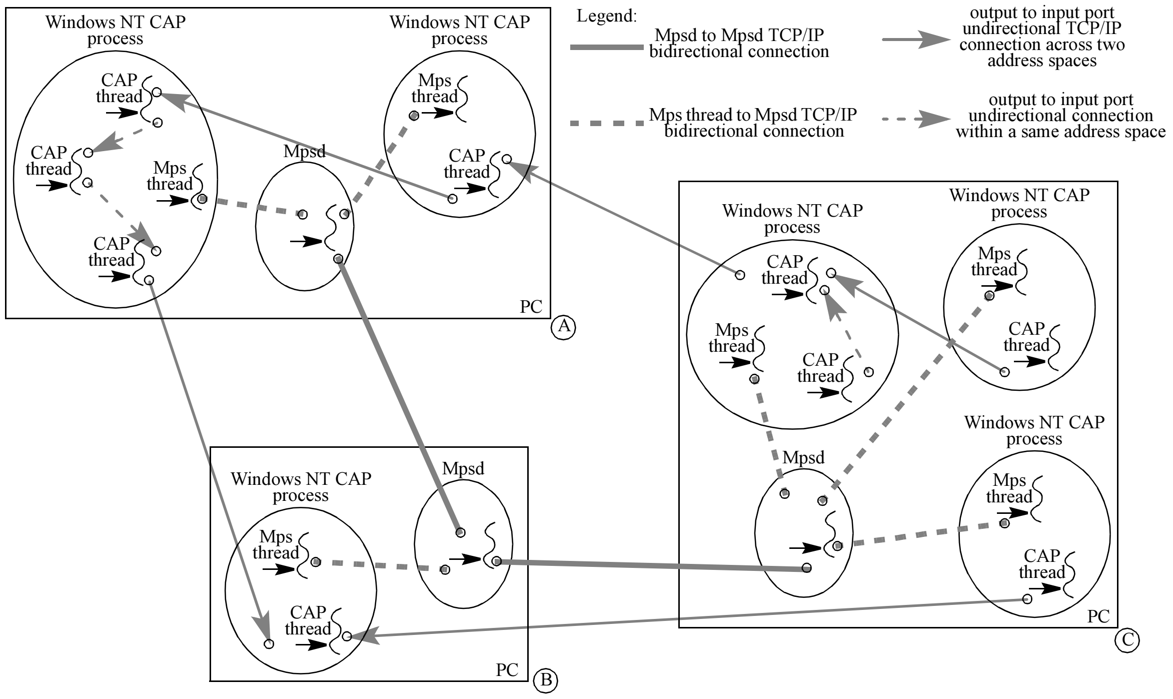
Figure 4-9. The 3-step mechanism that enables a scattered packet to be transmitted from one address space to another using a stream socket (TCP/IP protocol) and the overlapped gather/scatter *WSASend* and *WSARecv* routines

(see TCP passive and active open in [Postel81b] and the sockets terminology in [Microsoft96]), asynchronously sending C/C++ structures to active sockets and asynchronously receiving C/C++ structures from active sockets. The interest and the efficiency of the socket kernel resides in the fact that all network events, i.e. new incoming connection, lost connection, connection established, data received, data sent, etc., are asynchronously handled by a single thread, the message-passing system thread, called the *Mps thread.* All low-level tedious error-prone platform-dependent communication mechanisms such as the packet transfer mechanism (Program 4-9), the C/C++ structure serialization (Section 4.4.1), the coalescence of C/C++ structures into a same socket packet for improving performance (Section 4.5), are located within a same file enabling to optimize the code for a particular platform.

At the present time, MPS runs on Sun Solaris, Microsoft Windows NT and Digital Unix OSF 1. However, most advanced features such as the zero-copy serialization mechanism (Section 4.4.1, address-pack serialization) are only available on Windows NT platforms.

- A means for a thread to asynchronously receive tokens from any threads independently whether they are in the same address space, in a different address space but on the same PC, or located on another PC, i.e. communication of any threads to 1 thread. In the MPS terminology such a communication endpoint is called an *input port* comprising a FIFO queue (Figure 3-5, input token queue) where received tokens are inserted. Before a thread can receive tokens from an input port, it must be registered to the message-passing system. That is, the CAP runtime system must name the input port using a string of characters and a 32-bit instance number uniquely identifying the connection endpoint so that any other threads in the MPS network can use this logical name for sending tokens to this input port. The creation and the destruction of the input ports are completely dynamic. A Windows NT process can create and delete input ports whenever it wants during execution.
- A means for a thread to asynchronously send tokens to any threads independently whether they are in the same address space, in a different address space but on the same PC, or located on another PC. In the MPS terminology such a communication endpoint is called an *output port*. Before a thread can send tokens through an output port, it must be connected to an input port. That is, the CAP runtime system must open the output port by providing the name of the input port previously registered. At opening time, the message-passing system resolves the input port name into an IP address and a 16-bit TCP port number, and opens a TCP/IP connection between the output port and the input port. If the input port is located in the same address space as the output port, then instead of using a TCP/IP connection, the shared memory along with an inter-thread synchronization mechanism is used avoiding to serialize the tokens, i.e. only pointers are copied. Any number of output ports can be connected to a same input port. The creation and the destruction of the output ports are completely dynamic. A Windows NT process can create and delete output ports whenever it wants during execution.

A name resolution system, such as the Internet Domain Name System [Mockapetris87a, Mockapetris87b], has been implemented so that the message-passing system can resolve an input port name into an IP address and a 16-bit TCP port number when an output port is opened. Each PC participating in the parallel computation runs a message-passing system daemon, called the *Mpsd*, connected to one or several other Mpsd's so as to form a network of contributing PC's (Figure 4-10). Each Windows NT CAP process is connected to its local MPS daemon to exchange control messages.



Legend:
Mpsd to Mpsd TCP/IP bidirectional connection
output to input port undirectional TCP/IP connection across two address spaces
Mps thread to Mpsd TCP/IP bidirectional connection
output to input port undirectional connection within a same address space
Windows NT CAP process
CAP thread
Mps thread
Mpsd
PC
A
B
C

Figure 4-10. Message-passing system TCP/IP connections. On each PC runs a MPS daemon connected to one or several other Mpsd so as to form a network of contributing PC's. Each Windows NT CAP process is connected to its local Mpsd to exchange control messages.

When an input port is created, a passive TCP connection is opened listening for incoming network connections and a control message is sent to the local MPS daemon to register the input port name with the newly allocated TCP port number, i.e. to notify the MPS daemon that an input port is created on the following TCP port. When an output port is created two different cases arise. Either the input port is in the same address space or in a distinct address space. If the input port is in the same address space, then the output port is merely a pointer to the input port object and sent tokens are transferred using the shared memory, i.e. only pointers are copied. On the other hand, if the input port is located on a separate address space, then a control message is sent to the local MPS daemon for seeking its input port IP address and TCP port number. The local MPS daemon first searches in his internal list of registered input ports and in the case the port is not found, a control message is sent to all MPS daemons for seeking the input port. Once found, the output port establishes the TCP connection with the corresponding input port.

The network of MPS daemons is also used for spawning Windows NT processes on remote PC's when a CAP program is launched with a configuration file (Section 3.6, Program 3-6).

To evaluate the performance of CAP programs, the message-passing system provides numerous counters for measuring the incoming and outgoing token and network rates. Moreover, MPS is able to automatically sample these counters in constant intervals and maintain a history list. Figure 4-11 shows such a history list sampled at 500ms intervals when executing a CAP program.

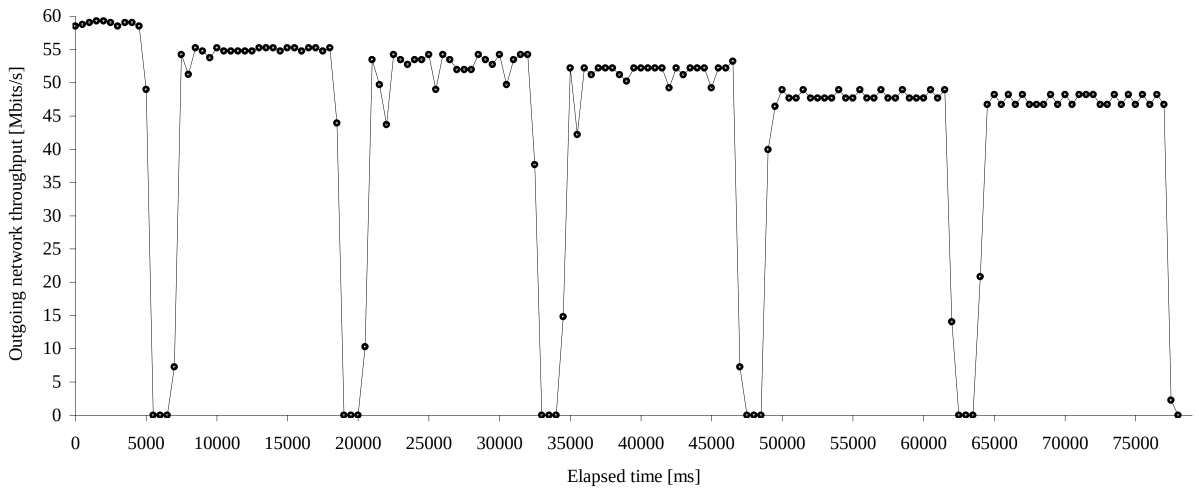


Figure 4-11. Outgoing network throughput with a sampling factor of 500ms

4.4.1 Serialization of CAP tokens

To move a C/C++ structure, i.e. a CAP token, from one address space to another, the structure's data should be prepared for transfer, sent over a communication channel linking the two address spaces, received in the destination address space, and finally restored the data into a C/C++ structure identical to the original one. This process of preparing a data structure for transfer between two address spaces is called *serialization.*

The serialization process consists of these four steps:
- The *packing* step prepares the data structure for the transfer.
- The *sending* step sends the prepared data structure over a communication channel, i.e. a TCP/IP stream socket for MPS, linking two address spaces.
- The *receiving* step receives the data in the destination address space.
- The *unpacking* step restores the received data into the original C/C++ structure in the destination address space.

The token-oriented message-passing system implements two types of serialization, the *copy-pack* serialization and the *address-pack* serialization. The copy-pack mechanism uses a temporary buffer for transmitting the structure's data thus enabling two computers with different data encoding (little endian, big endian, etc.), i.e. heterogeneous environment, to communicate. However, the use of a temporary buffer involves two memory-to-memory copies, one at the sending side and one at the receiving side. In the case where the two computers use the same data encoding, i.e. homogeneous environment, the address-pack serialization avoids these two copies by copying the addresses to the structure's data into a list of pointers to memory blocks (Program 4-1, *WSABUF* buffers).

When packing a token, the CAP's runtime system adds its *type index*, which is a 32-bit value identifying the type of the transmitted token so that in the destination address space, the corresponding unpack function is called. The unpack function creates a token of the original type and copies the received data into it.

In Figure 4-12, the copy-pack packing mechanism copies all the token's fields into a single memory buffer called the *transfer-buffer*. The CAP tool automatically generates instructions for packing the predefined C/C++ types, e.g. int, float, double, char. For user defined C/C++ structures and pointers, the CAP's runtime system calls, thanks to C++ function overloading, the appropriate user defined pack routine that recursively copies the structure's data fields into the same memory block. At the receiving address space, the token is first created based on the received type index and then data is copied from the transfer-buffer into the token's fields using the corresponding unpack routines.

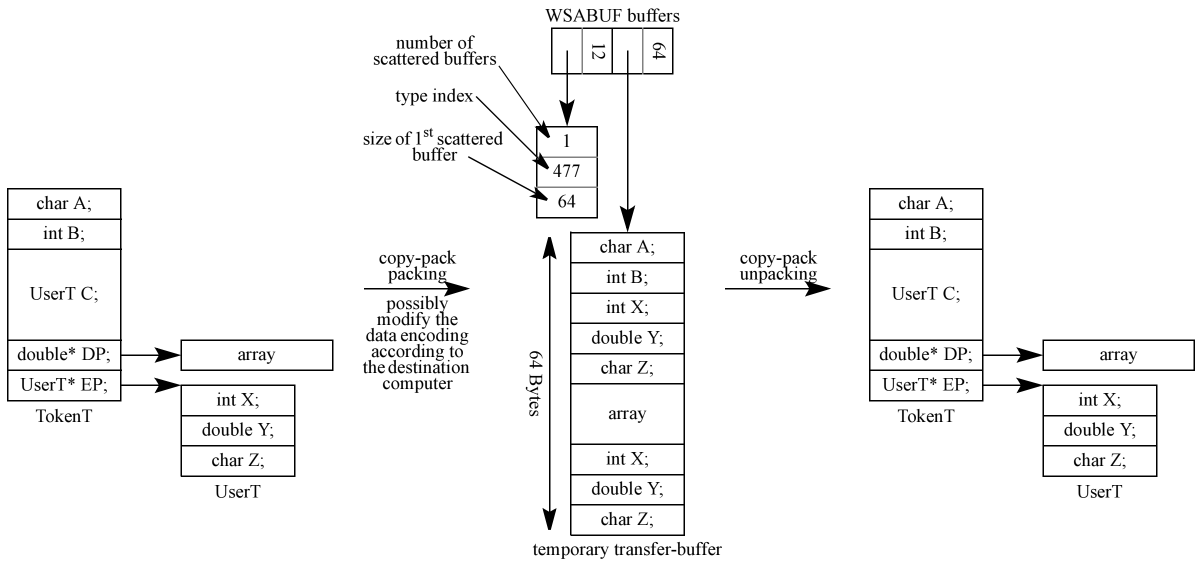


Figure 4-12. The copy-pack serialization uses a temporary transfer buffer to copy the structure's data so that the data encoding can be modified

In Figure 4-13, the address-pack packing mechanism creates a list of pointers to memory blocks (*WSABUF* buffers), and the overlapped gather send (Program 4-1, lines 13-17) handles the transfer of these scattered memory blocks. The CAP tool automatically generates instructions for packing the token's memory block. For user's defined C/C++ structures and pointers, the CAP's runtime system calls, thanks to C++ function overloading, the appropriate user's defined pack routine that recursively copies all the pointers to the structure's memory blocks into the list of *WSABUF* buffers. At the receiving address space, the token is first created based on the received type index, then a list of pointers to memory blocks is created using the corresponding unpack routines, and the overlapped scatter receive (Program 4-1, lines 20-24) handles the transfer. Since the memory block-oriented transfer process also copies pointers, e.g. in Figure 4-12 the *DP* and *EP* pointers, an additional *address-restore* stage restoring the clobbered pointers is required. Note that it is necessary to restore the pointer to the virtual function table in the case of a C++ class containing at least one virtual function.

Program 4-2 shows a *PrimeNumbersT* token comprising a user's defined *ArrayOfIntsT* object (line 66). To serialize such a C/C++ structure, the CAP's runtime system needs 4 routines. An address-list-size routine (lines 59-62) calculating the number of scattered memory blocks necessary for transmitting the whole structure's data. An address-pack routine (lines 22-29) copying the pointers to the structure's memory blocks (line 27) and their sizes (line 26) into the list of memory buffers to send (line 23 and Program 4-1 line 14). An address-unpack routine (line 31-49) allocating the internal structure's memory blocks (line 37) and copying their pointers into the list of memory buffers where the incoming network data will be stored (line 32 and Program 4-1 line 21). And finally, an address-restore routine (lines 51-57) restoring the clobbered pointers (line 55) previously saved in the unpack-routine (line 45). For transmitting *PrimeNumbersT* tokens, the CAP tool automatically generates appropriate calls to the 4 user's defined serialization routines and assigns a unique token type index.

To alleviate the task of writing these 4 serialization routines, which is often tricky, error prone and hard to debug, an ongoing extension to the CAP tool will automatically generate these 4 functions based on the C/C++ structure declaration along with a CAP specification.

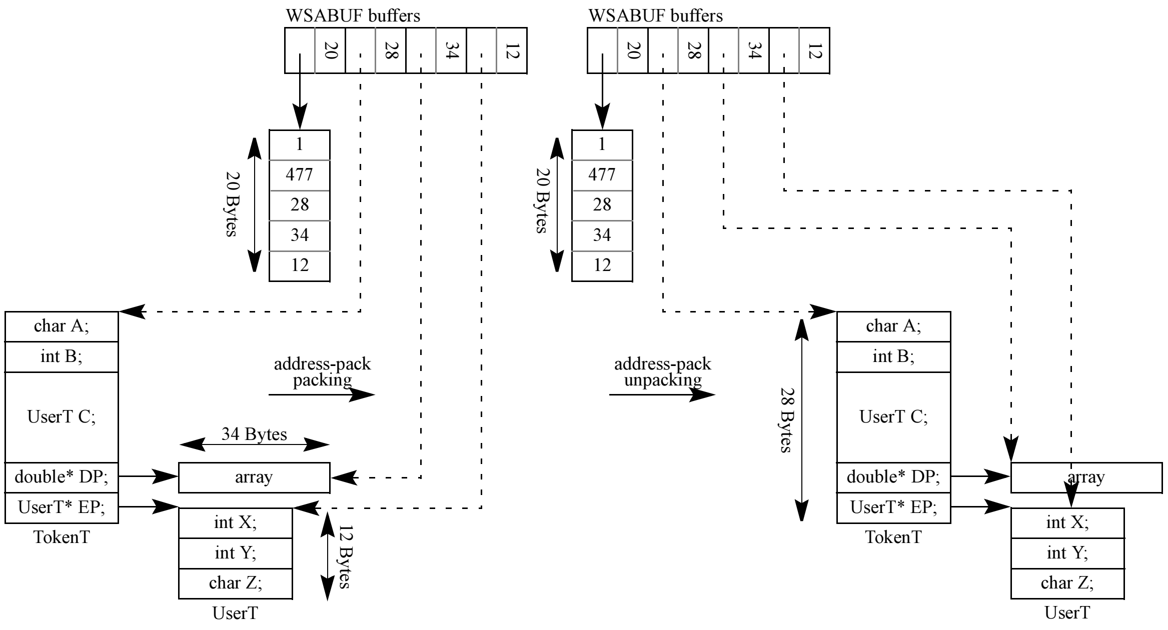


Figure 4-13. The address-pack serialization uses a list of scattered buffers (*WSABUF* buffers) for sending and receiving the structure's data with no memory-to-memory copy

```
class ArrayOfIntsT
{
public:
  ArrayOfIntsT();
  ~ArrayOfIntsT();

public:
  int Size;
  int* ArrayP;
}; // end class ArrayOfIntsT

ArrayOfIntsT::ArrayOfIntsT()
  : Size(0), ArrayP(0)
{
} // end ArrayOfIntsT::ArrayOfIntsT

ArrayOfIntsT::~ArrayOfIntsT()
{
  delete ArrayP;
} // end ArrayOfIntsT::~ArrayOfIntsT()

void capAddressPack(int& listSize,
                    WSABUF* &bufferP,
                    ArrayOfIntsT* udP)
{
  bufferP->Len = udP->Size;
  bufferP->BufP = (char*) udP->ArrayP;
  bufferP++; // Points to next memory block
} // end capAddressPack

void capAddressUnpack(int& listSize,
                      WSABUF* &bufferP,
                      ArrayOfIntsT* udP)
{
  if( bufferP->Len )
  {
    udP->ArrayP = new int[bufferP->Len];
  }
  else
  {
    udP->ArrayP = 0;
  } // end if

  bufferP->BufP = (char*) udP->ArrayP;
  thrAddressSave(listSize,
                 bufferP,
                 (void*) &(udP->ArrayP));
  bufferP++; // Points to next memory block
} // end capAddressUnpack

void capAddressRestore(int& listSize,
                       WSABUF* &bufferP,
                       ArrayOfIntsT* udP)
{
  thrAddressRestore(listSize, bufferP);
  bufferP++; // Points to next memory block
} // end capAddressRestore

int capAddressListSize(ArrayOfIntsT* udP)
{
  return 1; // size of list
} // end capAddressListSize

token PrimeNumbersT
{
  ArrayOfIntsT PrimeNumbers;
  int NumberOfPrimeNumbers;
}; // end token PrimeNumbers
```

Program 4-2. To serialize a user's defined C/C++ structure, the CAP's runtime system needs 4 routines: an address-list-size, an address-pack, an address-unpack and an address-restore routine

4.5 Performance evaluation of the token-oriented message-passing system

The communication mechanism comprising the token-oriented message-passing system, the Windows Sockets library, the TCP/IP stack protocol, the network card adapter and the Fast Ethernet network, is a potential bottleneck that may limit scalability of distributed-memory CAP applications when increasing the number of cooperating PC's or the number of contributing disks in the case of PS^2 applications.

This section evaluates the overlapped Windows Sockets application programming interface and the token-oriented message-passing system when faced with two communication patterns as they can be found in real CAP programs. The first pattern arises when a single token flows from one PC to another (Section 4.5.3). Since, in this case, the communication mechanism cannot pipeline network I/O requests, this experiment mainly evaluates performance in terms of latency. The second pattern arises when a bunch of tokens is transferred from one PC to another (Section 4.5.4). In this case, the communication mechanism pipelines the network I/O requests thus decreasing latency, and performance is mainly affected by throughput.

The aim of these two experiments are the following:

- Measure the performance of the Windows Sockets API and the token-oriented message-passing in terms of effective throughput, i.e. the number of bytes that the communication mechanism transmits per unit of time. These results will set a first limit of scalability in the case where the token throughput between PC's is the bottleneck.
- Measure the performance of the Windows Sockets API and the token-oriented message-passing system in terms of processor utilization, i.e. the percentage of elapsed time that processors spend sending or receiving data through a TCP/IP connection. Although PCI network card adapters, that equip most recent server PC's, use a DMA mechanism to move data between the main memory and the network card's internal buffers, these results will demonstrate that handling a TCP/IP connection is a processor consuming task setting a second limit of scalability in the case where processors are bottlenecks because they handle both user's computations (sequential operations) and communications occurring entirely within pre-emptive contexts.
- Evaluate the cost of the token-oriented message-passing system compared with the raw Windows Sockets communication mechanism.
- Demonstrate how coalescing tokens into a same socket packet may boost the performance of the message-passing system not only in terms of throughput, but also in terms of processor utilization.
- Understand how the overlapped Windows Sockets API and the underlying TCP/IP connection behave under various parameters (Section 4.3.2) and under various I/O request sizes. This knowledge will enable the token-oriented message-passing to be tuned so as to increase the effective throughput and to decrease the processor utilization.

When measuring networking performances power of 2 are used, i.e. KBytes stands for 1024 Bytes, MBytes stands for 1024 x 1024 Bytes and Mbits/s stands for 1024 x 1024 bits per second.

4.5.1 Theoretical maximum TCP/IP performance over a 100 Mbits/s Fast-Ethernet network

Stevens in his book [Stevens96, pp. 354–356] gives a comprehensive look at how to compute the theoretical maximum throughput that the TCP/IP stack protocol can handle over a 100 Mbits/s Fast Ethernet link. Table 4-1 gives the numbers of bytes exchanged for a full-sized –DATA– segment and an –ACK– segment. This table takes into account the overhead dues to: the Fast Ethernet preamble, the PAD bytes that are added to the acknowledgement so as to meet the minimum required Fast Ethernet packet size, the CRC and the minimum interpacket gap (916 ns, which equals 12 bytes at 100 Mbits/s).

We first assume that the sender transmits two full-sized data segments and then the receiver sends an –ACK– for these two data segments. This corresponds to a window size of 2920 Bytes since each data segment comprises 1460 user data. Figure 4-14 depicts the timing diagram of such a unidirectional transmission and Equation 4-5 gives the obtained theoretical maximum throughput (user data).

$$\text{Throughput} = \frac{2 \times 1460 \text{ Bytes}}{2 \times 1538 + 84 \text{ Bytes}} \times 100 \text{ Mbits/s} = 92.4 \text{ Mbits/s} \qquad \textbf{(4-5)}$$

Field	–DATA– segment #Bytes	–ACK– segment #Bytes
Fast Ethernet preamble	8	8
Fast Ethernet destination address	6	6
Fast Ethernet source address	6	6
Ethernet type field	2	2
IP header	20	20
TCP header	20	20
User data	1460	0
Pad to Fast Ethernet minimum	0	6
Fast Ethernet CRC	4	4
Interpacket gap on Fast Ethernet: 916 ns	12	12
Total	1538	84

Table 4-1. Field sizes for Fast Ethernet theoretical maximum throughput calculation

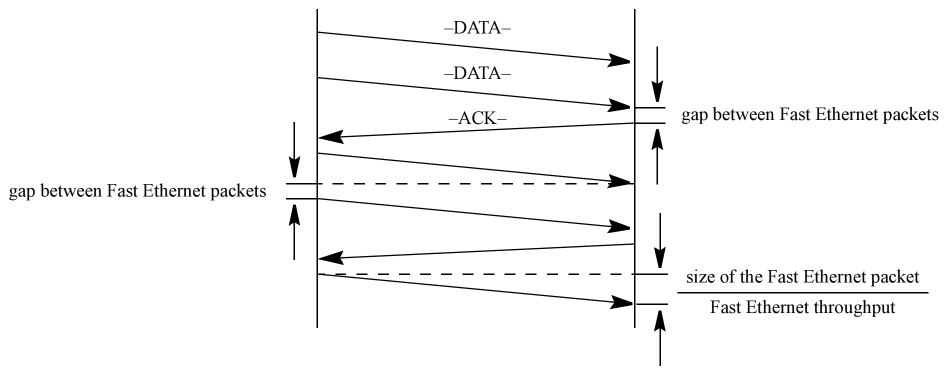

–DATA–
–DATA–
–ACK–
gap between Fast Ethernet packets
gap between Fast Ethernet packets
size of the Fast Ethernet packet
Fast Ethernet throughput

Figure 4-14. Timing diagram where the sender transmits two full-sized data segments and then the receiver sends an acknowledgment for these two data segments

If the TCP window is opened to its maximum size (65535), this enables the sender to transmit 44 full-sized data segments each of 1460 bytes. If the receiver sends an –ACK– every 22nd segment, then the obtained theoretical maximum throughput is given by Equation 4-6.

$$\text{Throughput} = \frac{22 \times 1460 \text{ Bytes}}{22 \times 1538 + 84 \text{ Bytes}} \times 100 \text{ Mbits/s} = 94.7 \text{ Mbits/s} \qquad \textbf{(4-6)}$$

This is the theoretical limit, and makes certain assumptions: an –ACK– sent by the receiver doesn't collide on the Ethernet with one of the sender's –DATA– segment, the sender can transmit two segments with the minimum Fast Ethernet spacing, and the receiver can generate the –ACK– within the minimum Fast Ethernet spacing.

The bottom line in all these numbers is that the real upper limit on how fast TCP can run is determined by the size of the TCP window and the speed of light, i.e. network throughput. As concluded by [Partridge93], many protocol performance problems are implementation deficiencies rather that inherent protocol limits.

4.5.2 The multi-PC environment

All the experiments described in this section have been conducted on two PC's running the Microsoft Windows NT 4.0 service pack 3 operating system and interconnected by a switched Fast Ethernet network (Figure 4-15). Each PC is a Bi-Pentium Pro 200MHz with the Intel PR440FX ATX motherboard comprising 64 MBytes of 60ns EDO RAM and an Intel EtherExpress Pro/100B PCI network card adapter.

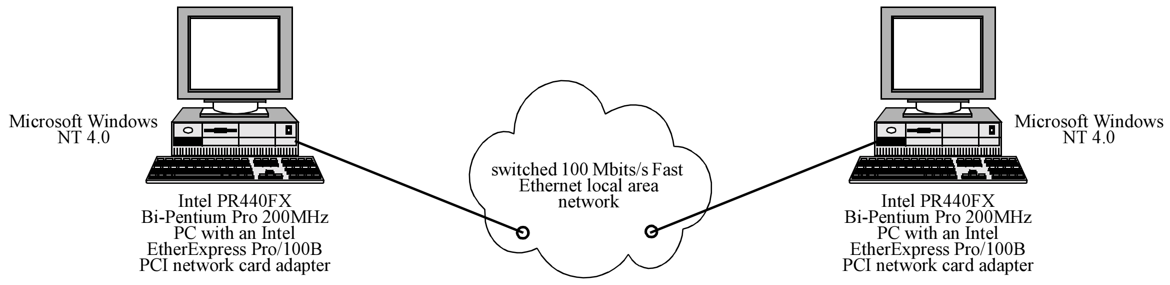

Microsoft Windows NT 4.0
Intel PR440FX Bi-Pentium Pro 200MHz PC with an Intel EtherExpress Pro/100B PCI network card adapter
switched 100 Mbits/s Fast Ethernet local area network
Microsoft Windows NT 4.0
Intel PR440FX Bi-Pentium Pro 200MHz PC with an Intel EtherExpress Pro/100B PCI network card adapter

Figure 4-15. The multi-PC environment comprises two Intel PR440FX Bi-Pentium Pro 200MHz with an Intel EtherExpress Pro/100B PCI network card adapter PC's interconnected through a switched 100 Mbits/s Fast Ethernet local area network

Experiences (not presented in this section for the sake of simplicity), have shown that the Nagle algorithm (Section 4.3.2) introduces a predominant latency of 220 ms when transmitting a single token from one PC to another, i.e. unpipelined token transfer (Section 4.5.3). Moreover, for optimal performance, the transmit buffer size and the receive buffer size should be between 8 KBytes (which is the default value for a socket) and 16 KBytes (Section 4.3.2). Poor performance has been observed when decreasing these two sizes.

Therefore, the results obtained in this section have been measured by disabling the Nagle algorithm and setting the transmit and receive buffer size to 8 KBytes.

4.5.3 Performance evaluation of the token-oriented message-passing system in terms of latency for an unpipelined token transfer

Communication networks exhibit a linear behaviour. The time for a packet to be transferred from one PC to another using a given communication library, e.g. the Windows Sockets library, and a given protocol stack, e.g. the TCP/IP protocol stack, depends linearly on the size of the packet. Therefore two parameters, i.e. *latency* and *throughput*, are sufficient to model the linear behaviour of a communication mechanism using Equation 4-7.

$$\text{TransferTime} = \text{Latency} + \frac{\text{PacketSize}}{\text{Throughput}} \tag{4-7}$$

As mentioned by Hennessy and Patterson [Hennessy96, pp. 641], communication latency is crucial since it affects both performance and how easy it is to program a multiprocessor. Unless latency is hidden, it directly affects performance either by tying up processor resources or by causing the processor to wait. With the CAP computer aided parallelization tool (Chapter 3), communication latencies and even communications are hidden by overlapping communications with computations, thanks to the inherent pipelined execution with the two split-merge parallel while (Section 3.8.6) and indexed parallel (Section 3.8.7) CAP constructs and the two asynchronous *SendToken* and *ReceiveToken* routines (Section 4.4).

To evaluate latency and throughput of a given communication mechanism, we plot the time for a packet or a token to be transferred from one PC to another as a function of the data set size, and linearize using a least-square fit approximation. The slope of the linearized curve gives the throughput. The intersection with the vertical axis (zero size packet) gives a measure of the latency.

To evaluate the time for a packet to be transferred from one PC to another, we measure its round-trip time and divide it by 2. In order to quantify the cost of the 3 mechanisms (Section 4.4) that coalesce tokens, serialize the coalesced tokens into socket packets and transfer the socket packets, two different experiments are conducted.

The first experiment consists of transferring a packet back and forth between two PC's for increasing sizes using the asynchronous Windows Sockets 2.0 *WSASend* and *WSARecv* routines without any programming overheads. In order to mimic a single token transmission (Figure 4-16), the socket packet contains one *WSABUF* buffer of increasing size. The results of this *packet-level* experiment evaluates, in terms of latency and throughput, the performance of the TCP/IP protocol stack and the asynchronous Windows Sockets interface giving an upper limit.

```
 1 struct PacketT {
 2   SOCKET Socket;
 3   WSABUF* CurrentP; int Index;
 4   WSABUF* BufferP; int BufferCount;
 5   void Reset() {
 6     for(int i = 0; i < BufferCount; i++) {
 7       CurrentP[i].Len = BufferP[i].Len;
 8       CurrentP[i].BufP = BufferP[i].BufP;
 9     } // end for
10     Index = 0;
11   } // end Reset
12 }; // end struct PacketT
13
14 SemaphoreT SEMAPHORE(0);
15
16 void WriteCompletion(int bWritten, PacketT* packetP) {
17   SEMAPHORE.Signal(); // Wake up main thread
18 } // end WriteCompletion
19 void ReadCompletion(int bRead, PacketT* packetP) {
20   while(packetP->Index < packetP->BufferCount &&
21         bRead >= packetP->CurrentP[packetP->Index].Len) {
22     bRead -= packetP->CurrentP[packetP->Index].Len
23     packetP->Index++;
24   } // end while
25   if(packetP->Index == packetP->BufferCount) {
26     SEMAPHORE.Signal(); return; // Wake up main thread
27   } // end if
28   packetP->CurrentP[packetP->Index].Len -= bRead;
29   packetP->CurrentP[packetP->Index].BufP += bRead;
30
31   WSARecv(packetP->Socket, // Initiate new read request
32           packetP->CurrentP + packetP->Index,
33           packetP->BufferCount - packetP->Index,
34           packetP,
35           ::ReadCompletion);
36 } // end ReadCompletion
```

Program 4-3. Read and write completion routines used in conjunction with the overlapped Windows Sockets 2.0 *WSASend* and *WSARecv* routines

Program 4-3 shows the read and write completion routines used in Program 4-4. Since the write completion occurs only when all the data that are in the packet are sent (Section 4.3.3), the *WriteCompletion* routine (lines 16-18) awakes the thread who initiates the I/O operation by signaling a semaphore (line 17). On the other hand, the read completion occurs regardless of whether or not the incoming data fills all the *WSABUF* buffers contained in the packet (Section 4.3.3). Therefore the *ReadCompletion* routine (lines 19-36) must check that all the buffers are filled before signaling the thread who initiates the I/O operation (line 26). If all the buffers are not filled, then the callback routine initiates a new receive operation with an updated packet (lines 32-35). This permits data transmissions to occur entirely within a pre-emptive context.

Program 4-4 shows the master and slave main routines (lines 1-17 for the master and lines 18-34 for the slave) that iteratively exchange a packet back and forth between a PC and another PC.

```
 1 void Master(PacketT* packetP) {
 2   for(int i = 0; i < NTIMES; i++) {
 3     packetP->Reset();
 4     WSASend(packetP->Socket, // Initiate write request
 5             packetP->CurrentP, packetP->BufferCount,
 6             packetP,
 7             ::WriteCompletion);
 8     SEMAPHORE.Wait(); // Wait for network completion
 9
10     packetP->Reset();
11     WSARecv(packetP->Socket, // Initiate read request
12             packetP->CurrentP, packetP->BufferCount,
13             packetP,
14             ::ReadCompletion);
15     SEMAPHORE.Wait(); // Wait for network completion
16   } // end for
17 } // end Master
18 void Slave(PacketT* packetP) {
19   for(int i = 0; i < NTIMES; i++) {
20     packetP->Reset();
21     WSARecv(packetP->Socket, // Initiate read request
22             packetP->CurrentP, packetP->BufferCount,
23             packetP,
24             ::ReadCompletion);
25     SEMAPHORE.Wait(); // Wait for network completion
26
27     packetP->Reset();
28     WSASend(packetP->Socket, // Initiate write request
29             packetP->CurrentP, packetP->BufferCount,
30             packetP,
31             ::WriteCompletion);
32     SEMAPHORE.Wait(); // Wait for network completion
33   } // end for
34 } // end Slave
```

Program 4-4. Packet-level round-trip time measurement using the overlapped Windows Sockets 2.0 *WSASend* and *WSARecv* routines

The second experiment consists of transferring a CAP token back and forth between two PC's for increasing sizes using a real CAP program (Programs 4-5 and 4-6) with the token-oriented message-passing system (Section 4.4). Remember that the asynchronous *SendToken* and *ReceiveToken* routines use the same asynchronous Windows Sockets 2.0 routines used in the packet-level experiment, but includes three additional mechanisms that coalesce tokens, serialize the coalesced tokens into packets and transfer packets through a stream-oriented connection. Therefore the results of this *CAP token-level* experiment evaluates, in terms of latency and throughput, how well does the token-oriented message-passing system and its underlying mechanisms compare with the packet-level measurements.

Program 4-5 declares a *TokenT* token (lines 70-72) comprising a single user buffer of any size (lines 36-37). Serialization routines (Section 4.4) are also shown, i.e. the pack routine (lines 42-48), the unpack routine (lines 50-57), the address restore routine (lines 59-64) and the transfer address list size routine (lines 66-68). In order to make a fair comparison of the CAP token-level measurements with the packet-level measurements, the socket packet corresponding to the serialization of a single *TokenT* token must be taken into account when plotting the token transfer time as a function of the data set size, i.e. socket packet size. Figure 4-16 depicts the socket packet that is transferred across the network by the token-oriented message-passing system. The CAP's runtime system adds a 28-byte header containing all the information to execute a parallel operation and an 8-byte trailer

```
struct BufferT {
  void* BufferP;
  int BufferSize;

  void Allocate(int bufferSize);
}; // end class BufferT

void capAddressPack(int& listSize,
                    WSABUF* &bufferP,
                    BufferT* udP) {
  bufferP->Len = udP->BufferSize;
  bufferP->BufP = udP->BufferP;
  bufferP++; // Points to next memory block
} // end capAddressPack

void capAddressUnpack(int& listSize,
                      WSABUF* &bufferP,
                      BufferT* udP) {
  udP->Allocate(bufferP->Len);
  bufferP->BufP = udP->BufferP;
  thrAddressSave(listSize, bufferP, &(udP->BufferP));
  bufferP++; // Points to next memory block
} // end capAddressUnpack

void capAddressRestore(int& listSize,
                       WSABUF* &bufferP,
                       BufferT* udP) {
  thrAddressRestore(listSize, bufferP);
  bufferP++; // Points to next memory block
} // end capAddressRestore

int capAddressListSize(BufferT* udP) {
  return 1; // Size of list
} // end capAddressListSize

token TokenT {
  BufferT Buffer;
}; // end token TokenT
```

Program 4-5. Declaration of a token with its serialization routines

containing some constants needed for certain parallel CAP constructs. The message passing system adds a 40-byte *WSABUF* buffer to transfer the packet through a stream-oriented connection (TCP/IP). Therefore, the size of the transferred socket packet equals 76 Bytes plus the user's buffer size.

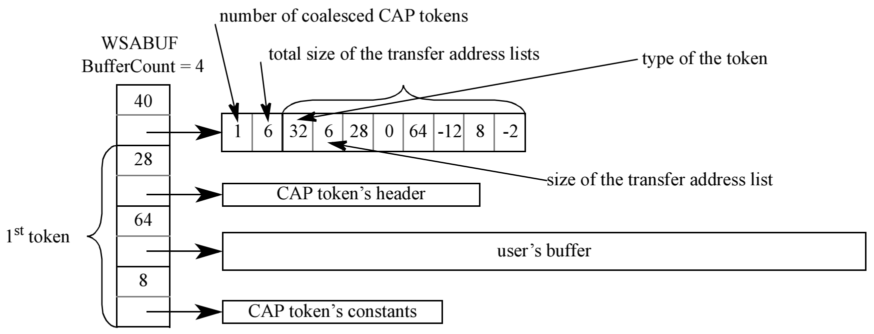


Figure 4-16. Socket packet after having coalesced 1 *TokenT* token and serialized it

Program 4-6 shows the CAP specification for measuring the token round-trip time. It is based on the *for* CAP construct (line 9) that iteratively redirects a token from the master (line 13) to the slave (line 11) and back to the master (line 13).

```
 1 const int NTIMES = 10000;
 2
 3 operation ServerT::RoundTripTimeMeasurement
 4   in TokenT* InputP
 5   out TokenT* OutputP
 6 {
 7   Master.{ }
 8   >->
 9   for(int Index = 0; Index < NTIMES; Index++)
10   (
11     Slave.{ }
12     >->
13     Master.{ }
14   ); // end for
15 } // end operation ServerT::RoundTripTimeMeasurement
16
17 int main(...)
18 {
19   ...
20   start_time();
21   call Server.RoundTripTimeMeasurement in ...
22   stop_time();
23 ...
24 } // end main
```

Program 4-6. CAP token-level round-trip time measurement using a for CAP construct

Results of the packet-level and CAP token-level experiments (Figure 4-17) show that the communication mechanism, i.e. the token-oriented message-passing system, the Windows Sockets library, the TCP/IP stack protocol and the Fast Ethernet network, exhibits two zones with different linear behaviours: a first zone for small-sized packets ranging from 16 Bytes to 2 KBytes and a second zone for medium-sized packets ranging from 2 KBytes to 64 KBytes. For large-sized packets, i.e. above 64 KBytes, the communication interface becomes saturated and a degradation of performance is observed.

Figure 4-17 depicts the two linear behaviours of the packet-level (Programs 4-3 and 4-4) and CAP token-level (Programs 4-5 and 4-6) interfaces and their linearization using the least-square fit approximation. The horizontal axis represents the size of the socket packet in Bytes and the vertical axis represents the time to transfer such a packet from one PC to another using either the low-level Windows Sockets library (packet-level) or the CAP programming environment (CAP token-level).

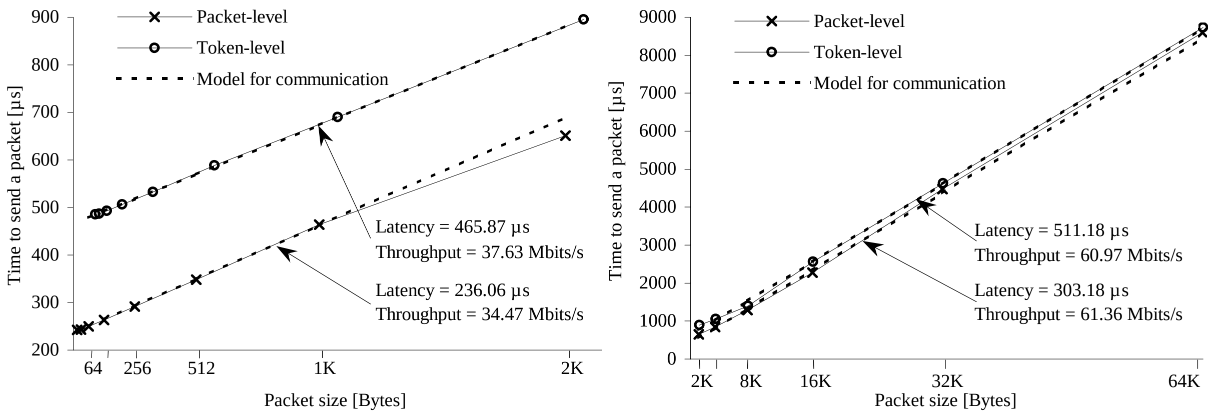


Figure 4-17. Packet-level and CAP token-level performances in terms of latency and throughput

Figure 4-18 shows the measured effective network throughputs (Equation 4-8) for the packet-level and CAP token-level experiments as a function of the transmitted packet size. The curve named “effective user throughput” gives the number of user’s data transferred per second where PacketSize = UserBufferSize + 76 Bytes (Figure 4-16, Equation 4-9). Dashed lines represent our linear model for communication using latencies and throughputs of Figure 4-17.

$$\text{EffectiveNetworkThroughput} = \frac{\text{PacketSize}}{\text{TransferTime}} = \frac{\text{PacketSize}}{\text{Latency} + \dfrac{\text{PacketSize}}{\text{Throughput}}} \tag{4-8}$$

$$\text{EffectiveUserThroughput} = \frac{\text{UserBufferSize}}{\text{TransferTime}} = \frac{\text{UserBufferSize}}{\left(\text{Latency} + \dfrac{76}{\text{Throughput}}\right) + \dfrac{\text{UserBufferSize}}{\text{Throughput}}} \tag{4-9}$$

Table 4-2 summarizes and compares latencies and throughputs of the Windows Sockets 2.0 application programming interface (packet-level) with the token-oriented message-passing system of the CAP parallel programming environment based on the Windows Sockets API.

As expected, the Windows Sockets application programming interface provides lower latencies than the token-oriented message-passing system for all socket packet sizes (Figure 4-18 and Table 4-2). This slight degradation of performance is due to the 3 mechanisms implemented in the message-passing system necessary for transmitting CAP tokens from one address space to another, i.e. coalescencing of tokens, the serialization into a packet (and deserialization) and the transfer of packets through a stream-oriented socket. The cost of these 3 mechanisms is an increase of less than 230 µs in latency, i.e. a 50% increase for small-sized packets and a 40% increase for medium-sized packets. Transmission throughputs are not affected by the message passing system for all configurations.

The cost of the 76 Bytes of data (Figure 4-16) that CAP runtime system and the message-passing system add to a user’s token (Program 4-5) is an increase of less than 16 µs (= 76 Bytes / 37.63 Mbits/s, Equation 4-9) in latency for small-sized tokens, i.e. a 3.3% increase. For medium- and large-sized tokens the cost is completely negligible.

From these two packet-level and CAP token-level experiments we can conclude that:

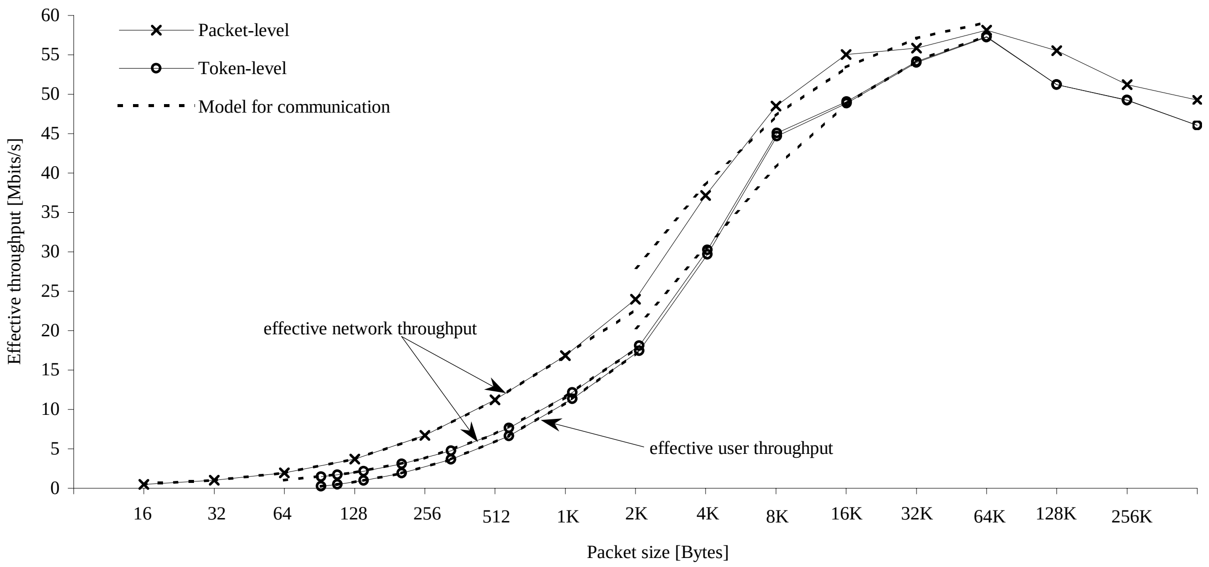


Figure 4-18. Effective throughputs of the packet-level and CAP token-level interfaces as a function of the socket packet size

Size of the socket packet	Packet-level interface	CAP token-level interface (function of the packet size)	CAP token-level interface (function of the user's buffer size)

Small: 16 Bytes - 2 KBytes	Latency = 236.06 µs
	Throughput = 34.47 Mbits/s
		Latency = 465.87 µs
		Throughput = 37.63 Mbits/s
			Latency = 481.28 µs
			Throughput = 37.63 Mbits/s

Medium: 2 KBytes - 32 KBytes	Latency = 303.18 µs
	Throughput = 61.36 Mbits/s
		Latency = 511.18 µs
		Throughput = 60.97 Mbits/s
			Latency = 511.18 µs
			Throughput = 60.97 Mbits/s

Large: > 32 KBytes	Not a linear behaviour, saturation	Not a linear behaviour, saturation	Not a linear behaviour, saturation

Table 4-2. Latencies and throughputs of the Windows Sockets API (packet-level) and the token-oriented message-passing system of the CAP parallel programming environment (CAP token-level)

- Due to the significant transmission latencies (between 200 and 500 µs), the effective network throughput strongly depends on the transmitted packet size. Up to a size of 64 KBytes, the effective network throughput increases from a few of Mbits/s up to 55 Mbits/s. From 64 KBytes, stream sockets operate poorly due to a saturation somewhere in the TCP/IP internal protocol buffers. These buffers influence the behaviour of the TCP sliding window protocol which regulates the data flow, thus degrading performance. Therefore, the token coalescence mechanism implemented in the token-oriented message-passing system attempts to transmit only medium-sized socket packets (approximately between 4 KBytes and 32 KBytes) in order to optimize the latency of the underlying transmission protocol.
- The cost of the token-oriented message-passing system has been evaluated in terms of latency and throughput. A reasonable increase of less than 230 µs in latency is observed while throughputs are not affected by the token transfer mechanisms. Since the *SendToken* and *ReceiveToken* routines are asynchronous, the CAP parallel programming tool, thanks to its inherent pipeline execution, is able to hide these high transmission latencies by overlapping communications with computations.
- The 76-byte data that CAP runtime system and the message passing system append to a user's token does not affect performance. The CAP runtime system automatically routes tokens without loss of performance.

4.5.4 Performance evaluation of the token-oriented message-passing system in terms of throughput and processor utilization for a pipelined token transfer

The previous section has demonstrated the efficiency of the message-passing system of the CAP tool mainly in terms of latency when transmitting a single token from one PC to another. This section evaluates the performances that programmers can expect when parallel CAP operations transfer not a single token but a bunch of tokens from one PC to another. This crucial pattern of communication, induced by the *parallel while* (Section

3.8.6) and *indexed parallel* (Section 3.8.7) CAP constructs, appears more frequently in CAP programs than a single token motion. The token-oriented message passing system and the underlying stream sockets are able to improve performance by pipelining network access requests thus reducing latencies, i.e. latency of the message-passing system, latency of the TCP/IP stack protocol and latency of the Fast Ethernet network.

The goal of this section is to evaluate the mechanism that coalesces tokens into a same socket packet and to measure the processor utilization when sending or receiving tokens so as to be able to quantify how well CAP's runtime system can overlap communications by computations.

The first experiment consists of transferring a bunch of packets from one PC to another for increasing sizes using the asynchronous Windows Sockets 2.0 *WSASend* and *WSARecv* routines without any programming overheads (Programs 4-3 and 4-7). In order to mimic the coalescence of 1, 5, 10 or 15 tokens performed by the token-oriented message-passing system, the transferred socket packets contain 1, 5, 10 or 15 *WSABUF* buffers per test. The results of this *packet-level* experiment characterizes, in terms of latency, throughput and processor utilization, the unidirectional pipelined performance of the Windows Sockets interface and the underlying TCP/IP stack protocol thus giving references to compare with the message-passing system measurements.

```
 1 void Sender(PacketT* packetP) {                          11 void Receiver(PacketT* packetP) {
 2   for(int i = 0; i < NTIMES; i++) {                     12   for(int i = 0; i < NTIMES; i++) {
 3     packetP->Reset();                                   13     packetP->Reset();
 4     WSASend(packetP->Socket, // Initiate write request  14     WSARecv(packetP->Socket, // Initiate read request
 5             packetP->CurrentP, packetP->BufferCount,    15             packetP->CurrentP, packetP->BufferCount,
 6             packetP,                                    16             packetP,
 7             ::WriteCompletion);                         17             ::ReadCompletion);
 8     SEMAPHORE.Wait(); // Wait for network completion    18     SEMAPHORE.Wait(); // Wait for network completion
 9   } // end for                                          19   } // end for
10 } // end Sender                                         20 } // end Receiver
```

Program 4-7. Unidirectional pipelined packet transfer measurement using the overlapped Windows Sockets 2.0 *WSASend* and *WSARecv* routines

Program 4-7 shows the sender and receiver main routines (lines 1-10 for the sender and lines 11-20 for the receiver) where the sender asynchronously sends *NTIMES* packets to the receiver located on a different PC. The *ReadCompletion* and *WriteCompletion* routines are shown in Program 4-3.

The second experiment consists of transferring a bunch of CAP tokens from one PC to another for increasing sizes using either a *parallel while* or an *indexed parallel* CAP construct (Program 4-8). To evaluate the token coalescence mechanism, several measurements are conducted with different coalescence factors (1, 5, 10 and 15). The results of this *CAP token-level* experiment characterizes, in terms of latency, throughput and processor utilization, the unidirectional pipelined performance of the token-oriented message-passing system and are compared with the above packet-level experiment so as to evaluate the relative performance of the message-passing system.

Program 4-8 shows the CAP specification for measuring the unidirectional pipelined token transfer time. It is based on the indexed parallel CAP construct (lines 24-30) that sends *NTIMES* tokens from the *Sender*'s PC to the *Receiver*'s PC.

Program 4-8 does not use the high-level flow-control mechanism provided by the CAP tool (Section 3.10), since flow-control requires additional communication between the *Receiver* and the *Sender* interfering with the performance measurements. A global semaphore, not shown for the sake of simplicity, regulates the number of split tokens: in the user's buffer allocation (line 9) the semaphore is decremented and in the *TokenT* token's destructor the semaphore is incremented, i.e. when the token is sent. Therefore, the semaphore's initial value corresponds to the number of tokens that are in the message-passing system output queue (Figure 3-5).

As in Section 4.5.3, to make a fair comparison of CAP token-level measurements with packet-level measurements, the size of the packets transmitted by the message-passing system must be taken into account when plotting the token transfer time as a function of the data set size. Figure 4-19 depicts the transferred socket packets after coalescing 3 *TokenT* tokens (Programs 4-5 and 4-8).

The size of a transferred socket packet is given by Equation 4-10 and its number of scattered buffers is given by Equation 4-11. The coalescence factor is the number of tokens that the message-passing system coalesces into a single packet.

```
const int NTIMES = 10000;
const int BUFFER_SIZE = 64;

void GenerateToken(TokenT* inputP,
                   TokenT* &subtokenP,
                   int index)
{
  subtokenP = new TokenT;
  subtokenP->Buffer.Allocate(BUFFER_SIZE);
} // end GenerateToken

void MergeToken(TokenT* outputP,
                TokenT* subtokenP,
                int index)
{
} // end MergeToken

operation ServerT::UnidirectionalPipelinedMeasurement
  in TokenT* InputP
  out TokenT* OutputP
{
  Master.{ } // Tokens generated by Master
  >->
  indexed
    (int Index = 0; Index < NTIMES; Index++)
  parallel
    (GenerateToken, MergeToken, Slave, TokenT Out1())
   (
    Slave.{ } // Tokens received by Slave
  ); // end indexed parallel
} // end operation ServerT::UnidirectionalPipelinedMeasurement

int main(...)
{
  ...
  start_time();
  call Server.UnidirectionalPipelinedMeasurement in ...
  stop_time();
...
} // end main
```

Program 4-8. Unidirectional pipelined token transfer measurement using an *indexed parallel* CAP construct

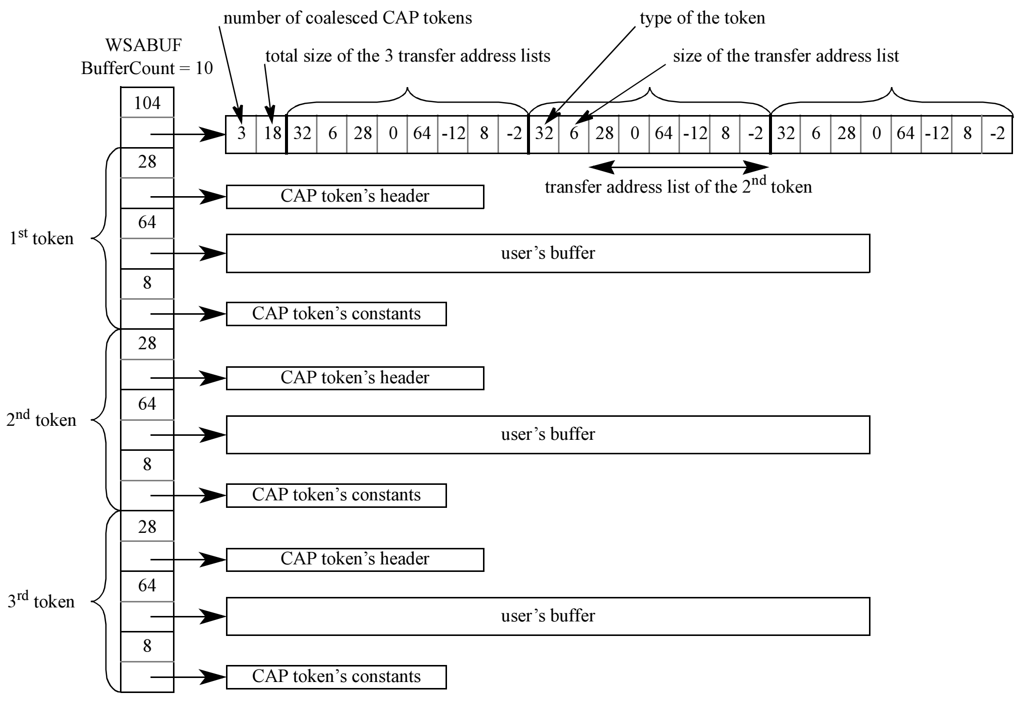


Figure 4-19. Socket packet after having coalesced 3 *TokenT* tokens and serialized it

$$\text{PacketSize} = 8\ \text{Bytes} + \text{CoalescenceFactor} \cdot (68\ \text{Bytes} + \text{UserBufferSize}) \tag{4-10}$$

$$\text{NumberOfScatteredBuffers} = 1 + \text{CoalescenceFactor} \cdot 3 \tag{4-11}$$

Figure 4-20 compares the effective network throughputs (Equation 4-8) for a unidirectional pipelined data transfer obtained with the overlapped Windows Sockets 2.0 programming interface (Program 4-7) and the token-oriented message-passing system (Program 4-8).

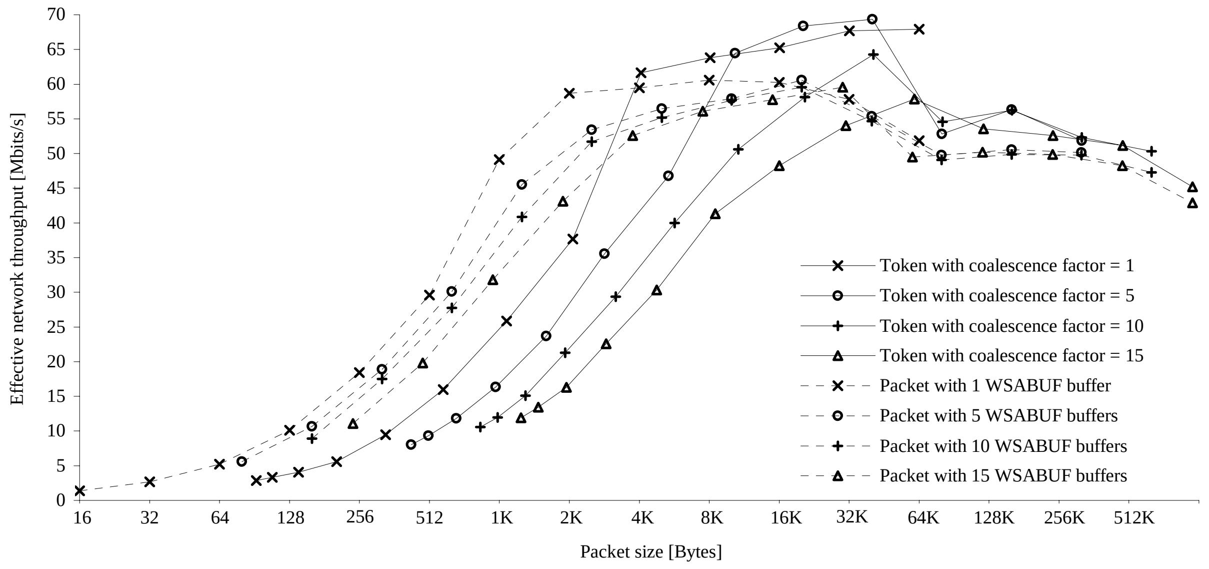


Figure 4-20. Comparing the effective network throughputs of the token-oriented message-passing system with the overlapped Windows Sockets 2.0 application programming interface for a unidirectional pipelined data transfer

Table 4-3 summarizes, after having linearized the 4 curves of Figure 4-20, latencies and throughputs of the overlapped Windows Sockets 2.0 application programming interface. Two different linear behaviours are observed. For small-sized packets, latency is predominant, while for medium-sized packets, latency is nearly negligible and the effective network throughput is approximately independent of the packet size. For large-sized packets, the communication interface becomes saturated and performance suffers due to the TCP sliding window protocol regulating the data flow. This effect is also visible in Figure 4-18 when sending one packet after the other without pipelining.

Packet size [Bytes]	Packet with 1 *WSABUF* buffer	Packet with 5 *WSABUF* buffers	Packet with 10 *WSABUF* buffers	Packet with 15 *WSABUF* buffers

Small: 16 – 1K	Latency = 87.08 µs
	Throughput = 89.87 Mbits/s
		Latency = 101.76 µs
		Throughput = 85.69 Mbits/s
			Latency = 115.49 µs
			Throughput = 79.81 Mbits/s
				Latency = 142.85 µs
				Throughput = 84.13 Mbits/s

Medium: 1KB – 32K	Latency = 16.25 µs
	Throughput = 60.91 Mbits/s
		Latency = 61.63 µs
		Throughput = 61.81 Mbits/s
			Latency = 74.12 µs
			Throughput = 61.27 Mbits/s
				Latency = 91.38 µs
				Throughput = 60.91 Mbits/s

Large: > 32K	Not a linear behaviour, saturation	Not a linear behaviour, saturation	Not a linear behaviour, saturation	Not a linear behaviour, saturation

Table 4-3. Latencies and throughputs of the overlapped Windows Sockets 2.0 API for a unidirectional pipelined packet transfer and for various numbers of scattered buffers

Since for coalescing tokens, the message-passing system increases the number of scattered buffers per packet, it is essential to evaluate how the number of *WSABUF* buffers affects performance. For both small- and medium-sized packets, increasing the number of scattered buffers only increases latency. Since for medium-sized packets, the latency factor is rather negligible compared with the throughput factor, the cost of coalescing tokens into several scattered buffers reduces as the packet size increases (Figure 4-20).

Fortunately, when the overlapped Windows Sockets interface is faced with a series of packet sending requests (Figure 4-20, Table 4-3), the operating system is able to pipeline the transfer requests and to optimize the network utilization thus reducing latency: for small-sized packets, latency is reduced by 63% and for medium-sized packets, latency is reduced by 95% when comparing with the unpipelined packet transfer (Table 4-2). Concerning throughput, the Sockets library is able to gather several small-sized packets within a same TCP segment thus increasing throughput up to 90 Mbits/s, while for large-sized packets, throughput is unchanged at 60 Mbits/s.

Table 4-4 summarizes, after having linearized the 4 curves of Figure 4-20, latencies and throughputs of the CAP's message-passing system. Surprisingly, it offers better throughput for medium-sized packets than the Windows Sockets experiment (70 Mbits/s in Table 4-4 instead of 60 Mbits/s in Table 4-3). This might be due to the numerous small-sized *WSABUF* buffers appearing in the socket packets (Figure 4-19) which slow down the sender thus enabling the receiver to unpack the tokens and post a new socket read request (Section 4.4.1) before the TCP sliding window protocol stops the sender.

Packet size [Bytes]	Coalescence factor 1	Coalescence factor 5	Coalescence factor 10	Coalescence factor 15
Small: 16 – 2K	Latency = 233.48 µs Throughput = 84.60 Mbits/s	Latency = 372.68 µs Throughput = 86.75 Mbits/s	Latency = 540.89 µs Throughput = 83.04 Mbits/s	Latency = 691.00 µs Throughput = 71.06 Mbits/s
Medium: 2K - 32K	Latency = 70.74 µs Throughput = 68.79 Mbits/s	Latency = 105.08 µs Throughput = 71.09 Mbits/s	Latency = 488.48 µs Throughput = 71.21 Mbits/s	Latency = 585.90 µs Throughput = 62.24 Mbits/s
Large: > 32K	Not a linear behaviour, saturation	Not a linear behaviour, saturation	Not a linear behaviour, saturation	Not a linear behaviour, saturation

Table 4-4. Latencies and throughput of the CAP's message-passing system for a unidirectional pipelined token transfer with various coalescence factors

As in the unpipelined data transfer experiments (Section 4.5.3), the cost of a token-oriented message-passing system is essentially an increase of latency (Tables 4-4 and 4-3). However, as the coalescence factor increases, latency per token decreases (Table 4-5). This beneficial effect is exploited in the message-passing system to increase the effective network and user throughput. For example when transmitting 1KB tokens, with a coalescence factor of 1, an effective network throughput of 25 Mbits/s is reached, while with a coalescence factor of 15, an effective network throughput of 47 Mbits/s is obtained, i.e. an 88% increase (Figure 4-20). By adapting the coalescence factor to the token sizes, i.e. to transmit only packets whose sizes are within the optimal range 16KB–32 KB, it is possible to maximize the effective user throughput.

Packet size [Bytes]	Coalescence factor 1	Coalescence factor 5	Coalescence factor 10	Coalescence factor 15
Small: 16 – 2K	Latency = 233.48 µs/token	Latency = 74.54 µs/token	Latency = 54.09 µs/token	Latency = 46.07 µs/token
Medium: 2K - 32K	Latency = 68.79 µs/token	Latency = 21.02 µs/token	Latency = 48.85 µs/token	Latency = 39.06 µs/token

Table 4-5. By increasing the coalescence factor, the latency per token is decreased

Since the CAP computer aided-parallelization tool is able to overlap communications by computations (Section 3.3), it is essential to evaluate how much processor resource is consumed for sending and receiving tokens and how CAP message-passing system can decrease processor utilization.

Figures 4-21 and 4-22 show the processor utilization for the sender and the receiver during a unidirectional pipelined packet transfer using the overlapped Windows Sockets 2.0 application programming interface (Program 4-7). Three different curves are depicted. The first one represents the user test program activities, i.e. the percentage of elapsed time that the processor spends in user mode executing user code. The second one represents the network and system activities, i.e. the percentage of time that the processor spends in privileged mode executing system code or handling network interrupts. The third one represents all the activities, i.e. the sum of the user's activities and the system's activities (system code, network interrupts). Since Bi-Pentium Pro PC's are used for the experiments (Section 4.5.2), all processor utilization percentages indicate the utilization of the 2 processors, i.e. a processor utilization of 50% corresponds to a processor 100% busy and a processor 0% busy.

From Figures 4-21 and 4-22, we remark that:

- The sender and receiver processor utilizations are quite independent of the number of scattered buffers meaning that the overhead of the gather send and scatter receive interfaces are negligible.
- For small-sized packets, the sender processor utilization is around 55% – 60%. For medium-sized packets, the sender processor utilization drops as low as 25% – 30% for an optimal packet size of 8K – 32K bytes. This also corresponds to the zone where the effective network throughput is maximum (Figure 4-20, 60 Mbits/s). Therefore, the sender processor utilization is not proportional to the output socket packet rate, but rather depends on the TCP/IP protocol efficiency, i.e. how well the TCP segments are filled with incoming user's data (internal pipeline) and how well the TCP sliding window protocol works.

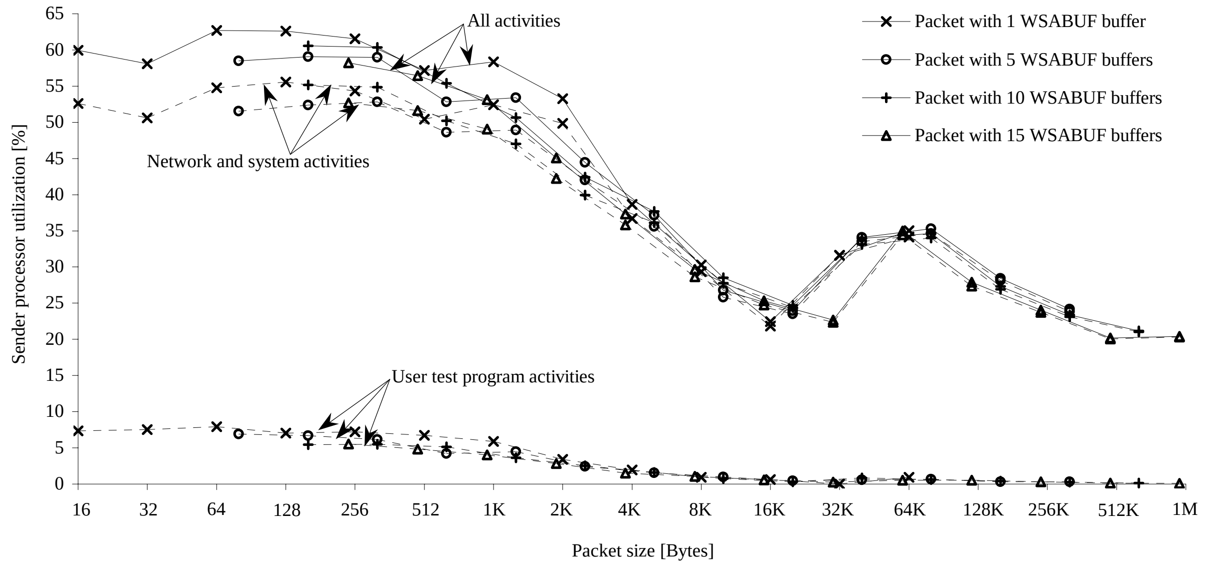


Figure 4-21. Processor utilization when sending various-sized packets across the network using the overlapped Windows Sockets 2.0 network interface

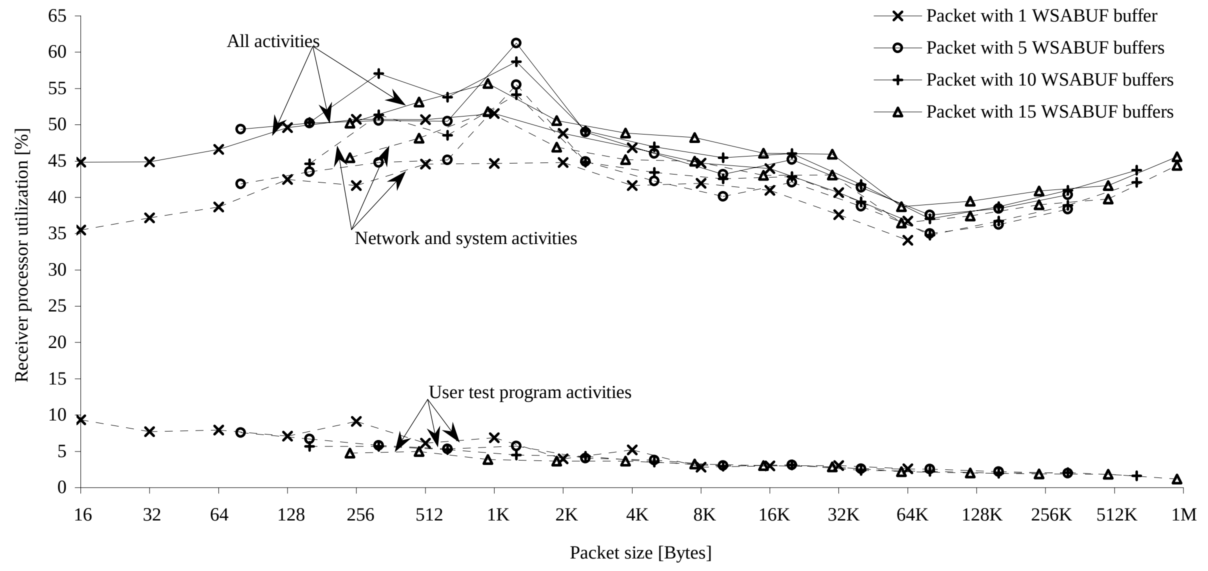


Figure 4-22. Processor utilization when receiving various-sized packets from the network using the overlapped Windows Sockets 2.0 network interface

- The receiver processor utilization tends to decrease as the packet size and the effective network throughput increase. For small-sized packets, the receiver processor utilization is around 50% – 60% and for medium-sized packets, it drops below 45%.
- Receiving packets consumes more processing power than sending packets.
- The TCP/IP stack protocol consumes much processing power. Particularly at the receiving site where approximately one processor is fully busy handling network interrupts.

Figures 4-23 and 4-24 show the processor utilization for the sender and the receiver during a unidirectional pipelined token transfer using CAP message-passing system (Program 4-8). Three different curves are depicted. The first one represents the test CAP program activities, i.e. the percentage of elapsed time that the processor spends executing Program 4-8. The second one represents the message-passing system thread, the network and the system activities, i.e. the percentage of time that the processor spends in user mode executing the message-passing system code, and in privileged mode executing system code and handling network interrupts. The third one represents all the activities, i.e. the sum of the test CAP program activities (first curve) and the message-passing system activities (second curve).

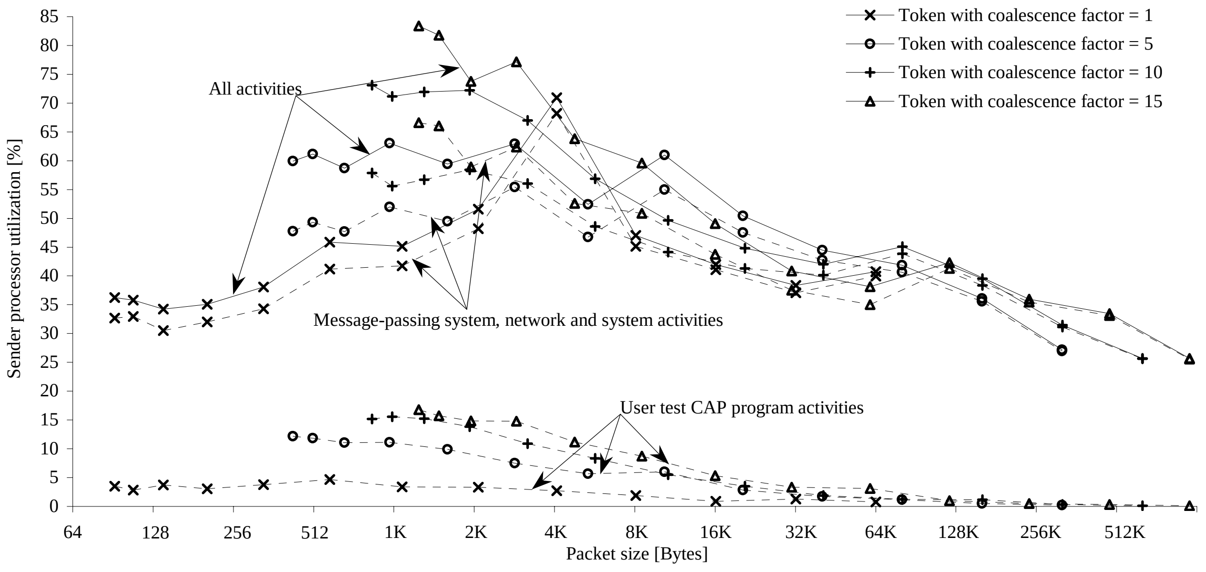


Figure 4-23. Processor utilization when sending various-sized tokens across the network using the CAP's message-passing system

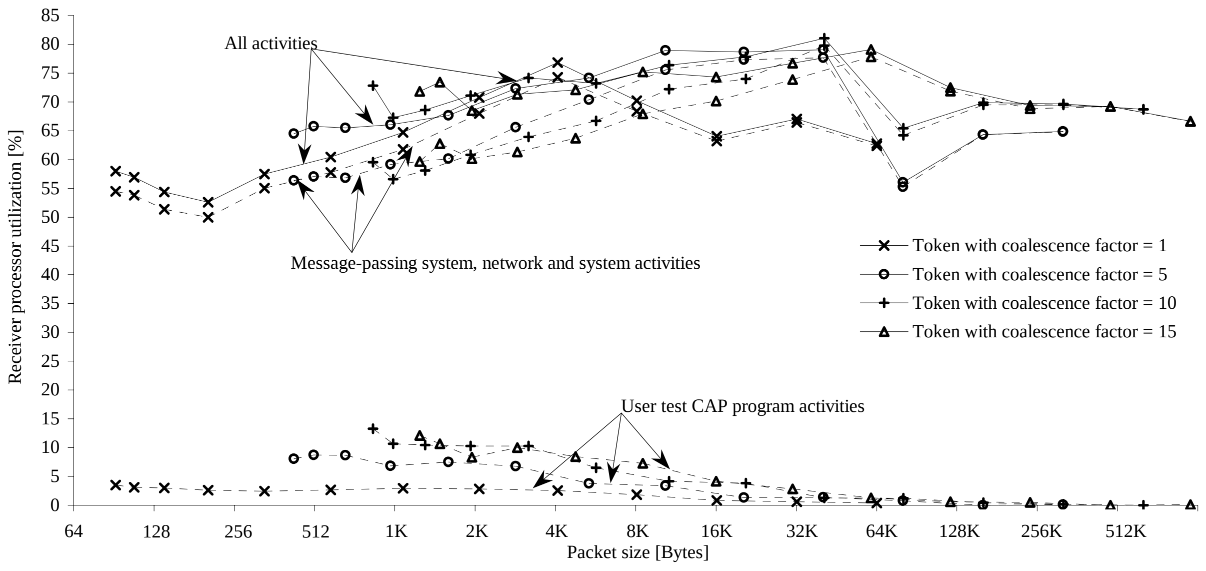


Figure 4-24. Processor utilization when receiving various-sized tokens from the network using the CAP's message-passing system

From Figures 4-23 and 4-24, we remark that:

- For medium-sized packets, the sender processor utilization is quite independent of the coalescence factor and is around 40% – 60%. Moreover, as the packet size increases the processor utilization decreases. Then, coalescing tokens not only increases the effective network throughput but also decreases the sender processor utilization. Note that increasing the effective network throughput, decreases the time during which the processor sends tokens (for a same amount of data to be transferred), therefore, also decreases the effective processor utilization.

  In order to illustrate this effect, let us take an example where a CAP program sends every second 2048 tokens of 2KBytes each, i.e. 32 Mbits/s. With a coalescence factor of 1, the sender processor utilization is 50% (Figure 4-23, packet size = 2KBytes) and the effective network throughput is 40 Mbits/s (Figure 4-20). Then, the transmission time is 2048[tokens] x 2[KBytes] / 40[Mbits/s] = 800 ms during which the processor is 50% busy. Therefore, over a period of 1 second, the mean effective sender processor utilization is 40%. With a coalescence factor of 16, the sender processor utilization is 45% (Figure 4-23, packet size = 32KBytes) and the effective network throughput is 55 Mbits/s (Figure 4-20). Then, the transmission time is 2048[tokens] x

2[KBytes] / 55[Mbits/s] = 582 ms during which the processor is 45% busy. Therefore, over a period of 1 second, the mean effective sender processor utilization is 26% (compared to 40% with a coalescence factor of 1).

- The token-oriented message-passing system introduces a maximum increase of 30% on the sender processor utilization compared with the overlapped Windows Sockets application programming interface.
- For medium sized-packets, the receiver processor utilization is around 65% – 80%. Despite the fact that the instantaneous receiver processor utilization slightly increases as the packet size and the effective network throughput increase, it is still worth coalescing tokens to reduce the receiver processor utilization.
  In order to illustrate this effect, let us take an example where a CAP program receives every second 2048 tokens of 2KBytes each, i.e. 32 Mbits/s. With a coalescence factor of 1, the receiver processor utilization is 70% (Figure 4-24, packet size = 2KBytes) and the effective network throughput is 40 Mbits/s (Figure 4-20). Then, the transmission time is 2048[tokens] x 2[KBytes] / 40[Mbits/s] = 800 ms during which the processor is 70% busy. Therefore, over a period of 1 second, the mean effective receiver processor utilization is 56%. With a coalescence factor of 16, the receiver processor utilization is 75% (Figure 4-24, packet size = 32KBytes) and the effective network throughput is 55 Mbits/s (Figure 4-20). Then, the transmission time is 2048[tokens] x 2[KBytes] / 55[Mbits/s] = 582 ms during which the processor is 75% busy. Therefore, over a period of 1 second, the mean effective receiver processor utilization is 44% (compared to 56% with a coalescence factor of 1).
- The token-oriented message-passing system introduces a maximum increase of 40% on the receiver processor utilization compared with the overlapped Windows Sockets application programming interface.

4.6 Summary

Section 4.2 has demonstrated that it is possible to access in parallel 15 disks hooked onto the same PC (with an efficiency of 89%) using the Windows NT file system without any loss of performance, i.e. compared with the experiment where the 15 disks are accessed using a low-level SCSI-2 block interface. Moreover, reading 23.51 MBytes/s of data from 15 disks with the NT file system does not consume much computing power (less than 14%) thus enabling PS^2 applications to overlap disk accesses with computations. These results also show that accessing disks through a low-level SCSI-2 block interface and implementing a custom single-disk file system will not improve performances. It is worth implementing the PS^2 customizable parallel file system directly on top of the native Windows NT file system for several reasons: firstly it greatly simplifies the development of PS^2, secondly it increases the ease of portability, thirdly it yields to a more reliable[1] parallel file system and fourthly it keeps the compatibility with other Windows applications (e.g. the Explorer application).

Section 4.3 has shown how MPS asynchronously transmits a packet, i.e. a serialized CAP token, through a TCP/IP connection using the Microsoft Windows Sockets 2.0 API. Section 4.4 has presented the two mechanisms used in MPS for serializing CAP tokens, i.e. the copy-pack and the address-pack serialization mechanisms. Since the copy-pack mechanism requires the use of a temporary buffer involving additional memory-to-memory copies, it is only used in heterogeneous environments where the transmitted data format has to be adapted to the destination machine (big/little endian, 32/64 bits). The address-pack mechanism, only used in homogeneous environments (e.g. in a multi-PC environment), enables CAP tokens to be serialized and transferred with no superfluous memory-to-memory copy providing optimal performance.

Section 4.5.3 has compared the performances of MPS with the raw performances of the asynchronous Windows Sockets 2.0 network interface for an unpipelined data transfer, i.e. when transmitting a single token at a time. Results of this experiment have shown that MPS increases the transfer latency by less than 230 μs, corresponding to a 40%-50% increase depending on the size of the transmitted packet. This growth in latency is due to the serialization mechanism and to the 3-step transfer mechanism slightly perturbing the underlying TCP/IP data transfer. On the other hand, MPS provides the same throughput (up to 61 Mbits/s on a Fast Ethernet network) as the raw Windows Sockets interface. This demonstrates that MPS does not add a cost proportional to the transferred CAP token but only a certain latency.

Section 4.5.4 has compared the performance of MPS with the raw performances of the asynchronous Windows Sockets 2.0 network interface for a pipelined data transfer, i.e. when transmitting numerous CAP tokens from one address space to another (induced by the *indexed parallel* or the *parallel while* CAP constructs). Results of this experiment have shown the benefits of the coalescence mechanism implemented in MPS coalescing several CAP

1. Reliable not in case of a disk crash but in case of an application crash or a PC crash. The NTFS file system incorporates data recovery mechanisms usable in case of a sudden PC crash.

tokens into a same socket packet. For example, when transmitting 1KB tokens, an 88% effective network throughput increase (25 Mbits/s with a coalescence factor of 1 to 47 Mbits/s with a coalescence factor of 15) has been obtained when coalescing 15 tokens into a same packet. Moreover, when faced with a pipelined token transfer, MPS is able to sustain an effective throughput of up to 70 Mbits/s over a Fast Ethernet network. As expected the TCP/IP stack protocol consumes much processing power making it difficult to overlap communications by computations, unless we use multi-processors PC's. For example on Bi-Pentium Pro 200MHz PC's, when transmitting CAP tokens at 70 Mbits/s, the sender processor utilization reaches 50%, i.e. one processor is totally devoted to communication, and the receiver processor utilization reaches 75%, i.e. more than one processor is devoted to communication.

Chapter 5

Design and Implementation of PS^2

5.1 Introduction

This chapter describes our extensible framework for developing parallel I/O- and compute- intensive applications on distributed memory commodity components. This framework consists of a CAP process hierarchy called the Parallel Storage-and-Processing Server (PS^2). The PS^2 process hierarchy comprises a fixed set of low-level parallel file system components as well as an extensible parallel processing system. The reusable low-level parallel file system components offered by a set of I/O threads enables files that are declustered across multiple disks to be accessed. The extensible parallel processing system comprises a set of compute threads that programmers can freely customized by incorporating application-specific or library-specific processing operations. The I/O threads and the compute threads are distributed among the distributed memory PC's. Each contributing PC comprises at least one I/O thread performing parallel disk accesses on locally hooked disks and at least one compute thread running application-specific or library-specific processing operations. Application or library programmers can, thanks to the CAP formalism, easily and elegantly extend the functionalities of PS^2 by combining the predefined low-level parallel file access components with the application-specific or library-specific processing operations in order to yield efficient pipelined parallel I/O- and compute- intensive CAP operations. These parallel CAP operations may be incorporated into C/C++ high-level libraries offering specific abstractions of parallel files and appropriate processing routines.

5.2 Trends in parallel storage systems

Most parallel file systems are based on a Unix-like interface, i.e. a file is seen as an addressable linear sequence of bytes (or records). Examples include the Bridge file system [Dibble88, Dibble89], the Intel Concurrent File System (CFS) on the iPSC/860 [Pierce89, Nitzberg92], the Scalable File System (sfs) on the CM-5 [LoVerso93], the CMMD I/O library also on the CM-5 [Best93], the ParFiSys parallel file system [Carretero96a, Carretero96b], and the Scotch parallel storage system [Gibson95]. These systems designed for parallel machines stripe data transparently across the available I/O devices thus reducing the bottleneck of slow disk access throughput. By hiding the underlying parallel nature of files, these parallel file systems provide compatibility with existing Unix file systems. Any deviation from the Unix semantics of stream files is painful for application developers.

The research community has agreed upon the fact that conventional sequential views of files such as "stream of bytes" (Unix-like interfaces) are ill-suited for parallel file systems and for high-performance parallel computing. This rudimentary interface prevents users from tailoring their I/O patterns to match the available disks and precludes any optimization of the I/O access pattern from different processes [Corbett96, Reed95, Kotz94b, Nieuwejaar94, Nieuwejaar96]. In other words, a conventional Unix-like interface does not provide an efficient method for application programmers to describing their I/O access patterns.

First generation parallel file systems are naive extensions of sequential files, focusing on the coordination of file pointers [Pierce89, Nitzberg92, Best93]. I/O access and prefetching do not consider any information about interleaved access patterns by different processes. This results in a lot of thrashing, lost bandwidth, and poor performance. New parallel file systems must feature mechanisms, i.e. interfaces, that let users control how the system manages I/O, how it distributes data across the storage devices, when and what it prefetches, and what caching algorithms it uses. They must also provide collective-I/O interfaces [Reed95, Kotz94b], in which all compute processors may cooperate to make single large requests, making it easier for the data management system to coordinate I/O for better performance [Rosario93, Nitzberg92].

However, a single solution cannot suit all applications [Kotz96, Krieger97]. Flexibility is needed for performance and the traditional functionality of parallel file systems should be separated into two components: a fixed core that is standard on all platforms, encapsulating only primitive abstractions and interfaces, and a set of high-level libraries providing a variety of abstractions and application-programmer interfaces (API). For better flexibility and performances, advanced parallel file systems should even allow application-specific code to run on I/O nodes directly where data reside. There are many benefits from running application-selected code on I/O nodes.

Application-specific optimizations can be applied to I/O-node caching and prefetching. Mechanisms like disk-directed I/O [Kotz94b] can be implemented. Incoming data can be filtered in a data-dependent way, passing only the necessary data to the compute nodes, saving network bandwidth and compute-node memory [Kotz95b]. Format conversion, compression, and decompression are also possible. In short, there are many ways that we can optimize memory and disk activity at the I/O node, and reduce disk and network traffic, by moving what is essentially application code to run at the I/O node in addition to the compute nodes.

5.3 The PS^2 philosophy

PS^2 addresses simultaneously two issues. The issue of developing efficient parallel processing applications on distributed memory PC's that perform intensive computations on large data sets, and the issue of developing parallel storage systems capable of declustering these large application data sets across multiple disks meeting the I/O requirement of these parallel applications. For high-performance, these two issues are deeply dependent. An application performing a parallel intensive computation on a large data set better knows how to distribute its data across multiple disks and how to access them in parallel. So far, the research community has separated parallel computation with parallel storage. On one hand we have parallel programming environments and toolkits (e.g. MPI [MPI94, MPI97], PVM [Sunderam90]) providing supports for parallel intensive computation, e.g. primitives for creating heavy-weight processes, primitives for synchronizing and coordinating parallel activities, primitives for transmitting information between processors. On the other hand we have parallel file systems (e.g. Scotch [Gibson95], Intel CFS [Pierce89], Galley [Nieuwejaar96], Vesta [Corbett96], HFS [Krieger97]) providing supports for parallel storage and parallel file access, e.g. file declustering, coordination of file pointers, prefetching, caching, abstractions of parallel files, collective I/O interfaces.

PS^2 offers an approach to parallel storage systems similar to Galley [Nieuwejaar96] and HFS [Krieger97] (Figure 5-1). Instead of designing a new parallel storage system that is intended to directly meet the specific needs of every parallel compute intensive application[1], PS^2 has been devised as an extensible framework for developing a wide variety of high-level parallel I/O- and compute- intensive libraries, each of which designed to meet the needs of specific applications. In other words, PS^2 enables breaking the traditional functionality of parallel storage systems by providing: comprising an extensible framework (PS^2) comprising a library of low-level reusable parallel file system components. On top of these parallel file system components, application programmers may build applications or high-level libraries providing a variety of file abstractions (e.g. 2D images, matricies) along with appropriate processing operations (e.g. filtering a 2D image, resampling a 2D image, LU matrix decomposition, matrix multiplication).

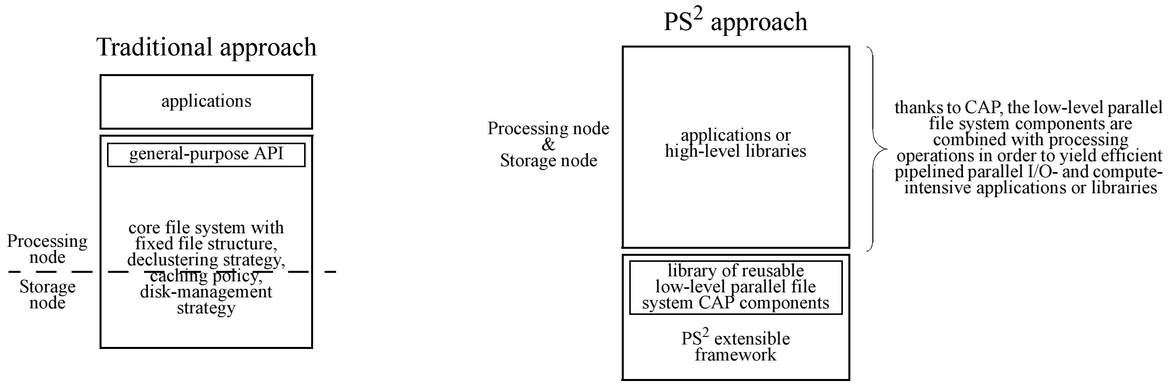

Traditional approach
applications
general-purpose API
core file system with fixed file structure, declustering strategy, caching policy, disk-management strategy
Processing node
Storage node
PS^2 approach
Processing node & Storage node
applications or high-level libraries
library of reusable low-level parallel file system CAP components
PS^2 extensible framework
thanks to CAP, the low-level parallel file system components are combined with processing operations in order to yield efficient pipelined parallel I/O- and compute-intensive applications or librairies

Figure 5-1. PS^2 offers an approach to parallel storage systems similar to Galley and HFS. Traditional systems depend on a fixed "core" file system that attempts to serve all applications through a common general-purpose API. With PS^2, the fixed core file system is shrunk to a library of reusable low-level parallel file system CAP components which can, thanks to the CAP formalism, be combined with processing operations in order to yield efficient pipelined parallel I/O- and compute- intensive applications or libraries.

1. i.e. separation between parallel processing and parallel storage.

PS^2 offers an extensible framework for developing parallel I/O- and compute- intensive high-level libraries on parallel storage-and-processing server architectures (PS^2 server architectures). A PS^2 server architecture comprises a number of PC's connected to a Fast Ethernet network and offering data storage and processing services to application threads located over the network on a number of client PC's. Contrary to most current massively parallel supercomputers [Feitelson95] (e.g. CM-5, Intel iPSC and Paragon, Meiko CS-2, IBM SP2), a PS^2 server architecture makes no difference between I/O nodes and compute nodes, i.e. there is no notion of I/O nodes dedicated to disk accesses and compute nodes dedicated to computations. Instead, PS^2 introduces storage and processing PC's, called *SP nodes*, offering data storage and processing services to application threads running on *client nodes*.

PS^2 defines an extensible CAP process hierarchy (Section 3.5) comprising a set of storage threads, a set of compute threads. Storage threads offer low-level parallel file system CAP components enabling to access files that are declustered across multiple disks distributed over several SP nodes. Library programmers can freely extend the functionalities of compute threads by incorporating library-specific processing operations. The storage threads and the compute threads are distributed between the SP nodes so as each contributing SP node comprises at least one storage thread performing parallel disk accesses on locally hooked disks and at least one compute thread running library-specific processing operations. Library programmers can, thanks to the CAP formalism, easily and elegantly extend the functionalities of the PS^2 process hierarchy by combining the predefined low-level parallel file access components with the library-specific processing operations in order to yield efficient pipelined parallel I/O- and compute- intensive CAP operations. Then, these parallel CAP operations can be incorporated into C/C++ high-level libraries offering specific abstractions of parallel files and processing routines on those abstractions.

To minimize the amount of data transferred between SP nodes and client nodes, library-specific processing operations can, thanks to the PS^2 flexibility and the CAP formalism, be directly executed on the SP nodes where data resides. A compute thread can process data that are read by its companion storage thread located on the same SP node. There are many benefits from running library-specific code on SP nodes, i.e. where data resides [Kotz96].

- Application or library programmers may conceive caching and prefetching mechanisms tailored to the application I/O access pattern on top of the PS^2 reusable low-level parallel file access components.
- Data accessed from disks may be processed and only the necessary data passed to the client nodes, avoiding superfluous data communication.
- Application data may be preprocessed and written to disk in appropriate format providing better performances, e.g. data compression.

With PS^2, developing a new parallel I/O- and compute- intensive operation operating on a large application data set declustered across multiple disks may consist in (example):

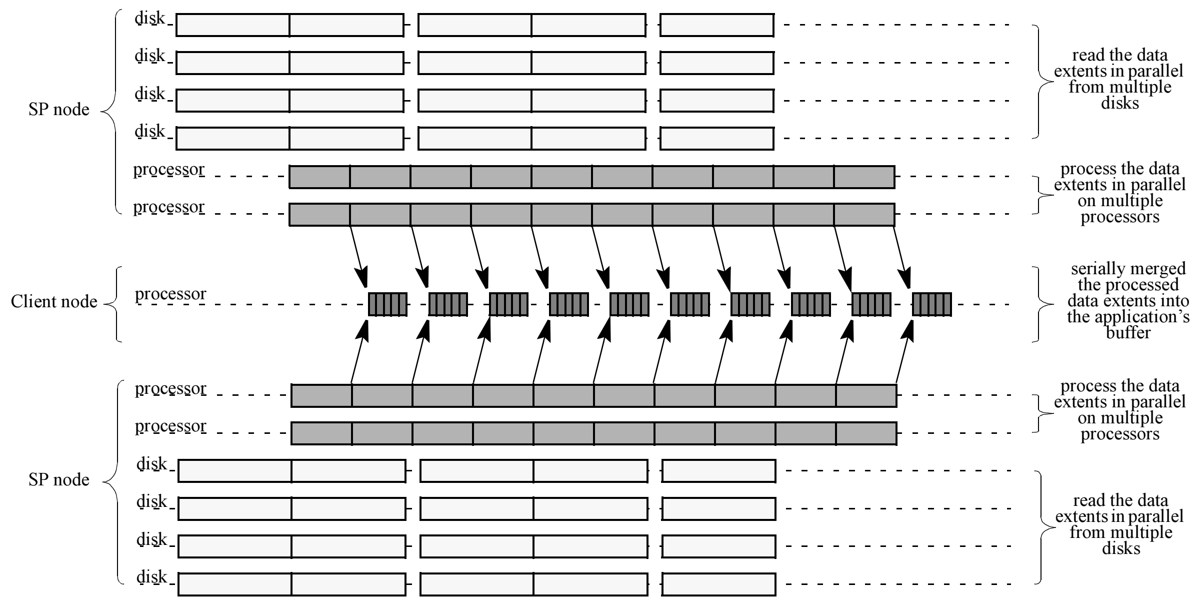
1. dividing the application data set into data parts, called data *extents*,
2. striping these data extents on several disks,
3. on the SP nodes, reading each of the data extents from the multiple disks,
4. on the SP nodes, performing a library-specific processing operation on each of the read data extents,
5. transmitting the processed data extents from the SP nodes to the client node,
6. and finally on the client node, merging the processed data extents into the application's buffer.

Thanks to the CAP methodology (Section 3.3), disk access operations[1] (i.e. reading data extents), library-specific processing operations (i.e. processing data extents), data communications (i.e. transmitting processed data extents to the client node), and collecting the results (i.e. merging the processed data extents) are executed in a pipelined parallel manner (Figure 5-2).

In a pipelined parallel execution (Figure 5-2), pipelining is achieved at three levels:

- A library-specific processing operation is performed by the SP node on one data extent while the SP node reads the next data extents,
- a processed data extent is asynchronously sent across the network to the client node while the next data extent is processed,
- a processed data extent is merged by the client node while the next processed data extent is asynchronously transmitted across the network from the SP node to the client node.

1. The low-level parallel file system components offered by the storage threads.



SP node
disk
disk
disk
disk
processor
processor
read the data extents in parallel from multiple disks
process the data extents in parallel on multiple processors
Client node
processor
serially merged the processed data extents into the application's buffer
SP node
processor
processor
disk
disk
disk
disk
process the data extents in parallel on multiple processors
read the data extents in parallel from multiple disks

Figure 5-2. With PS^2 disk accesses, computations, and communications are executed in a pipelined parallel manner

Parallelization occurs at two levels:

- several data extents are simultaneously read from different disks; the number of disks in the PS^2 server architecture can be increased to improve I/O throughput,
- application-specific or library-specific processing operation on data extents are done in parallel by several SP node processors; the number of SP node processors can be increased to improve processing performance.

Provided that there are enough disks hooked onto the SP nodes to meet the I/O requirement of an application, i.e. when the execution time of the parallel I/O- and compute- intensive operation is limited by the processing power at the SP nodes (Figure 5-2), a pipelined parallel execution enables hiding slow disk accesses and high-latency network communications. Therefore, PS^2 enables application programmers to design parallel I/O and compute operations on striped data sets that have the same execution time as if the large application data sets would be stored in huge main memories instead of being striped across multiple disks hooked on several SP nodes.

5.4 PS^2: a parallel storage and processing server based on commodity components

A parallel storage and processing server architecture (PS^2 server architecture) comprises several storage and processing nodes (SP nodes), each consisting of a number of processors and disks connected to a local area network (Figure 5-3). SP nodes offer data storage and processing services to clients located over the network. An example of a PS^2 server architecture is a cluster of 3 Bi-Pentium Pro 200MHz PC's, each with 9 disks distributed among 3 SCSI strings, and interconnected through a switched 100 Mbits/s Fast Ethernet network. PS^2 applications, i.e. applications developed using the PS^2 customizable parallel file system, can also run on a single mono-processor PC with a single disk.

Even on an architecture comprising a single mono-processor PC with a single disk, applications can benefit from the PS^2 methodology. Disk accesses and processing operations are pipelined, i.e. I/O's operate in a completely asynchronous manner, ensuring that either the processor or the disk is the bottleneck, i.e. is 100% active. By increasing the number of processors per SP node, the number of disks per SP node, or the number of SP nodes, we can potentially scale a PS^2 application, until the execution time is limited by one of the bottlenecks, i.e. disks, SP node processing power, SP node network interface, network throughput, client network interface, or client processing power.

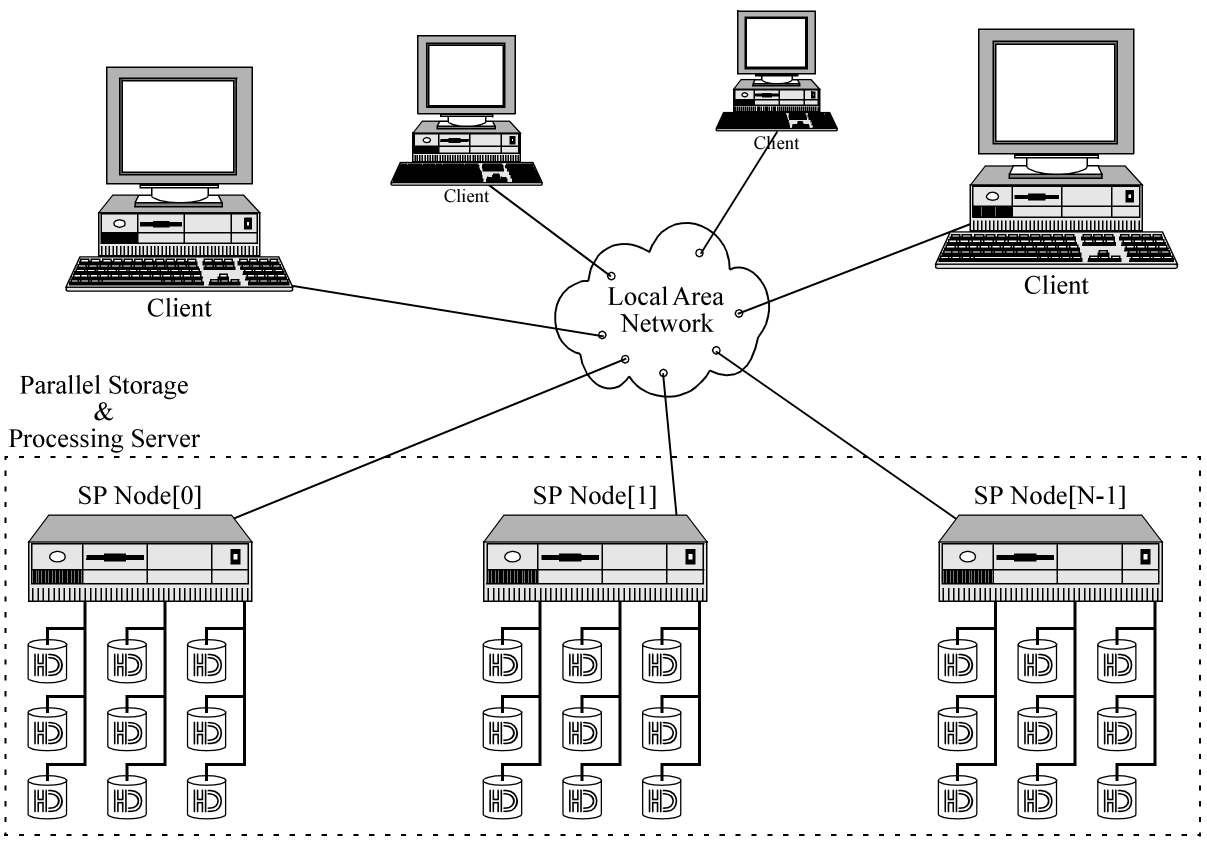


Figure 5-3. Architecture of the parallel storage and processing server based on commodity components, i.e. PC's, Fast Ethernet, SCSI-2 disks, Windows NT, etc.

5.5 Parallel file structure

PS^2 does not impose any declustering strategy on an application's data. Instead, PS^2 provides application or library programmers with the ability to fully, easily, and efficiently program their declustering strategy according to their own needs using the CAP language. This control is particularly important when implementing I/O-optimal algorithms [Cormen93]. To allow this behaviour, a *parallel file*, i.e. a file whose data is declustered across multiple *virtual disks*[1], is composed of one or more *extent files*, i.e. local files or subfiles, which may be directly addressed by the application using a 32-bit value called the *extent file index*. Each extent file resides entirely on a single virtual disk, and no virtual disk contains more than one extent file from any parallel file. When a parallel file is created, the programmer is asked to specify how many extent files the parallel file contains and on which virtual disks the extent files will be created. The number of extent files and their locations remain fixed throughout the life of the parallel file. The *striping factor* of a parallel file represents the number of extent files it contains, or in other words the number of virtual disks across which it is declustered.

As in the Galley parallel file system [Nieuwejaar96], the use of extent files gives applications the ability both to control how the data is distributed across the virtual disks, and to control the degree of parallelism exercised on every subsequent access. Of course, many application programmers will not want to handle low-level details such as the declustering algorithm and will even not program with the CAP language. We therefore anticipate that most end users will use high-level C/C++ libraries, e.g. an image library (Section 6.2), that provide a variety of abstractions with appropriate declustering strategies, but hide the details of these strategies from the end users.

1. Since PS^2 is based on top of native file systems, e.g. the Windows NT file system, the term virtual disk is more appropriate than the term disk. Indeed, the term virtual disk refers to a (NTFS) directory stored on a physical disk. Therefore, a same physical disk may contain several virtual disks. However, for the sake of simplicity, we assume in this dissertation that a physical disk cannot contain more than one virtual disk, i.e. a virtual disk corresponds to a (NTFS) directory on a dedicated physical disk.

5.5.1 Extent files

Each extent file is structured as a collection of *extents*. An extent within a particular extent file is addressable by a 32-bit value called the *local extent index*. An extent is a variable size block of data representing the unit of I/O transfer, i.e. applications may read or write entire extents only. In order to offer added flexibility for building C/C++ libraries on top of PS^2 parallel files, an extent is decomposed into a variable size header and a variable size body. Both are optional, i.e. an extent can contain only a header, only a body, or a header and a body which are stored contiguously on a virtual disk. For high-performance, an extent should have a size of approximately 50 KBytes in order to balance the disk latency time with the disk transfer time (Section 4.2). Unlike the number of extent files in a parallel file, the number of extents in an extent file is not fixed. Libraries and applications may add extents to or remove extents from an extent file at any time. There is no requirement that all extent files have the same number of extents, or that all extents have the same size.

The final two-dimensional parallel file structure of PS^2 is shown in Figure 5-4. A particular extent is addressed using two 32-bit unsigned values: an extent file index ranges from 0 to N-1, where N is the striping factor, and a local extent index within that extent file which ranges from 0 to (2^32)-1. Extents within an extent file are not necessarily stored by successive extent indices, i.e. an extent file may contain extents with scattered indices.

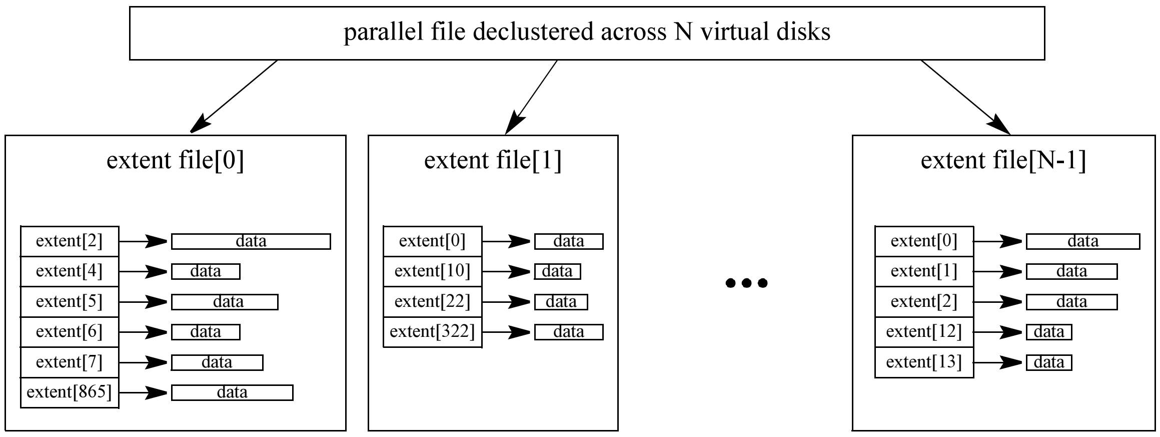


Figure 5-4. A parallel file is composed of one or more extent files. Each extent file is structured as a collection of extents and resides entirely on a single virtual disk, and no virtual disk contains more than one extent file from any parallel file. An extent is a variable size block of data representing the unit of I/O transfer. An extent within a particular extent file is directly addressed by the application using an extent file index selecting the extent file and an local extent index selecting the extent within that extent file.

The use of such a two-dimensional parallel file structure (Figure 5-4) provides library and application programmers with the ability to fully control:

- the declustering strategy,
- the location of each basic striping unit (extent),
- and the degree of parallelism exercised on every subsequent access.

Extents comprising headers and bodies are likely to be useful when implementing parallel data access libraries. In addition to storing the application's data, many libraries also need to store persistent, library-specific metadata. One example of such a library is the image-oriented library for PS^2 (Section 6.2). Along with each 2D tile, a tile descriptor indicates whether the tile is compressed or not and the compression algorithm used. With PS^2, such a library stores each 2D tile in an extent body and its associated meta-information in the extent header. Another example of such a library would be the storage of compressed files according to [Seamons95]. Rather than compressing the whole file at once, making it difficult to modify or extract data in the middle of the file, the file is broken into a series of chunks, which are compressed independently. With PS^2, such a storage system could store one uncompressed or compressed data chunk into one extent body and the necessary meta-information about that chunk in the corresponding extent header.

PS^2 ensures that an extent comprising a variable size header and a variable size body is always stored contiguously in the NTFS extent file, i.e. the header and the body of an extent form the unit of I/O transfer. Recall that extents can have any sizes even within a same extent file and the user when reading an extent is not asked to provide its header size and its body size. Besides reading the extent's data, PS^2 also retrieves its header size and its body size mostly without any additional overhead, i.e. without additional NTFS I/O requests (Section 5.10.2). By doing this, a parallel data access library is able to read or write with one NTFS I/O request both library-specific metadata and application data (which can have variable sizes), thus reducing the number of I/O accesses and increasing the effective disk throughput. This is not feasible with conventional sequential views of files where the library must first read from one file the library-specific metadata providing the file extent size and other necessary information for accessing the extent and then read the application's data extent from another file.

Moreover, the use of extents as the basic unit of NTFS I/O transfer enables applications to have full control on the number of NTFS I/O requests and on their sizes. Since for performance consideration (Section 4.2), the NT file system cache is disabled for all extent accesses, NTFS imposes that all I/O requests are aligned on 512 Bytes (both the position and the size of the request) corresponding to the standard SCSI-2 logical disk block size. Then, if an application reads an extent whose size is 513 Bytes, 1024 Bytes of data are read. By specifying the extent size, a fine control over the NTFS I/O requests is exercised. It optimizes the effective disk I/O throughput influenced by the number of disk arm motions and the amounts of data read in each seek.

5.6 PS^2 file tree and its internal representation

PS^2 organizes parallel files as a tree with a single root node called *root* (written "/"); every non-leaf node of the parallel file system structure is a *directory* of files, and files at the leaf nodes of the tree are either empty directories or *parallel files*. The name of a file, i.e. either a directory or a parallel file, is given by a *path name* that describes how to locate the file in the PS^2 hierarchy. A path name is a sequence of component names separated by slash characters "/"; a component is a sequence of characters that designates a file name that is uniquely contained in the previous directory component. In PS^2 a path name starts always with a slash character (absolute path name) and specifies a file that can be found by starting at the file system root, and traversing the file tree, following the branches that lead to successive component names of the path name. Thus, the path names "/Images/Lausanne.ps2", "/Archive/Sun-Star.ps2", and "/Archive/Old" designates two parallel files and one directory in the PS^2 tree shown in Figure 5-5. Note that the ".ps2" extension is not interpreted by PS^2. A file, i.e. a directory or a parallel file, may have any name. In particular with or without an extension. This hierarchical structure of files is similar to the UNIX file system tree [Bach86] except that PS^2 does not support relative path names, i.e. path names that are designated relative to the current directory of the process.

To implement such a hierarchical directory structure (Figure 5-5, PS^2 tree), PS^2 relies on the native Windows NT file system, i.e. the PS^2 directory tree is replicated on all disks starting at their roots (Figure 5-5, NTFS trees). For example, when a user creates a PS^2 directory, e.g. "/Archive/New", internally and in parallel, PS^2 creates an NTFS directory on each disk whose path name is the concatenation of the disk's root path name with the PS^2 directory path name, i.e. on PC[0] the "D:\PsPs\Disk0\Archive\New" directory is created, on PC[1] the "M:\Demo\PsPs\Disk1\Archive\New" directory is created and on PC[2] the "C:\PsPs\Disk2\Archive\New" directory is created. The same technique is used to create parallel files, but instead of creating an NTFS extent file on each disk, PS^2 creates an extent file only on the requested disks. For example, when a user creates a PS^2 parallel file, e.g. "/Images/New.ps2" on the disks 2 and 0 (striping factor of 2), internally and in parallel, PS^2 creates the first NTFS extent file (extent file index = 0) "C:\PsPs\Disk2\Images\New.ps2" on PC[2] and the second NTFS extent file (extent file index = 1) "D:\PsPs\Disk0\Images\New.ps2" on PC[0].

When an application wishes to open or delete a parallel file, PS^2 must first locate the extent files composing that parallel file, i.e. locate on which disks the extent files reside. Contrary to some other parallel file systems, e.g. the Galley file system [Nieuwejaar96] or the Vesta file system [Corbett96], PS^2 does not rely on metadata files to locate extent files, instead PS^2 interrogates in parallel the native file system (NTFS) on each disk. For example, to locate the "/Archive/Daemon.ps2" parallel file, PS^2 interrogates in parallel the first disk on PC[0] to check the existence of the "D:\PsPs\Disk0\Archive\Daemon.ps2" extent file, the second disk on PC[1] to check the existence of the "M:\Demo\PsPs\Disk1\Archive\Daemon.ps2" extent file, and the third disk on PC[2] to check the existence of the "C:\PsPs\Disk2\Archive\Daemon.ps2" extent file. Based on the NTFS answers, PS^2 is able to

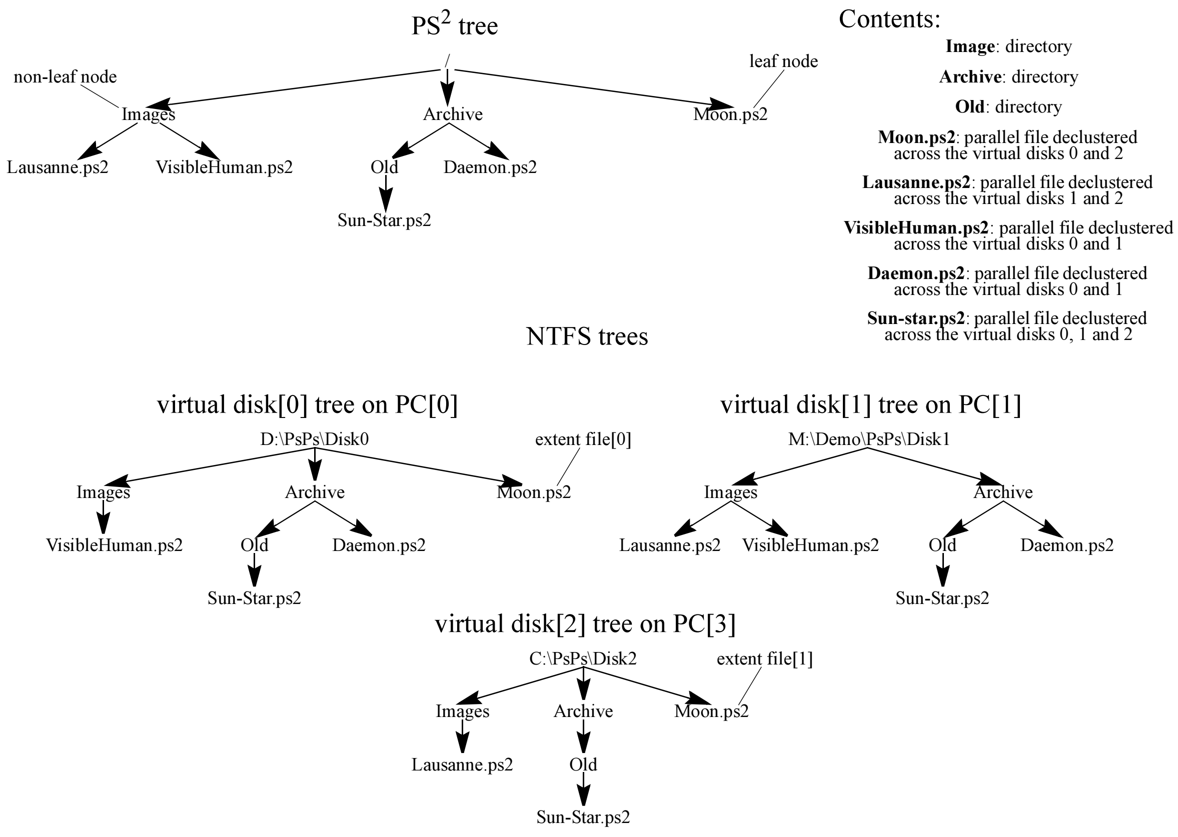

PS2 tree
/
non-leaf node
leaf node
Images
Archive
Moon.ps2
Lausanne.ps2
VisibleHuman.ps2
Old
Daemon.ps2
Sun-Star.ps2
Contents:
Image: directory
Archive: directory
Old: directory
Moon.ps2: parallel file declustered across the virtual disks 0 and 2
Lausanne.ps2: parallel file declustered across the virtual disks 1 and 2
VisibleHuman.ps2: parallel file declustered across the virtual disks 0 and 1
Daemon.ps2: parallel file declustered across the virtual disks 0 and 1
Sun-star.ps2: parallel file declustered across the virtual disks 0, 1 and 2
NTFS trees
virtual disk[0] tree on PC[0]
D:\PsPs\Disk0
extent file[0]
virtual disk[1] tree on PC[1]
M:\Demo\PsPs\Disk1
virtual disk[2] tree on PC[3]
C:\PsPs\Disk2
extent file[1]

Figure 5-5. The mapping of a PS^2 tree onto 3 virtual disks located on PC[0] with root D:\PsPs\Disk0, on PC[1] with root M:\Demo\PsPs\Disk1 and on PC[2] with C:\PsPs\Disk2

determine the striping factor of the parallel file and the location of each extent file, i.e. extent file[0] through extent file[StripingFactor-1]. This method of locating extent files works even if a user mixes up the extent files of a parallel file, e.g. when copying extent files from one location to another using the Windows "Explorer" application. For improved scalability and performance, PS^2 could maintain a cache of metadata distributed on all storage and processing PC's. To find the PC that manages a given parallel file's metadata, a hash function may be applied to the parallel file name. Vesta and Galley use this hashing scheme for their naming system [Corbett96, Nieuwejaar96].

PS^2 provides CAP operations to create and delete a directory, and list the contents of a directory. Of course, PS^2 applies the same rules as the UNIX file system, e.g. a directory may be deleted only if it is empty. Regarding protection of files (directories or parallel files), PS^2 relies on the native Windows NT file system and does not implement any operations to modify file access permissions.

In order to alleviate the task of managing such a replicated tree with numerous extent files distributed over the disks (Figure 5-5), future releases of PS^2 should provide a kit of commands similar to the Windows NT commands, e.g. Cacls, Copy, Dir, Del, Mkdir, Move, etc, but instead of performing an operation on NTFS files, the operation is performed on PS^2 files, i.e. the corresponding Windows NT command is performed on all NTFS files, i.e. either directories or extent files, forming the PS^2 files. For example copying a PS^2 parallel file consists of copying all the extent files, which are NTFS files, distributed in several directories to another locations. Another approach would be to extend the functionalities of the "Explorer" Windows NT application. Instead of displaying the NTFS extent files, we customize the "Explorer" application so that it treats a group of NTFS extent files as a single PS^2 parallel file (Figure 5-6).

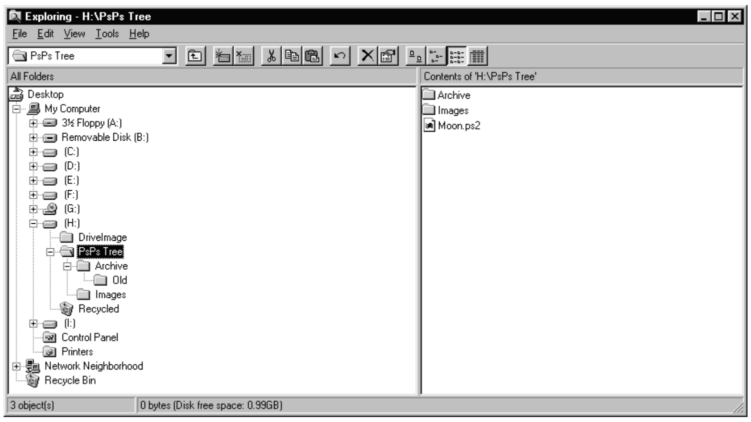


Figure 5-6. Explorer treats the Moon.ps2 parallel file as a single regular file but in fact behind that file there are 2 NTFS extent files residing at two different locations D:\PsPs\Disk0\Moon.ps2 on PC[0] and C:\PsPs\Disk2\Moon.ps2 on PC[2]

5.7 The PS^2 access structures

As introduced in Section 5.3, PS^2 defines a CAP process hierarchy, henceforth called the *PS^2 server process hierarchy* or simply the *PS^2 server,* offering reusable low-level parallel file system CAP components.

In order to introduce some terminologies and programming structures, Program 5-1 shows a simplified version of the PS^2 server process hierarchy. A PS^2 server (lines 1-9) consists in a number of *VirtualDiskServer* hierarchical processes (line 4) and a number of *ComputeServer* threads (line 5). A *VirtualDiskServer* process (lines 18-22) only contains one *ExtentServer* thread[1] (line 21). To summarize, a PS^2 server contains a set of *ComputeServer* threads (*ComputeServer[0]*, *ComputeServer[1]*, ...) and a set of *ExtentServer* threads (*VirtualDiskServer[0].ExtentServer*, *VirtualDiskServer[1].ExtentServer*, ...). The *ComputeServer* threads perform application-specific or library-specific processing operations on extent data, i.e. computations. The *ExtentServer* threads perform extent access operations, i.e. disk I/O's.

```
 1 process Ps2ServerT
 2 {
 3 subprocesses:
 4   VirtualDiskServerT VirtualDiskServer[];
 5   ComputeServerT ComputeServer[];
 6
 7 operations:
 8   ... // parallel file directory operations
 9 }; // end process Ps2ServerT
10
11 process ComputeServerT
12 {
13 operations: // must remain empty
14   // A ComputeServer thread does not offer any
15   // predefined operations. It is fully customizable.
16 }; // end process ComputeServerT
17
18 process VirtualDiskServerT
19 {
20 subprocesses:
21   ExtentServerT ExtentServer;
22 }; // end process VirtualDiskServerT
23
24 process ExtentServerT
25 {
26 operations:
27   ... // extent access operations
28 }; // end process VirtualDiskServerT
29
```

Program 5-1. CAP specification of the PS^2 server (simplified version)

1. In the simplified PS^2 server declaration (Program 5-1) the use of an intermediate *VirtualDiskServer* hierarchical process containing a single *ExtentServer* thread may seem cumbersome. Section 5.8 explains the reason of this additional level in the hierarchy of process.

A particular *ExtentServer* thread is addressed by a 32-bit *VirtualDiskServerIndex* value ranging from 0 to *NumberOfVirtualDiskServers*[1]-1, e.g. *VirtualDiskServer[5].ExtentServer* designates the 6th *ExtentServer* thread in the PS^2 server process hierarchy. A particular *ComputeServer* thread is addressed by a 32-bit *ComputeServerIndex* value ranging from 0 to *NumberOfComputeServers*[2]-1, e.g. ComputeServer[7] designates the 8th *ComputeServer* thread in the PS^2 server process hierarchy.

The library of reusable low-level parallel file system CAP components is divided into two parts (Program 5-1). The first part (line 8) provided by the *PS2ServerT* hierarchical process consists in parallel file directory operations, i.e. parallel operations manipulating parallel files (open, close, create, and delete) and directories (create, delete, and list). The second part (line 27) provided by the *ExtentServerT* thread consists in extent access operations, i.e. sequential operations accessing extents in extent files (read, write, and delete).

The *ComputeServer* threads do not offer any processing operations (Program 5-1, lines 14-15). They are customized, i.e. their functionalities are extended, by programmers who incorporate application-specific or library-specific processing operations. Thanks to a CAP configuration file (Section 3.6), the *ComputeServer* threads performing the added processing operations can be mapped into the same Windows NT processes as the *ExtentServer* threads performing extent access operations, i.e. I/O's. Therefore, disk I/O's and computations can be closely coupled for high performance.

When developing new parallel I/O- and compute- intensive operations, the functionality of the *PS2ServerT* (Program 5-1, lines 1-9) hierarchical process is extended, i.e. application or library programmers define new parallel operations by combining the application-specific or library-specific processing operations (provided by the *ComputeServer* threads) with the predefined extent access operations (provided by the *ExtentServer* threads) using the parallel CAP constructs described in Chapter 3. Program 5-2 shows a simplified example where a new *Ps2ServerT::ParallelIOAndComputeOperation* parallel processing operation on parallel files (line 8) is incorporated into the *PS2ServerT* hierarchical process. This new parallel operation combines an application-specific or library-specific processing operation with an extent access operation taken from the PS^2 library of reusable low-level parallel file system components (lines 15-17).

```
 1 leaf operation ComputeServerT::ProcessingOperation
 2   in ...
 3   out ...
 4 {
 5   // C/C++ code
 6 } // end leaf operation ComputeServerT::ProcessingOperation
 7
 8 operation Ps2ServerT::ParallelIOAndComputeOperation
 9   in ...
10   out ...
11 {
12   // Parallel CAP constructs combining processing operations with extent access operations
13
14   // For example:
15   ComputeServer[ComputeServerIndex].ProcessingOperation
16   >->
17   VirtualDiskServer[VirtualDiskServerIndex].ExtentServer.WriteExtent
18   ...
19 } // end operation Ps2ServerT::ParallelIOAndComputeOperation
```

Program 5-2. How the PS^2 framework is customized for developing a new parallel processing operations on parallel files (simplified example)

Application or library programmers need to be able:

1. The *NumberOfVirtualDiskServers* value is a PS^2 application parameter provided as an argument when launching the PS^2 application (Section 5.11). The number of virtual disks must be multiple of the number of virtual disk servers.
2. The *NumberOfComputeServers* value is a PS^2 application parameter provided as an argument when launching the PS^2 application (Section 5.11). The number of virtual disks must be multiple of the number of compute servers.

- to direct an extent access request[1] to an *ExtentServer* thread where the desired extent file is accessible[2], i.e. to compute the *VirtualDiskServerIndex* value selecting the appropriate *ExtentServer* thread in the PS^2 server process hierarchy (Program 5-2, line 17),
- to direct a processing request[3] to a *ComputeServer* thread which is located in the same address space as the *ExtentServer* thread who performs the extent access operation (read, write, or delete extent), i.e. to compute the *ComputeServerIndex* value selecting the appropriate *ComputeServer* thread in the PS^2 server process hierarchy (Program 5-2, line 15),

PS^2 assigns the virtual disks to the *ExtentServer* threads, and distributes, thanks to a CAP configuration file (Section 3.6), the *ExtentServer* threads and the *ComputeServer* threads onto the PS^2 server architecture (Figure 5-3) according to the following rules:

1. The virtual disks are evenly distributed between the *ExtentServer* threads, i.e. PS^2 assigns to each *ExtentServer* thread the same number of virtual disks. For example, if a PS^2 application is launched with 10 *ExtentServer* threads and 20 virtual disks, PS^2 assigns 2 virtual disks per *ExtentServer* threads, i.e. the *VirtualDiskServer[0].ExtentServer* thread accesses extents located only in the 0th and 1st virtual disks[4], the *VirtualDiskServer[1].ExtentServer* thread accesses extents located only in the 2nd and 3rd virtual disks, etc, and the *VirtualDiskServer[9].ExtentServer* thread accesses extents located only in the 18th and 19th virtual disks.
2. The *VirtualDiskServer* hierarchical processes (*ExtentServer* threads) are evenly distributed between the SP nodes, i.e. there is the same number of *ExtentServer* threads per SP node.
3. The *ComputeServer* threads are evenly distributed between the SP nodes, i.e. there is the same number of *ComputeServer* threads per SP node.
4. The *ExtentServer* threads and the *ComputeServer* threads located on the same SP node reside in a same Windows NT process so as to benefit from the shared memory for transferring tokens from *ExtentServer* threads to their companion *ComputeServer* threads and vice versa.

Figure 5-7 depicts an example on how PS^2 assigns the virtual disks to the *ExtentServer* threads, and how the *ExtentServer* threads and the *ComputeServer* threads are distributed onto a 3-SP node PS^2 server architecture. The 18 virtual disks (corresponding to 18 NTFS directories in 18 different physical disks) are evenly distributed between the 6 *ExtentServer* threads, i.e. the *VirtualDiskServer[0].ExtentServer* thread accesses extents only located in the 0th, 1st and 2nd virtual disks, the *VirtualDiskServer[1].ExtentServer* thread accesses extents only located in the 3rd, 4th and 5th virtual disks, the *VirtualDiskServer[2].ExtentServer* thread accesses extents only located in the 6th, 7th and 8th virtual disks, etc, and the *VirtualDiskServer[5].ExtentServer* thread accesses extents only located in the 15th, 16th and 17th virtual disks. The 6 *ExtentServer* threads are evenly distributed between the 3 SP nodes, i.e. the *VirtualDiskServer[0-1].ExtentServer* threads reside in the 0th SP node, the *VirtualDiskServer[2-3].ExtentServer* threads reside in the 1st SP node, the *VirtualDiskServer[4-5].ExtentServer* threads reside in the 2nd SP node. The 9 *ComputeServer* threads are also evenly distributed between the 3 SP nodes, i.e. the *ComputeServer[0-1-2]* threads reside in the 0th SP node, the *ComputeServer[3-4-5]* threads reside in the 1st SP node, and the *ComputeServer[6-7-8]* threads reside in the 2nd SP node.

Based on the above 4 static mapping rules, the PS^2 framework provides application or library programmers with structures and macros enabling to redirect extent access requests to appropriate *ExtentServer* threads, i.e. to compute the *VirtualDiskServerIndex*, and to redirect processing requests to appropriate *ComputeServer* threads, i.e. to compute the *ComputeServerIndex*. In order to be able to develop processing operations on parallel files,

1. An extent access request is a token that is redirected to an *ExtentServer* thread for performing an extent access operation (read, write, and delete) on an extent file belonging to a parallel file. The extent access operations are part of the library of reusable low-level parallel file system CAP components.
2. An extent access request must be directed to an *ExtentServer* thread located on the SP node where the accessed disk (i.e. the disk where the desired extent file resides) is hooked onto.
3. A processing request is a token that is redirected to a *ComputeServer* thread for performing an application-specific or a library-specific processing operation on extent data.
4. In other words, the *ExtentServer[0]* thread accesses extents located only in extent files stored in the 0th and 1st virtual disks.

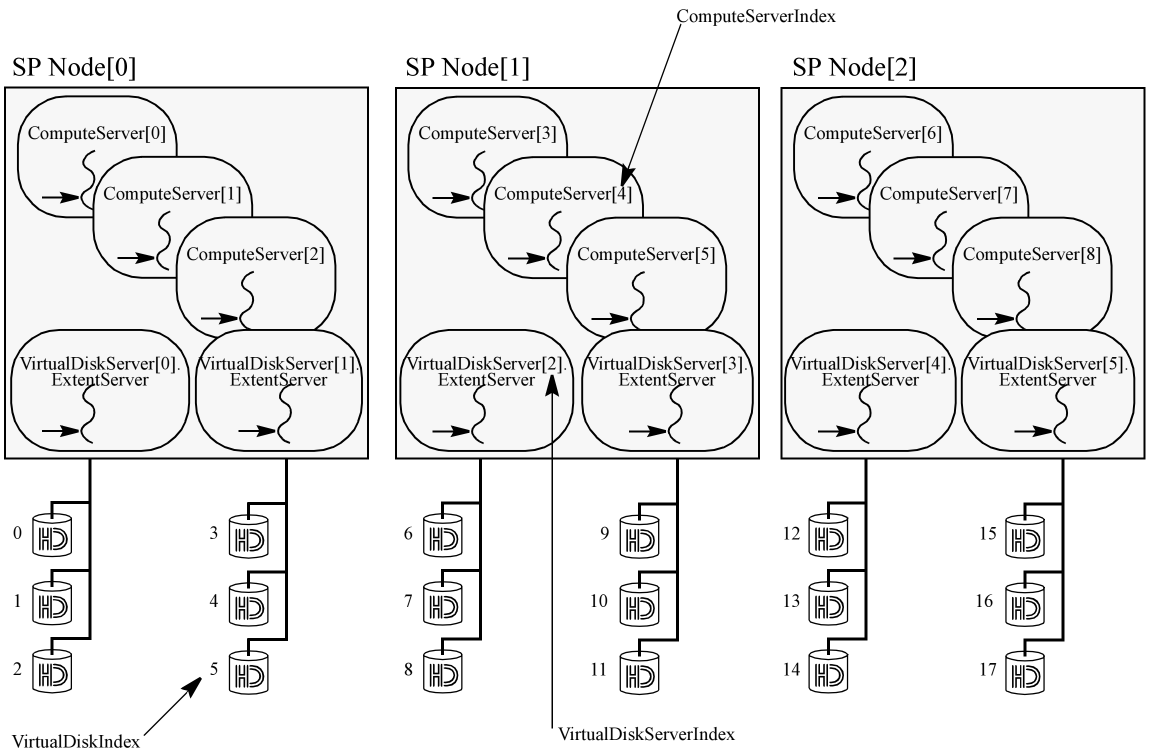


Figure 5-7. How PS^2 assigns the virtual disks to the *ExtentServer* threads and how PS^2, thanks to a CAP configuration file, distributes the *VirtualDiskServer[].ExtentServer* threads and the *ComputeServer[]* threads onto a PS^2 server architecture

when opening a parallel file, PS^2 returns a table of *VirtualDiskIndex* specifying for each extent file its location, i.e. on which virtual disk it resides. For example, the *VirtualDiskIndex[k]* value specifies the index of the virtual disk where the kth extent file resides. Based on the location of the extent files making up a parallel file (the *VirtualDiskIndex* mapping table), application or library programmers are able to redirect extent access requests and processing requests using the two macros shown in Programs 5-3 and 5-4.

Program 5-3 shows the macro converting the index of the virtual disk where the kth extent file resides (*VirtualDiskIndex[k]*) to a virtual disk server index designing the *ExtentServer* thread that is able to access the extents in that extent file. Since PS^2 evenly distributes the virtual disks between the *VirtualDiskServer* hierarchical processes, i.e. the *ExtentServer* threads (Figure 5-7), a virtual disk server index is computed by dividing (integer division) a virtual disk index by the number of virtual disks assigned per *VirtualDiskServer* hierarchical process (i.e. *NumberOfVirtualDisks* / *NumberOfVirtualDiskServers*). Note that PS^2 imposes that the *NumberOfVirtualDisks* be multiple of the *NumberOfVirtualDiskServers.*

```
inline int ToVirtualDiskServerIndex(int virtualDiskIndex)
{
  return virtualDiskIndex / (NumberOfVirtualDisks / NumberOfVirtualDiskServers);
} // end ToDiskServerIndex
```

Program 5-3. Macro that convert a virtual disk index to a virtual disk server index

Program 5-4 shows the macro converting the index of the virtual disk where the kth extent file resides (*VirtualDiskIndex[k]*) to a compute server index designing the *ComputeServer* thread located in the same address space as the *ExtentServer* thread accessing extents in that extent file. Since PS^2 evenly distributes the virtual disks between the *ComputeServer* threads (Figure 5-7), a compute server index is computed by dividing (integer division) a virtual disk index by the number of virtual disks assigned per *ComputeServer* threads (i.e. *NumberOfVirtualDisks* / *NumberOfComputeServers*). Note that PS^2 imposes that the *NumberOfVirtualDisks* be multiple of the *NumberOfComputeServers*.

```
inline int ToComputeServerIndex(int virtualDiskIndex)
{
  return virtualDiskIndex / (NumberOfVirtualDisks / NumberOfComputeServers);
} // end ToComputeServerIndex
```

Program 5-4. Macro that convert a virtual disk index to a compute server index

Program 5-5 shows an example on how to redirect extent access requests on *ExtentServer* threads and processing requests on *ComputeServer* threads based on the indices of the extent files making up a parallel file (in Figure 5-5, *ExtentFileIndex* = 2). At line 5, the processing request is redirected to the *ComputeServer* thread located in the same address space as the *ExtentServer* thread at line 7, i.e. the *ExtentServer* thread accessing extents in the 2nd extent file. At line 7, the extent access request is redirected to the *ExtentServer* thread accessing extents in the 2nd extent file. By doing this, application or library programmers are able to redirect the extent access requests to SP nodes where the extent files reside, i.e. for better performance, each SP node issues disk I/O requests only to locally hooked physical disks. Moreover, programmers are also able to redirect the processing requests on the same SP nodes where the extent access requests are performed.

```
1  operation Ps2ServerT::ParallelIOAndComputeOperation
2    in ...                                                    No of extent file
3    out ...
4  {
5    ComputeServer[ToComputeServerIndex(thisTokenP->VirtualDiskIndex[2])].ProcessingOperation
6    >->
7    VirtualDiskServer[ToVirtualDiskServerIndex(thisTokenP->VirtualDiskIndex[2])].ExtentServer.WriteExtent
8  } // end operation Ps2ServerT::ParallelIOAndComputeOperation
```

Program 5-5. Thanks to the *ToComputeServerIndex* and *ToVirtualDiskServerIndex* macros, and the *VirtualDiskIndex* table of a parallel file, application or library programmers are able to redirect extent access requests to the *ExtentServer* thread capable of accessing extents in the desired extent file (in figure the 2nd extent file) and to redirect processing operations to the *ComputeServer* thread located in the same address space as the *ExtentServer* thread accessing extents in the desired extent file (in figure the 2nd extent file)

5.8 Synthesizing the PS^2 parallel storage and processing server using the CAP computer-aided parallelization tool

The PS^2 server process hierarchy is composed of 4 types of CAP threads:

- There is an *InterfaceServer* thread responsible for coordinating parallel file directory operations, i.e. open parallel file, close parallel file, create parallel file, delete parallel file, create directory, delete directory, and list directory. The *InterfaceServer* thread maintains a list of open parallel files and a list of ongoing file operations so as to be able to resolve race conditions on files (parallel files and directories), e.g. when simultaneously a first *application thread* creates a parallel file and a second *application thread* deletes the directory into which the first thread wants to create the parallel file. In case of a race condition, the *InterfaceServer* thread serializes the offending parallel file directory operations preventing PS^2 from becoming inconsistent.
- There are *ExtentFileServer* threads offering extent file directory operations, i.e. open extent file, close extent file, create extent file, delete extent file, create extent file directory, delete extent file directory, and list extent file directory. Each *ExtentFileServer* thread is based on an extent-oriented single-disk file system, called EFS *Extent File System* (Section 5.10). Since the extent file directory calls of EFS (open, close, create, delete, list) are synchronous, a given *ExtentFileServer* thread can handle one extent file directory request at a time, i.e. the extent file directory operations are processed by an *ExtentFileServer* thread sequentially one after the other without any overlapping with subsequent extent file directory operations; even if the subsequent extent file directory operations involve different virtual disks, e.g. the first extent file directory operation is to create an extent file in "C:\PsPs\Disk1" and the second extent file directory operation is to delete an extent file in "D:\PsPs\Disk2". Since we assume that simultaneous extent file directory operations are rare, no effort has been made to make the *ExtentFileServer* work asynchronously.
- There are *ExtentServer* threads offering extent access operations, i.e. read extent, write extent, and delete extent. As the *ExtentFileServer* threads, each *ExtentServer* thread is based on EFS (Section 5.10) for reading, writing and deleting extents. Contrary to the extent file directory calls, the extent calls of EFS are

asynchronous thus enabling a single *ExtentServer* thread to read, write, and delete extents in parallel from multiple virtual disks. Moreover, the use of asynchronous I/O calls allows to pipeline I/O requests on each I/O device, i.e. to have more than one outstanding I/O request per virtual disk (Section 4.2, filling factor). Recent disks benefit from receiving several I/O requests; this improves the efficiency of their advanced technologies such as the rotational-seek sorting [Patwardhan94] and multiple-head parallel access [Sutton94].

- There are *ComputeServer* threads responsible for handling application- or library- specific processing operations. A *ComputeServer* thread, thanks to the CAP formalism (Section 3.7, additional operation declaration), is fully extensible (or customizable). By default a *ComputeServer* thread does not offer any sequential operations. Application or library programmers are free to add processing operations to *ComputeServer* threads and to combine these added functionalities with the predefined low-level parallel file access components offered by the *ExtentServer* threads.

An *ExtentFileServer* thread grouped with an *ExtentServer* thread represents a *VirtualDiskServer* hierarchical CAP process. Having two threads, one offering extent file directory operations and the other offering extent access operations, per *VirtualDiskServer* process enables extents to be asynchronously read, written, and deleted, with one thread, while the other thread is synchronously running extent file directory operations. Moreover, extent access operations can be handled at a higher priority than extent file directory operations by having the priority of the *ExtentServer* thread greater than the priority of the *ExtentFileServer* thread.

We can assign several virtual disks to a same *VirtualDiskServer* process, i.e. a single *ExtentFileServer* thread and a single *ExtentServer* thread access asynchronously extent files located on different virtual disks. This feature is essential for reducing the number of running threads per SP nodes. Although threads are lighter than Windows NT processes, they still require memory and consume resources such as the processor to run the scheduler for a context switching. For example, we have demonstrations where a single PC must handle up to 60 virtual disks. For these kind of applications, it would be ridiculous to have 60 *VirtualDiskServer* processes, i.e. 120 threads, when 2 threads are sufficient.

Figure 5-8 shows an example on how the PS^2 threads can be mapped onto a PS^2 server comprising N SP nodes. On the client side, the *application threads* and the *InterfaceServer* thread are running. On each SP node, an *ExtentFileServer* thread, an *ExtentServer* thread, and a number of *ComputeServer* threads are running. The *application threads* communicate with the *InterfaceServer* thread for all parallel file directory operations. The *InterfaceServer* thread communicates with the *ExtentFileServer* threads to execute these parallel file directory operations. In order to maintain the consistency between the virtual disks, an *application thread* never communicates directly with the *ExtentFileServer*; for example for creating an extent file. On the other hand, *application threads* communicate directly with the *ExtentServer* threads to read extents, write extents, and delete extents. For pipelined parallel I/O and compute operations, the *application thread*, the *ComputeServer* threads and the *ExtentServer* threads interact with each other as specified in the corresponding CAP parallel constructs.

The thread-to-process mapping defined by a configuration text file (Sections 3.6 and 5.11) is under the control of application or library programmers. According to the PS^2 server architecture and to the application itself, programmers may adapt the PS^2 process and thread configuration. For example application or library programmers may:

- increase the number of *ComputeServer* threads per SP nodes in order to benefit from multiprocessor SP PC's,
- map the *ComputeServer* threads into the same Windows NT processes as the *ExtentServer* threads so as to benefit from the shared memory to exchange data extents without copies,
- map the *ComputeServer* threads onto specific compute PC's and map the *ExtentServer* threads onto specific I/O PC's,
- map the *InterfaceServer* thread on one of the SP nodes, either in a separate Windows NT process or in a same Windows NT process as a *ComputeServer* thread or as an *ExtentServer* thread.

Although the mapping of the PS^2 threads is under the control of application programmers, some restrictions are imposed:

- There must be always a single *InterfaceServer* thread somewhere in the network, either in a pre-existing Windows NT process or in a completely separate Windows NT process.
- Since an *ExtentFileServer* thread and its companion *ExtentServer* thread (forming together a *VirtualDiskServer* hierarchical CAP process) share extent file objects through a shared memory, i.e. the *ExtentFileServer* thread creates and deletes extent file objects and the *ExtentServer* thread uses these objects for reading, writing, and deleting extents, they must be mapped into a same Windows NT process.

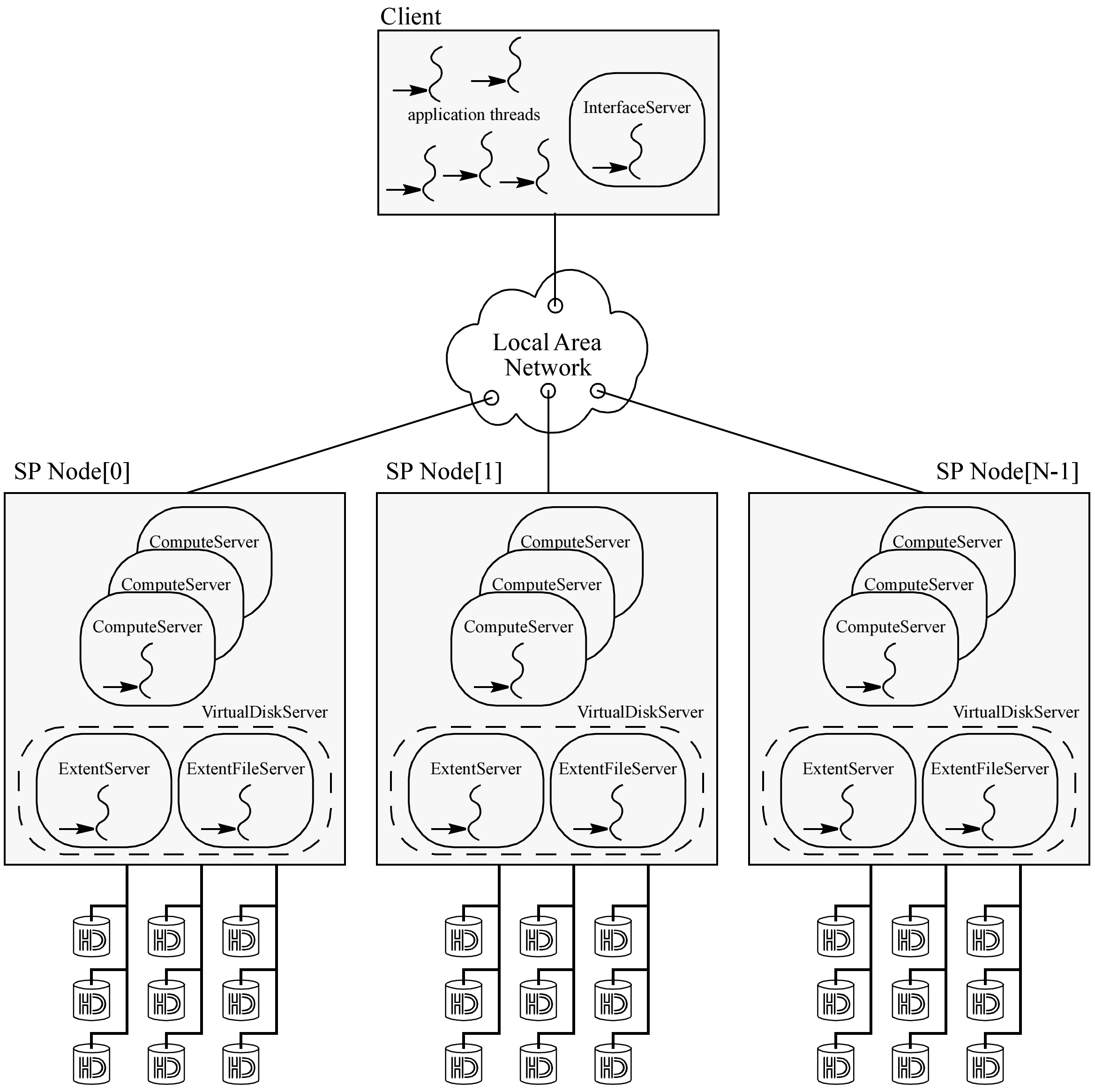


Figure 5-8. PS^2 threads and how they are mapped onto a PS^2 server architecture

- The *VirtualDiskServer* processes, i.e. the *ExtentFileServer* threads and the *ExtentServer* threads, must be located on PC's where the assigned virtual disks, i.e. the NTFS directories, are accessible. For example, if a local NTFS directory, e.g. "C:\PsPs\Disk2", is assigned to a *VirtualDiskServer* process, then the *VirtualDiskServer* process must reside on the PC where the local NTFS directory exists. On the other hand, if the assigned directory is a network directory specified as a UNC name (NFS equivalent), e.g. "\\FileServer\PsPs\Disk2", then the *VirtualDiskServer* process may reside anywhere in the network.
- Since several virtual disks can be assigned to a same *VirtualDiskServer* process and accessing a physical disk is more efficient through its local NTFS directory, e.g. "C:\PsPs\Disk2", than through its network directory, e.g. "\\FileServer\PsPs\Disk2", each SP node should run as many *VirtualDiskServer* processes as processors (independently of the number of disks hooked on the SP node). For example, if 15 disks are hooked on a Bi-Pentium PC, two *VirtualDiskServer* processes are enough to handle I/O accesses on these 15 disks.

5.8.1 CAP specification of the PS^2 server

Program 5-6 shows the CAP declaration of the parallel storage and processing server (*Ps2ServerT*) hierarchical process (lines 1-40). At line 42, the *Ps2ServerT* process declaration is instantiated giving a *Ps2Server* hierarchical process. The *Ps2Server* hierarchical process is composed of one *InterfaceServer* thread (line 4), several *VirtualDiskServer* hierarchical processes (line 5), and several *ComputeServer* threads (line 6). As already mentioned in Section 5.8, the *InterfaceServer* thread coordinates parallel file directory operations (open parallel file, close parallel file, create parallel file, delete parallel file, create directory, delete directory, and list directory) and resolves race conditions on parallel files when the *Ps2Server* hierarchical process is faced with simultaneous parallel file directory access requests. The *VirtualDiskServer* hierarchical processes offer extent file directory operations (open extent file, close extent file, create extent file, delete extent file, create extent file directory, delete extent file directory, and list extent file directory) and extent access operations (read extent, write extent, and delete extent) on extent files stored on virtual disks. The *ComputeServer* threads are fully customizable by application or library programmers for offering processing operations to be combined with extent access operations.

```
 1 process Ps2ServerT
 2 {
 3 subprocesses:
 4   InterfaceServerT InterfaceServer;
 5   VirtualDiskServerT VirtualDiskServer[];
 6   ComputeServerT ComputeServer[];
 7
 8 operations:
 9   LocateParallelFile
10     in LocateParallelFileIT* InputP
11     out LocateParallelFileOT* OutputP;
12
13   OpenParallelFile
14     in OpenParallelFileIT* InputP
15     out OpenParallelFileOT* OutputP;
16
17   CloseParallelFile
18     in CloseParallelFileIT* InputP
19     out CloseParallelFileOT* OutputP;
20
21   CreateParallelFile
22     in CreateParallelFileIT* InputP
23     out CreateParallelFileOT* OutputP;
24
25   DeleteParallelFile
26     in ps2DeleteParallelFileIT* InputP
27     out ps2DeleteParallelFileOT* OutputP;
28
29   CreateDirectory
30     in CreateDirectoryIT* InputP
31     out CreateDirectoryOT* OutputP;
32
33   DeleteDirectory
34     in DeleteDirectoryIT* InputP
35     out DeleteDirectoryOT* OutputP;
36
37   ListDirectory
38     in ListDirectoryIT* InputP
39     out ListDirectoryOT* OutputP;
40 }; // end process ps2ServerT
41
42 Ps2ServerT Ps2Server;
```

Program 5-6. CAP specification of the PS^2 server (detailed version)

The *Ps2Server* hierarchical process offers 8 predefined parallel operations for manipulating parallel files (lines 9-27) and directories (lines 29-39). The declaration of the input tokens (those ending with *IT*), i.e. the input of operations, and the declaration of the output tokens (those ending with *OT*), i.e. the output of operations, are shown in Program 5-7. Each output tokens contains an error code reporting possible errors, e.g. parallel file not found.

A brief description of each parallel operation offered by the *Ps2Server* hierarchical process (Program 5-6) is given below (line numbers refer to Program 5-7):

- ***Ps2ServerT::LocateParallelFile***
  The *Ps2ServerT::LocateParallelFile* operation locates the extent files making up a parallel file, i.e. locates on which virtual disks the extent files reside. The input of the operation is the path name of the parallel file (line 3). The output is the virtual disk indices where the extent files reside (line 8), i.e. *VirtualDiskIndex[k]* index corresponds to the index of the virtual disk where the kth extent file (i.e. whose extent file index is k) resides. This parallel operation is internally used by the *Ps2ServerT::OpenParallelFile* and *Ps2ServerT::DeleteParallelFile* operations for locating extent files before opening or deleting them.
- ***Ps2ServerT::OpenParallelFile***
  The *Ps2ServerT::OpenParallelFile* operation opens a parallel file, i.e. opens all the extent files making up the parallel file. Once a parallel file is open one can read from, write to, or delete from the extent files extents using the extent access operations offered by the *VirtualDiskServer[VirtualDiskServerIndex].ExtentServer* threads (Program 5-14). The input of the operation is the path name of the parallel file (line 14) and the opening flags (line 15). The opening flags specify if the parallel file is open for only reading extents, only writing extents, or both, and specify also the share modes, i.e. if subsequent opens can be performed on the parallel file for read or write access. The output of the operation is a parallel file descriptor (line 20), the virtual disk indices where the extent files reside (line 21), and the extent file descriptors (line 22) so that *application threads* may directly access the *VirtualDiskServer[VirtualDiskServerIndex].ExtentServer* threads for reading, writing, or deleting extents.
- ***Ps2ServerT::CloseParallelFile***
  The *Ps2ServerT::CloseParallelFile* operation closes an open parallel file, i.e. closes all the extent files making up the parallel file. The input of the operation is the parallel file descriptor (line 28) and the output is merely the error code (line 33). Once a parallel file is closed, its descriptor becomes invalid.
- ***Ps2ServerT::CreateParallelFile***
  The *Ps2ServerT::CreateParallelFile* operation creates a parallel file, i.e. creates all the extent files making up the parallel file. The input of the operation is the path name of the parallel file (line 38) and the virtual disk

```
 1 token LocateParallelFileIT
 2 {
 3   PathNameT PathName;
 4 }; // end LocateParallelFileIT
 5
 6 token LocateParallelFileOT
 7 {
 8   ArrayT<int> VirtualDiskIndex;
 9   int ErrorCode;
10 }; // end LocateParallelFileOT
11 -----------------------------------------------
12 token OpenParallelFileIT
13 {
14   PathNameT PathName;
15   int OpeningFlags;
16 }; // end token OpenParallelFileIT
17
18 token OpenParallelFileOT
19 {
20   int ParallelFileDescriptor;
21   ArrayT<int> VirtualDiskIndex;
22   ArrayT<int> ExtentFileDescriptor;
23   int ErrorCode;
24 }; // end token OpenParallelFileOT
25 -----------------------------------------------
26 token CloseParallelFileIT
27 {
28   int ParallelFileDescriptor;
29 }; // end token CloseParallelFileIT
30
31 token CloseParallelFileOT
32 {
33   int ErrorCode;
34 }; // end token CloseParallelFileOT
35 -----------------------------------------------
36 token CreateParallelFileIT
37 {
38   PathNameT PathName;
39   ArrayT<int> VirtualDiskIndex;
40 }; // end token CreateParallelFileIT
41
42 token CreateParallelFileOT
43 {
44   int ErrorCode;
45 }; // end token CreateParallelFileOT
46 -----------------------------------------------
47 token DeleteParallelFileIT
48 {
49   PathNameT PathName;
50 }; // end token DeleteParallelFileIT
51
52 token DeleteParallelFileOT
53 {
54   int ErrorCode;
55 }; // end token DeleteParallelFileOT
56 -----------------------------------------------
57 token CreateDirectoryIT
58 {
59   PathNameT PathName;
60 }; // end token CreateDirectoryIT
61
62 token CreateDirectoryOT
63 {
64   int ErrorCode;
65 }; // end token CreateDirectoryOT
66 -----------------------------------------------
67 token DeleteDirectoryIT
68 {
69   PathNameT PathName;
70 }; // end token DeleteDirectoryIT
71
72 token DeleteDirectoryOT
73 {
74   int ErrorCode;
75 }; // end token DeleteDirectoryOT
76 -----------------------------------------------
77 token ListDirectoryIT
78 {
79   PathNameT PathName;
80 }; // end token ListDirectoryIT
81
82 token ListDirectoryOT
83 {
84   ArrayT<DirEntryT> List;
85   int ErrorCode;
86 }; // end token ListDirectoryOT
```

Program 5-7. CAP specification of the PS^2 server tokens

indices where to create the extent files (line 39). PS^2 cannot create more than one extent file per virtual disk, e.g. VirtualDiskIndex[3] = 8 and VirtualDiskIndex[6] = 8, meaning that the 3rd and the 6th extent files are to be created on the 8th virtual disk, is illegal. The output of the operation is merely the error code (line 44).

- ***Ps2ServerT::DeleteParallelFile***
  The *Ps2ServerT::DeleteParallelFile* operation deletes a parallel file, i.e. deletes all the extent files making up the parallel file. The input of the operation is the path name of the parallel file (line 49) and the output is merely the error code (line 54).
- ***Ps2ServerT::CreateDirectory***
  The *Ps2ServerT::CreateDirectory* operation creates a directory, i.e. creates an extent file directory on each virtual disk. The input of the operation is the path name of the directory (line 59) and the output is merely the error code (line 74).
- ***Ps2ServerT::DeleteDirectory***
  The *Ps2ServerT::DeleteDirectory* operation deletes a directory, i.e. deletes all the extent file directory on each virtual disk. The input of the operation is the path name of the directory (line 69) and the output is merely the error code (line 74). A directory can be deleted only if all the extent file directories are empty.
- ***Ps2ServerT::ListDirectory***
  The *Ps2ServerT::ListDirectory* operation lists the contents of a directory, i.e. lists the contents of the extent file directory on each virtual disk and checks the coherence of the parallel files, i.e. for each parallel file PS^2 ensures that all the extent files making up the parallel file are present and coherent. The input of the operation is the path name of the directory (line 79) and the output is the contents of the directory (line 84). A *DirEntryT* structure (line 84) contains a path name and a flag indicating whether the path name refers to a parallel file or a directory.

5.8.2 CAP specification of the PS^2 interface server

Program 5-8 shows the CAP declaration of the *InterfaceServerT* thread (lines 12-20). For the sake of simplicity, sequential operations (line 19) are not shown. As shown in Program 5-6, the *Ps2Server* hierarchical process contains one *InterfaceServer* thread (line 4) coordinating the parallel file directory operations and maintaining the consistency of the virtual disks. Since all parallel file directory operations are handled by the *InterfaceServer* thread, it forms a single access point for all these operations. This might become a congestion point limiting the scalability of PS^2 when increasing the number *application threads* intensively doing parallel file directory operations. However, PS^2 has been designed assuming that parallel file opening and creation operations are rare. PS^2 is a library of reusable low-level parallel file system components linked within an application comprising several *application threads*, but not thousands, mostly doing parallel computations and parallel I/O's directly with the *ComputeServer* threads and the *ExtentServer* threads without involving the *InterfaceServer* thread. Therefore, within that context, having a single access point for parallel file directory operations, i.e. a single *InterfaceServer* thread, should not degrade performances.

```
 1 const int PARALLEL_FILE_TABLE_SIZE = 500;
 2
 3 struct ParallelFileT
 4 {
 5   PathNameT PathName;
 6   int OpeningFlags;
 7   ArrayT<int> VirtualDiskIndex;
 8   ArrayT<int> ExtentFileDescriptor;
 9   int ReferenceCount;
10 }; // end struct ParallelFileT
11
12 process InterfaceServerT
13 {
14 variables:
15   ConcurrentAccessResolverT ConcurrentAccessResolver;
16   ParallelFileT* ParallelFileTable[PARALLEL_FILE_TABLE_SIZE];
17
18 operations:
19   // ...
20 }; // end process InterfaceServerT
```

Program 5-8. CAP specification of the PS^2 interface server

The *InterfaceServer* thread (its declaration is shown in Program 5-8) maintains a table of all open parallel files (line 16). Each time a parallel file is opened, the *InterfaceServer* thread first checks whether that parallel file is already open or not so as to verify if the opening flags (line 6) allow subsequent opening operations. The reference count value (line 9) indicates how many times a parallel file has been opened, i.e. how many parallel file descriptors refer to that entry.

In order to maximize the utilization of the *ExtentFileServer* threads, the *InterfaceServer* thread is designed so as to be able to execute simultaneously several parallel file directory operations, e.g. to create a parallel file (consisting of creating the extent files), to delete a directory (consisting of deleting the extent file directories), and to open a parallel file (consisting of opening the extent files). However, there are race conditions that must be resolved in order to keep stored parallel files consistent and PS^2 coherent. For example, a first *application thread* opens the "/image.ps2" parallel file and simultaneously a second *application thread* deletes that parallel file. In order to avoid deleting an open parallel file leading to an incoherence, the *InterfaceServer* thread detects race conditions and serializes the offending parallel file directory operations, i.e. executes one offending operation after the other is completed.

Program 5-9 shows the declaration of the *ConcurrentAccessResolverT* class resolving race conditions on parallel files and directories. The *InterfaceServer* thread contains such an object (Program 5-8, line 15) for serializing conflicting operations. Each time an operation on a parallel file is started, e.g. the *Ps2ServerT::OpenParallelFile* operation, the parallel file is locked using the *LockParallelFile* method (line 4). Once the operation is completed, the parallel file is unlocked using the *UnlockParallelFile* method (line 5). Each time an operation on a directory is started, e.g. the *Ps2ServerT::CreateDirectory* operation, the directory is locked using the *LockDirectory* method (line 6). Once the operation is completed the directory is unlocked using the *UnlockDirectory* method (line 7). The return boolean values (lines 4 and 6) indicate whether the parallel file or the directory has been successfully locked or not.

```
1 class ConcurrentAccessResolverT
2 {
3 public:
4   bool LockParallelFile(const PathNameT& pathName, capTokenT* tokenP);
5   void UnlockParallelFile(const PathNameT& pathName);
6   bool LockDirectory(const PathNameT& pathName, capTokenT* tokenP);
7   void UnlockDirectory(const PathNameT& pathName);
8 }; // end class ConcurrentAccessResolverT
```

Program 5-9. Concurrent accesses to a same directory or a same parallel file are resolved using a *ConcurrentAccessResolverT* object

A parallel file can be locked if it is unlocked and the directory where it resides is also unlocked. A directory can be locked if it is unlocked, the parent directory is unlocked, and all files (parallel files and directories) contained in that directory are also unlocked. If a parallel file or a directory cannot be locked, i.e. there are conflicting files that are locked, the *capDoNotCallSuccessor* CAP-library function (Section 3.7.1) is applied on the token of the offending operation (lines 4 and 6, second argument) to temporarily suspend its execution. Once the conflict is resolved, i.e. the last conflicting file (parallel file or directory) is unlocked, the suspended operation is resumed by calling the *capCallSuccessor* CAP-library function (Section 3.7.1) on its token. This strategy ensures that once a parallel file or a directory is locked, the *InterfaceServer* thread can safely perform the required parallel file directory operation on that file without having in parallel another parallel file directory operation on the same parallel file or on the same directory possibly leading to an incoherence within PS^2.

5.8.3 CAP specification of the PS^2 compute server

Program 5-10 shows the declaration of the *ComputeServerT* thread. A *ComputeServerT* thread does not offer any predefined sequential operations. Application or library developers are free to extend the functionalities of PS^2, i.e. to customize PS^2, by adding library-specific or application-specific processing operations on *ComputeServer* threads contained in the *Ps2Server* hierarchical process (Program 5-6, line 6). Thanks to the declaration of additional operations (Program 3-7), these added application-specific processing operations are declared and defined in separate application-specific CAP source files.

```
1 process ComputeServerT
2 {
3 operations: // must remain empty
4 }; // end process ComputeServerT
```

Program 5-10. CAP specification of the PS^2 computer server

5.8.4 CAP specification of the PS^2 virtual disk server

Program 5-11 shows the CAP declaration of the *VirtualDiskServerT* hierarchical process which comprises an *ExtentFileServer* thread (line 15) and an *ExtentServer* thread (line 16). A *VirtualDiskServerT* hierarchical process does not offer any parallel operations. It is merely used for declaring variables shared among the *ExtentFileServer* thread and the *ExtentServer* thread (line 12). Thanks to the asynchronous extent access operations of EFS (Section 5.10), a same *VirtualDiskServerT* hierarchical process can serve in parallel several virtual disks. For each served virtual disk, a *VirtualDiskT* structure (lines 3-7) is created comprising an EFS object (line 5) and a table of open extent files.

```
 1 const int EXTENT_FILE_TABLE_SIZE = 500;
 2
 3 struct VirtualDiskT
 4 {
 5   ExtentFileSystemT ExtentFileSystem;
 6   ExtentFileT* ExtentFileTable[EXTENT_FILE_TABLE_SIZE];
 7 }; // end struct VirtualDiskT
 8
 9 process VirtualDiskServerT
10 {
11 variables:
12   VirtualDiskT VirtualDisk[];
13
14 subprocesses:
15   ExtentFileServerT ExtentFileServer;
16   ExtentServerT ExtentServer;
17
18 operations:
19 }; // end VirtualDiskServerT
```

Program 5-11. CAP specification of the PS^2 virtual disk server

5.8.5 CAP specification of the PS^2 extent file server

Program 5-12 shows the CAP declaration of the *ExtentFileServerT* thread offering extent file directory operations (open extent file, close extent file, create extent file, delete extent file, create extent file directory, delete extent file directory, list extent file directory). Each of these extent file directory operations are implemented using the extent file system described in Section 5.10. An *ExtentFileServerT* thread offers 10 predefined operations for manipulating extent files (lines 12-26 and 40-42), extent file directories (lines 28-38), and extent file systems (lines 4-10). As already mentioned in Section 5.8, all these extent file directory operations are synchronous, i.e. an *ExtentFileServerT* thread executes one operation at a time. These 10 extent file directory operations are internally used by the *InterfaceServer* thread when executing parallel file directory operations. Under no circumstances should programmers use the extent file directory operations offered by an *VirtualDiskServer[VirtualDiskServerIndex].ExtentFileServer* thread contained in the *Ps2Server* hierarchical process hierarchy (Program 5-6, line 5).

The declaration of the input tokens (those ending with *IT*), i.e. the input of operations, and the declaration of the output tokens (those ending with *OT*), i.e. the output of operations, are shown in Program 5-13. Since an *ExtentFileServer* thread can serve several virtual disks, each input token contains the index of the virtual disk (*VirtualDiskIndex*) to which the extent file directory operation refers (e.g. line 3). This *VirtualDiskIndex* value is an index within the table of virtual disks of the *VirtualDiskServer* hierarchical process (Program 5-11, line 12). Each output token contains an error code reporting possible errors, e.g. extent file not found.

```
 1 process ExtentFileServerT
 2 {
 3 operations:
 4   MountExtentFileSystem
 5     in MountExtentFileSystemIT* InputP
 6     out MountExtentFileSystemOT* OutputP;
 7
 8   UnmountExtentFileSystem
 9     in UnmountExtentFileSystemIT* InputP
10     out UnmountExtentFileSystemOT* OutputP;
11
12   OpenExtentFile
13     in OpenExtentFileIT* InputP
14     out OpenExtentFileOT* OutputP;
15
16   CloseExtentFile
17     in CloseExtentFileIT* InputP
18     out CloseExtentFileOT* OutputP;
19
20   CreateExtentFile
21     in CreateExtentFileIT* InputP
22     out CreateExtentFileOT* OutputP;
23
24   DeleteExtentFile
25     in DeleteExtentFileIT* InputP
26     out DeleteExtentFileOT* OutputP;
27
28   CreateExtentFileDirectory
29     in CreateExtentFileDirectoryIT* InputP
30     out CreateExtentFileDirectoryOT* OutputP;
31
32   DeleteExtentFileDirectory
33     in DeleteExtentFileDirectoryIT* InputP
34     out DeleteExtentFileDirectoryOT* OutputP;
35
36   ListExtentFileDirectory
37     in ListExtentFileDirectoryIT* InputP
38     out ListExtentFileDirectoryOT* OutputP;
39
40   LocateExtentFile
41     in LocateExtentFileIT* InputP
42     out LocateExtentFileOT* OutputP;
43 }; // end process ExtentFileServerT
```

Program 5-12. CAP specification of the PS^2 extent file server

A brief description of each sequential operation offered by an *ExtentFileServerT* thread (Program 5-12) is given below (line numbers refer to Program 5-13):

- ***ExtentFileServerT::MountExtentFileSystem***
  The *ExtentFileServerT::MountExtentFileSystem* operation mounts an NTFS directory onto a virtual disk, i.e. this NTFS directory, either a local NTFS directory, e.g. "C:\PsPs\Disk2" or a network NTFS directory, e.g. "\\FileServer\PsPs\Disk2", becomes the root of the virtual disk. Subsequent operations on that virtual disk are relative to that NTFS directory. For example, creating the extent file "/Archive/Old/Sun-Star.ps2" (Figure 5-5) creates an NTFS file starting from the root directory, e.g. "C:\PsPs\Disk2\Archive\Old\Sun-Star.ps2" or "\\FileServer\PsPs\Disk2\Archive\Old\Sun-Star.ps2". This operation must be called once before any other operations. The input of the operation is the path name of the root directory (line 4) to mount and a boolean value indicating if its a read only virtual disk (line 5). The output of the operation is merely the error code (line 10).
- ***ExtentFileServerT::UnmountExtentFileSystem***
  The *ExtentFileServerT::UnmountExtentFileSystem* operation unmounts the NTFS root directory from a virtual disk. Before calling this operation, all extent files must be closed. The input of the operation is the index of a mounted virtual disk (line 15) and the output is merely the error code (line 20).
- ***ExtentFileServerT::OpenExtentFile***
  The *ExtentFileServerT::OpenExtentFile* operation opens an extent file. The open extent file object (*ExtentFileT* class) is kept in the extent file table (Program 5-11, line 6) of the virtual disk so that subsequent operations may refer to that extent file. The input of the operation is the path name of the extent file (line 26) and the opening flags (line 27) specifying whether to open the extent file for only reading extents, only

writing extents, or both. Moreover, there is a flag for disabling the EFS extent cache for this particular extent file, i.e. subsequent read and write extent operations to that extent file will go directly to the disk without visiting the extent cache. The output of the operation is the extent file descriptor which is an index in the extent file table of the virtual disk (Program 5-11, line 6).

- ***ExtentFileServerT::CloseExtentFile***
  The *ExtentFileServerT::CloseExtentFile* operation closes an extent file. Before closing an extent files, all operations on that extent file, e.g. *ExtentServerT::ReadExtent*, must be completed. The input of the operation is the extent file descriptor (line 39) and the output is merely the error code (line 44). Once an extent file is closed, its descriptor becomes invalid and can be reallocated to another extent file.
- ***ExtentFileServerT::CreateExtentFile***
  The *ExtentFileServerT::CreateExtentFile* operation creates an extent file. The input of the operation is the path name of the extent file (line 50), its extent file index (remember that all extent files making up a parallel file are numbered from zero, line 51) and the striping factor of the parallel file (line 52). The stored extent file index is used when the *InterfaceServer* thread locates the extent files making up a parallel file (*Ps2ServerT::LocateParallelFile* operation). The output of the operation is merely the error code (line 57).
- ***ExtentFileServerT::DeleteExtentFile***
  The *ExtentFileServerT::DeleteExtentFile* operation deletes an extent file from a virtual disk. The input of the operation is the path name of the extent file (line 63) and the output is merely the error code (line 68).
- ***ExtentFileServerT::CreateExtentFileDirectory***
  The *ExtentFileServerT::CreateExtentFileDirectory* operation creates an extent file directory on the virtual disk. The input of the operation is the path name of the extent file directory (line 74) and the output is merely the error code (line 79).
- ***ExtentFileServerT::DeleteExtentFileDirectory***
  The *ExtentFileServerT::DeleteExtentFileDirectory* operation deletes an extent file directory from the virtual disk. The input of the operation is the path name of the extent file directory (line 85) and the output is merely the error code (line 90).
- ***ExtentFileServerT::ListExtentFileDirectory***
  The *ExtentFileServerT::ListExtentFileDirectory* operation lists the contents of an extent file directory. The input of the operation is the path name of the extent file directory (line 96) and the output is the contents of the directory (line 101). An *ExtentDirEntryT* structure contains a path name, a flag indicating whether the path name refers to an extent file or an extent file directory, and in case of an extent file its extent file index and the striping factor of the parallel file.
- ***ExtentFileServerT::LocateExtentFile***
  The *ExtentFileServerT::LocateExtentFile* operation is used to retrieve the extent file index of an extent file. This operation is used by the *InterfaceServer* thread for locating the extent files making up a parallel file (*Ps2ServerT::LocateParallelFile* operation). The input of the operation is the path name of the extent file (line 108) and the output is its extent file index (line 113) and the striping factor of the parallel file (line 114).

5.8.6 CAP specification of the PS^2 extent server

Program 5-14 shows the CAP declaration of the *ExtentServerT* thread offering 3 operations for reading from, writing to, and deleting from extent files extents. These reusable low-level extent access operations can be combined with application-specific or library-specific processing operations offered by the *ComputeServer* threads (Program 5-6, line 6). All these extent access operations are asynchronous, i.e. an *ExtentServer* thread performs extent accesses asynchronously without blocking. This enables a single *ExtentServer* thread to read in parallel from, or write in parallel to several extent files located on different virtual disks.

The declaration of the input tokens (those ending with *IT*), i.e. the input of operations, and the declaration of the output tokens (those ending with *OT*), i.e. the output of operations, are shown in Program 5-15. Since an *ExtentServerT* thread can serve several virtual disks, each input token contains the index of the virtual disk (*VirtualDiskIndex*) to which the extent access operation refers (e.g. line 3). This *VirtualDiskIndex* value is an index within the table of virtual disks of the *DiskServerT* process (Program 5-11, line 12). Each output token contains an error code reporting possible errors, e.g. virtual disk full.

A brief description of each operation offered by an *ExtentServerT* thread (Program 5-14) is given below (line numbers refer to Program 5-15):

- ***ExtentServerT::ReadExtent***
  The *ExtentServerT::ReadExtent* operation reads an extent from an extent file. If the extent has not been previously written, an empty extent is returned (it is not an error). The input of the operation is the descriptor

```
token MountExtentFileSystemIT
{
  int VirtualDiskIndex;
  PathNameT RootPathName;
  bool ReadOnlyExtentFileSystem;
}; // end token MountExtentFileSystemIT

token MountExtentFileSystemOT
{
  int ErrorCode;
}; // end token MountExtentFileSystemOT
-----------------------------------------------------
token UnmountExtentFileSystemIT
{
  int VirtualDiskIndex;
}; // end token UnmountExtentFileSystemIT

token UnmountExtentFileSystemOT
{
  int ErrorCode;
}; // end token UnmountExtentFileSystemOT
-----------------------------------------------------
token OpenExtentFileIT
{
  int VirtualDiskIndex;
  PathNameT PathName;
  int OpeningFlags;
}; // end token OpenExtentFileIT

token OpenExtentFileOT
{
  int ExtentFileDescriptor;
  int ErrorCode;
}; // end token OpenExtentFileOT
-----------------------------------------------------
token CloseExtentFileIT
{
  int VirtualDiskIndex;
  int ExtentFileDescriptor;
}; // end token CloseExtentFileIT

token CloseExtentFileOT
{
  int ErrorCode;
}; // end token CloseExtentFileOT
-----------------------------------------------------
token CreateExtentFileIT
{
  int VirtualDiskIndex;
  PathNameT PathName;
  int ExtentFileIndex;
  int StripingFactor;
}; // end token CreateExtentFileIT

token CreateExtentFileOT
{
  int ErrorCode;
}; // end token CreateExtentFileOT
-----------------------------------------------------
token DeleteExtentFileIT
{
  int VirtualDiskIndex;
  PathNameT PathName;
}; // end token DeleteExtentFileIT

token DeleteExtentFileOT
{
  int ErrorCode;
}; // end token DeleteExtentFileOT
-----------------------------------------------------
token CreateExtentFileDirectoryIT
{
  int VirtualDiskIndex;
  PathNameT PathName;
}; // end token CreateExtentFileDirectoryIT

token CreateExtentFileDirectoryOT
{
  int ErrorCode;
}; // end token CreateExtentFileDirectoryOT
-----------------------------------------------------
token DeleteExtentFileDirectoryIT
{
  int VirtualDiskIndex;
  PathNameT PathName;
}; // end token DeleteExtentFileDirectoryIT

token DeleteExtentFileDirectoryOT
{
  int ErrorCode;
}; // end token DeleteExtentFileDirectoryOT
-----------------------------------------------------
token ListExtentFileDirectoryIT
{
  int VirtualDiskIndex;
  PathNameT PathName;
}; // end token ListExtentFileDirectoryIT

token ListExtentFileDirectoryOT
{
  ArrayT<ExtentDirEntryT> List;
  int ErrorCode;
}; // end token ListExtentFileDirectoryOT
-----------------------------------------------------
token LocateExtentFileIT
{
  int VirtualDiskIndex;
  PathNameT PathName;
}; // end token LocateExtentFileIT

token LocateExtentFileOT
{
  int ExtentFileIndex;
  int StripingFactor;
  int ErrorCode;
}; // end token LocateExtentFileIT
```

Program 5-13. CAP specification of the PS^2 extent file server tokens

```
process ExtentServerT
{
operations:
  ReadExtent
    in ReadExtentIT* InputP
    out ReadExtentOT* OutputP;

  WriteExtent
    in WriteExtentIT* InputP
    out WriteExtentOT* OutputP;

  DeleteExtent
    in DeleteExtentIT* InputP
    out DeleteExtentOT* OutputP;
}; // end process ExtentServerT
```

Program 5-14. CAP specification of the PS^2 extent server

of the extent file (line 4) and the local extent index (line 5), i.e. which extent has to be read? The output of the operation is the extent (line 10) comprising a buffer with the data (header and body), the size of the header in bytes, and the size of the body in bytes.

- ***ExtentServerT::WriteExtent***
  The *ExtentServerT::WriteExtent* operation writes an extent to an extent file. If the extent exists, i.e. was previously written, its contents is destroyed. The input of the operation is the descriptor of the extent file (line 17), the local extent index (line 18), i.e. which extent is written, and the extent (line 19). The output is merely the error code (line 24).
- ***ExtentServerT::DeleteExtent***
  The *ExtentServerT::DeleteExtent* operation deletes an extent from an extent file. If the extent does not exist, i.e. has not been previously written, the operation does nothing and returns successfully. The input of the operation is the descriptor of the extent file (line 30) and the index of the extent to delete (line 31). The output is merely the error code (line 36).

```
 1 token ReadExtentIT
 2 {
 3   int VirtualDiskIndex;
 4   int ExtentFileDescriptor;
 5   int ExtentIndex;
 6 }; // end token ReadExtentIT
 7
 8 token ReadExtentOT
 9 {
10   ExtentT Extent;
11   int ErrorCode;
12 }; // end token ReadExtentOT
13 -------------------------------------------------
14 token WriteExtentIT
15 {
16   int VirtualDiskIndex;
17   int ExtentFileDescriptor;
18   int ExtentIndex;
19   ExtentT Extent;
20 }; // end token WriteExtentIT
21
22 token WriteExtentOT
23 {
24   int ErrorCode;
25 }; // end token WriteExtentOT
26 -------------------------------------------------
27 token DeleteExtentIT
28 {
29   int VirtualDiskIndex;
30   int ExtentFileDescriptor;
31   int ExtentIndex;
32 }; // end token DeleteExtentIT
33
34 token DeleteExtentOT
35 {
36   int ErrorCode;
37 }; // end token DeleteExtentOT
```

Program 5-15. CAP specification of the PS^2 extent server tokens

5.9 CAP-based synthesis of the parallel file directory operations

This section describes how the parallel file directory operations offered by the PS^2 server (*Ps2ServerT* hierarchical process, Program 5-6) are synthesized using the CAP computer-aided parallelization tool. For the sake of simplicity, only the parallel *Ps2ServerT::OpenParallelFile* operation is shown. Other parallel file directory operations are similar.

Figure 5-9 shows the graphical CAP specification of the parallel *Ps2ServerT::OpenParallelFile* operation. The input of the macro-dataflow graph is an *OpenParallelFileIT* token comprising the name of the parallel file to open and the opening flags. The input token is first redirected to the *InterfaceServerT::Stage1* sequential operation performed by the *InterfaceServer* thread who locks the parallel file using the *ConcurrentAccessResolver* object (Program 5-8, line 15). If the parallel file is already locked, then the *capDoNotCallSuccessor* CAP-library function is applied on the token suspending the execution of the *Ps2ServerT::OpenParallelFile* operation until the parallel file is unlocked. If there is no error, the *InterfaceServerT::Stage1*'s output token is redirected to the *InterfaceServerT::Stage2* sequential operation also performed by the *InterfaceServer* thread. In *InterfaceServerT::Stage2*, the parallel file table (Program 5-8, line 16) is browsed in order to check whether the parallel file is already open or not. If the parallel file is already open, the *InterfaceServer* thread verifies that successive opens are allowed, i.e. the parallel file was first open with the appropriate sharing flags. If the parallel file is not already open, the *InterfaceServer* thread allocates a new entry within the parallel file table, and the output token is redirected to the *Ps2ServerT::LocateParallelFile* parallel operation. The parallel *Ps2ServerT::LocateParallelFile* operation is incorporated into the schedule using the design pattern shown in Section 3.8.5 enabling to transfer essential data through the incorporated operation. The parallel *Ps2ServerT::LocateParallelFile* operation locates the extent files making up the parallel file, and if there is no error, the extent files are then opened in parallel using an *indexed parallel* CAP construct. The *indexed parallel* construct's input token is divided by a split function into *OpenExtentFileIT* subtokens (Program 5-13), containing the name of each of the extent files making up the parallel file. The *OpenExtentFileIT* subtokens are routed to the appropriate *ExtentFileServer* threads, which open the extent files. The *ExtentFileServerT::OpenExtentFile* sequential operation performed by each *ExtentFileServer* threads returns an *OpenExtentFileOT* token comprising an extent file descriptor which is merged into the *indexed parallel* construct's output token. Once all the extent file descriptors are merged, the successor is called. If one of the extent files has not been successfully opened (partial open), then, for maintaining the parallel file system consistency, all successfully open extent files are closed using a same *indexed parallel* CAP construct. Finally, the *InterfaceServerT::Stage3* sequential operation performed by the *InterfaceServer* thread unlocks the parallel file (Program 5-9), which may resume another parallel file directory operation waiting for this parallel file to become unlocked (by calling the *capCallSuccessor* CAP-library function).

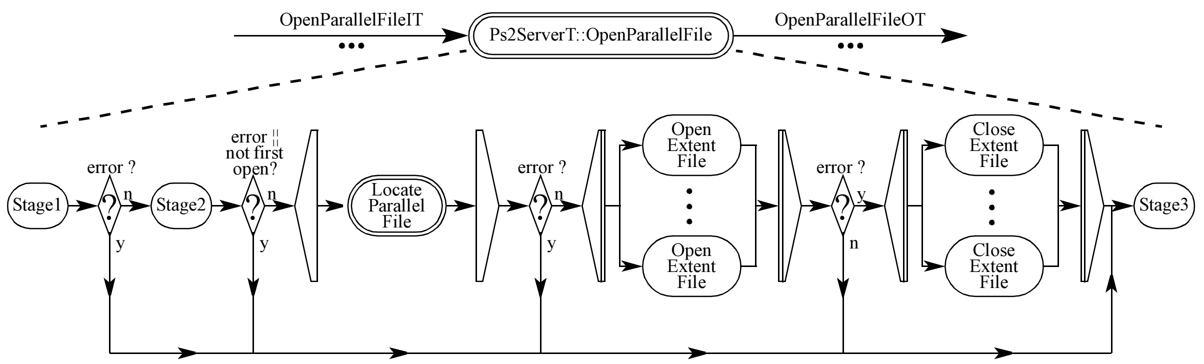


Figure 5-9. Graphical CAP specification of the parallel *Ps2ServerT::OpenParallelFile* operation

Program 5-16 shows the CAP specification of the parallel *ps2Server::OpenParallelFile* operation corresponding to the DAG in Figure 5-9. Split and merge functions are not shown for the sake of simplicity.

```
 1 operation Ps2ServerT::OpenParallelFile
 2   in OpenParallelFileIT* InputP
 3   out OpenParallelFileOT* OutputP
 4 {
 5   InterfaceServer.Stage1 // Locks the parallel file
 6   >->
 7   if ( thisTokenP->ErrorCode == 0 )
 8   (
 9     InterfaceServer.Stage2 // Parallel file is already open ?
10     >->
11     if ( (thisTokenP->ErrorCode == 0) && (thisTokenP->FirstOpen) )
12     ( // Lines 13-20: design pattern for incorporating an operation into the schedule
13       parallel (InterfaceServer, remote OpenParallelFileIOT Result1(*thisTokenP))
14       (
15         (
16           GenerateLocateParallelFileRequest,  // Split routine
17           LocateParallelFile,  // Incorporated operation
18           MergeLocateParallelFileAnswer  // Merge routine
19         ) // end parallel branch
20       ) // end parallel
21       >->
22       if ( thisTokenP->Error == 0 )
23       (
24         indexed
25           (int ExtentFileIndex = 0; ExtentFileIndex < thisTokenP->VirtualDiskIndex.Size(); ExtentFileIndex++)
26         parallel
27           (
28             GenerateOpenExtentFileRequest, MergeOpenExtentFileAnswer,
29             InterfaceServer, remote OpenParallelFileIOT Result2(*thisTokenP)
30           )
31         (
32           VirtualDiskServer[
33                             ToVirtualDiskServerIndex(parallelInputTokenP->VirtualDiskIndex[ExtentFileIndex])
34                            ].ExtentFileServer.OpenExtentFile
35         ) // end indexed parallel
36         >->
37         if ( thisTokenP->Error != 0 )
38         (
39           indexed
40             (int ExtentFileIndex = 0; ExtentFileIndex < thisTokenP->VirtualDiskIndex.Size(); ExtentFileIndex++)
41           parallel
42             (
43               GenerateCloseExtentFileRequest, MergeCloseExtentFileAnswer,
44               InterfaceServer, remote OpenParallelFileIOT Result3(*thisTokenP)
45             )
46           (
47             VirtualDiskServer[
48                               ToVirtualDiskServerIndex(parallelInputTokenP->VirtualDiskIndex[ExtentFileIndex])
49                              ].ExtentFileServer.CloseExtentFile
50           ) // end indexed parallel
51         ) // end if
52       ) // end if
53     ) // end if
54   ) // end if
55   >->
56   InterfaceServer.Stage3; // Unlock the parallel file
57 } // end operation Ps2ServerT::OpenParallelFile
```

Program 5-16. CAP specification of the parallel *Ps2ServerT::OpenParallelFile* operation

The extent files are opened in parallel using an *indexed parallel* CAP construct (Program 5-16, lines 24-35). Open extent file requests are routed to the appropriate *ExtentFileServer* threads (lines 32-34), i.e. where the extent files reside, using the *ToVirtualDiskServerIndex* macro (Program 5-3) which converts a virtual disk index to a virtual disk server index (line 33). The index of the virtual disk where a particular extent file resides is retrieved using the table of virtual disk indices returned by the parallel *Ps2ServerT::LocateParallelFile* operation (Program 5-7, *LocateParallelFileOT* output token). This *VirtualDiskIndex* table converts an extent file index to a virtual disk index where this particular extent file resides, i.e. the *VirtualDiskIndex[k]* value corresponds to the index of the virtual disk where the (k+1)th extent file resides. The two mappings, from an extent file index to a virtual disk index, and from a virtual disk index to a virtual disk server index, enable a parallel file to be created on a subset of virtual disks and to mix up extent files making a parallel file amongst the virtual disks.

All other parallel file directory operations, e.g. *Ps2ServerT::CreateParallelFile*, *Ps2ServerT::CloseParallelFile*, etc, are designed using the same pattern. *ExtentFileServerT::CreateExtentFile*, *ExtentFileServerT::CloseExtentFile*, etc, performed by the *ExtentFileServer* threads are programmed using an *indexed parallel* CAP construct.

5.10 Design and implementation of a single-disk extent-oriented file system: the EFS extent file system

The extent file system is a single-disk extent-oriented file system. It provides a portable abstraction of extent files above a native file system, e.g. the Windows NT file system. EFS comprises a write-through cache of extents with a least recently used (LRU) replacement policy. For improved flexibility and customization, the extent cache may be disabled at each extent request, e.g. at each extent reading request.

Many single-disk file systems perform some kind of prefetching, i.e. data is read from the disk into the buffer cache before any process actually requests it. Prefetching is an attempt to reduce the latency of file access perceived by the process. EFS does not attempt to prefetch extents into the extent cache for two reasons. First, indiscriminate prefetching can cause thrashing in the extent cache [Nitzberg92]. Second, prefetching is based on the assumption that the system can intelligently guess what an application is going to request next. However, with parallel I/O- and compute- intensive operations developed with the CAP computer-aided parallelization tool and the PS^2 low-level parallel file system components, there is no need for EFS to make guesses about their behaviour. Thanks to the two split-merge CAP constructs (Sections 3.8.6 and 3.8.7) and to the pipelining inherent to CAP (Section 3.8.1), parallel operations automatically perform a sort of prefetching (Figure 5-10); the *ComputeServer* threads perform computation on extents while the *ExtentServer* threads read the next extents from the extent files (read pipeline). Moreover, with flow-controlled split-merge constructs (Section 3.10) and their filling factors, one can easily control the number of “prefetched” extents.

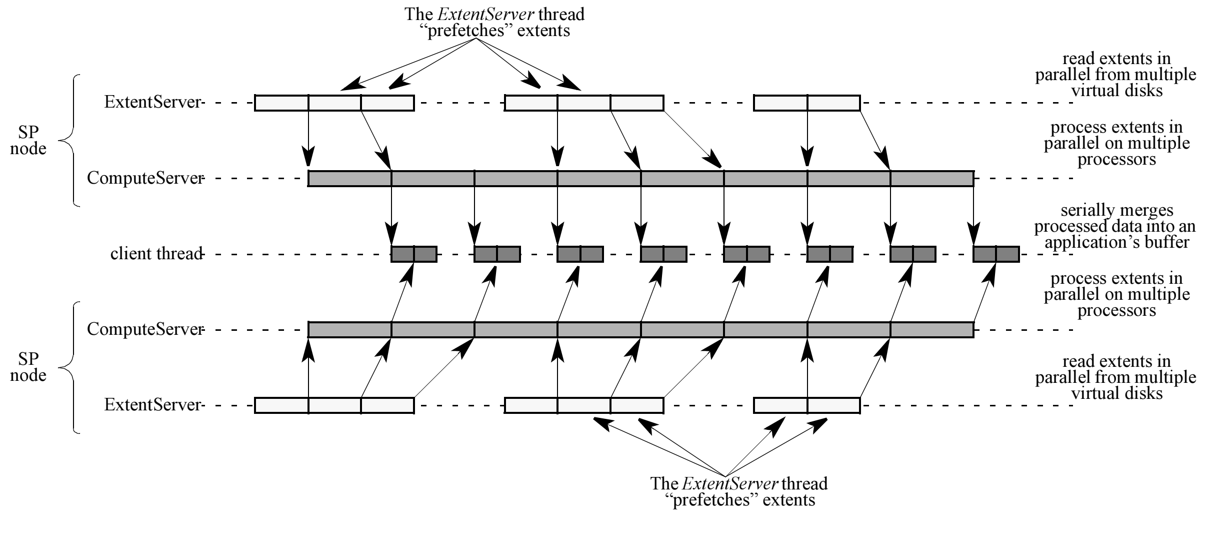

The *ExtentServer* thread “prefetches” extents
read extents in parallel from multiple virtual disks
SP node
ExtentServer
ComputeServer
process extents in parallel on multiple processors
client thread
serially merges processed data into an application’s buffer
ComputeServer
process extents in parallel on multiple processors
SP node
ExtentServer
read extents in parallel from multiple virtual disks
The *ExtentServer* thread “prefetches” extents

Figure 5-10. Thanks to the pipelined parallel execution of PS^2 operations, the *ExtentServer* threads are able to perform a sort of intelligent or informed extent prefetching from multiple virtual disks enabling a PS^2 operation to behave as if extents are read from memory with no delay (as long as the PS^2 server architecture contains enough disks to meet the I/O throughput requirement of the operation)

Program 5-17 shows the application programmer interface of EFS. The *ExtentFileSystemT* class implements all the methods manipulating extent files and extent file directories, e.g. open extent file, create extent file directory (lines 28-42). The *ExtentFileSystemT::OpenExtentFile* method (lines 31-32) returns an *ExtentFileT* extent file object from its path name. The *ExtentFileT* class implements all the extent access methods, i.e. read extent, write extent, and delete extent (lines 13-19). Each of these 3 methods is asynchronous, i.e. the I/O request is initiated and the method returns immediately. Once the I/O is completed, e.g. the extent is written, the completion routine is called. Each asynchronous I/O method (lines 16-18) features a single argument called the *IORequestP* packet comprising the pointer to the completion routine (line 8) and a flag (line 5) indicating whether to process this extent access request with the extent cache disabled or not.

```
 1 typedef void (*CompletionRoutineT) (void* IORequestP);
 2
 1 struct IORequestT
 2 {
 3 public:
 4   void* InfoP; // Private data for user
 5   bool ExtentCacheOn; // Enable the extent cache
 6   int ExtentIndex; // Local extent index
 7   ExtentT Extent; // Extent
 8   CompletionRoutineT CompletionP; // Completion routine
 9   int Reason; // Reason completion called
10 }; // end struct IORequest
11
12
13 class ExtentFileT
14 {
15 public:
16   void DeleteExtent(IORequestT* IORequestP);
17   void ReadExtent(IORequestT* IORequestP);
18   void WriteExtent(IORequestT* IORequestP);
19 }; // end class ExtentFileT
20 struct DirEntryT
21 {
22   PathT PathName;
23   int ExtentFileIndex;
24   int StripeFactor;
25 }; // end struct DirEntryT
26
27
28 class ExtentFileSystemT
29 {
30 public:
31   int OpenExtentFile(const PathNameT& pathName,
32                      ExtentFileT* &extentFileP);
33   int CloseExtentFile(ExtentFileT* &extentFileP);
34   int CreateExtentFile(const PathNameT& pathName,
35                        const int extentFileIndex,
36                        const int stripeFactor);
37   int DeleteExtentFile(const PathNameT& pathName);
38   int CreateDirectory(const PathNameT& pathName);
39   int DeleteDirectory(const PathNameT& pathName);
40   int ListDirectory(const PathNameT& pathName,
41                     ArrayT<DirEntryT>& list) const;
42 }; // end class ExtentFileSystemT
```

Program 5-17. Application programming interface of EFS

All the extent file directory operations offered by an *ExtentFileServer* thread (Program 5-12) and all the extent access operations offered by an *ExtentServer* thread (Program 5-14) are implemented using EFS (Program 5-17).

Program 5-18 shows the C/C++ specification of the *ExtentServerT::ReadExtent* sequential operation based on the asynchronous *ExtentFileT::ReadExtent* method. At line 19, the *VirtualDiskT* virtual disk object (Program 5-11) is retrieved using the virtual disk index of the read extent request. *capParentP* is the pointer to the parent process in the hierarchy, i.e. the pointer to the *VirtualDiskServer* hierarchical process. Since the virtual disks are evenly distributed between the *VirtualDiskServer* hierarchical processes (i.e. the same number of virtual disks is assigned to each *VirtualDiskServer* process), the virtual disk index modulo the number of virtual disks per virtual disk server gives the local virtual disk index (line 19). At line 29, the *capDoNotCallSuccessor* CAP-library function is applied on the output token for suspending the execution of the *ExtentServerT::ReadExtent* operation, i.e. the successor will not be automatically called by the CAP runtime system at the end of the sequential operation. At line 31, the asynchronous *ExtentFileT::ReadExtent* method is called. Once the extent is read, the *::ReadCompletion* routine (line 1) is called and the *capCallSuccessor* CAP-library function is applied on the output token resuming the execution of the parallel *ExtentServerT::ReadExtent* operation, i.e. the successor is explicitly called (line 9).

5.10.1 Extent file structure

When creating an extent file, one does not have to specify its size, i.e. the number of extents and their sizes. An extent file grows as we write extents and shrinks as we delete extents. Since extents in a same extent file may have different header and body sizes, each extent, apart from the extent's data, must store the size of the extent's header and the size of the extent's body. Therefore, to efficiently and easily implement dynamic growing and shrinking extent files, two NTFS regular files are used per extent file. The first NTFS file is used for storing the extents (data headers and data bodies) and the second is used for storing the table of extent addresses and extent sizes, i.e. a table containing for each stored extent its position in the first NTFS file and its header and body size. The first NTFS file is called the *extent data file* and the second NTFS file is called the *extent address table file*. Figure 5-11 shows the internal structure of an extent address table file and an extent data file.

```
static void ReadCompletion(IORequestT* IORequestP)
{
  ReadExtentOT* OutputP = (ReadExtentOT*) IORequestP->InfoP;

  OutputP->Extent = IORequestP->Extent;
  OutputP->ErrorCode = IORequestP->Reason;
  delete IORequestP;

  capCallSuccessor(OutputP);
} // end ReadCompletion

leaf operation ExtentServer::ReadExtent
  in ReadExtentIT* InputP
  out ReadExtentOT* OutputP
{
  VirtualDiskT* VirtualDiskP;
  IORequestT* IORequestP;

  VirtualDiskP = capParentP->VirtualDisk[InputP->VirtualDiskIndex % NumberOfVirtualDisksPerVirtualDiskServer];
  IORequestP = new IORequestP;

  IOPRequestP->InfoP = OutputP;
  IORequestP->ExtentCacheOn = VirtualDiskP->ExtentFileTable[InputP->ExtentFileDescriptor].ExtentCacheOn;
  IORequestP->ExtentIndex = InputP->ExtentIndex;
  IORequestP->Extent = NULL;
  IORequestP->IOCompletion = ::ReadCompletion;
  IORequestP->Reason = 0; // No error

  capDoNotCallSuccessor(OutputP);

  VirtualDiskP->ExtentFileTable[InputP->ExtentFileDescriptor].ExtentFileP->ReadExtent(IORequestP);
} // end leaf operation ExtentServer::ReadExtent
```

Program 5-18. Asynchronous programming of the *ExtentServerT::ReadExtent* sequential operation using the *capDoNotCallSuccessor* and *capCallSuccessor* CAP-library functions

Internal structure of an extent address table file

The first 2048 bytes of an extent address table file are reserved as header for storing the metadata, i.e. the file signature specifying that the NTFS file is an extent address table file, the version of the extent address table file, the endian code specifying which endian mode (little or big endian) is used in this file for storing 32-bit value, the index of this extent file, and the striping factor of the parallel file. The table of extent addresses starts at the offset 2048 bytes. Each entry of the table is made of 4 32-bit values. The first two values contain the size of the extent's header and the size of the extent's body in bytes. The last two values of an entry contain the byte offset within the extent data file of the block where the extent's data is stored (*ExtentPosition* field), i.e. the address of the extent, and the size of this block (*OriginalSize* field), i.e. the space reserved in the extent data file for storing the extent. Of course, the size of a block where an extent is stored must be greater than or equal to the size of the extent's header plus the size of the extent's body, i.e. *OriginalSize* >= *HeaderSize* + *BodySize*. The extent address table is organized so as to have the address and the size of the i^{th} extent stored in the i^{th} entry enabling to quickly find the address of an extent.

Internal structure of an extent data file

The first 2048 bytes of an extent data file are reserved as header for storing the metadata, i.e. the file signature specifying that the NTFS file is an extent data file, the version of the extent data file, the endian code specifying which endian mode (little or big endian) is used in this file for storing 32-bit value, the index of this extent file, the striping factor of the parallel file, and the position of the first hole. Extents are stored in the extent data file in the order they are written and the space reserved for an extent, i.e. the *OriginalSize* field in the extent address table file, is always a multiple of 2048 bytes so as to be able to read extent data files residing on CD's where the block size is 2048 bytes (remember that for performance EFS disables the NTFS cache forcing to align data accesses in NT files, i.e. the position and the size of an I/O request must be multiple of the block size of the device). The header and the body of an extent are always contiguously stored within an extent data file, i.e. with a single NTFS I/O request, both the header and the body are read or written. Since we can delete extents from an extent file, an extent data file can contain holes. A hole is created whenever an extent is deleted. Each extent data file contains a list of holes where each hole contains its size and the position of the next hole in the list. The last hole in the list contains a null pointer as position of the next hole. An extent data file grows as we write extents and shrinks as we delete extents.

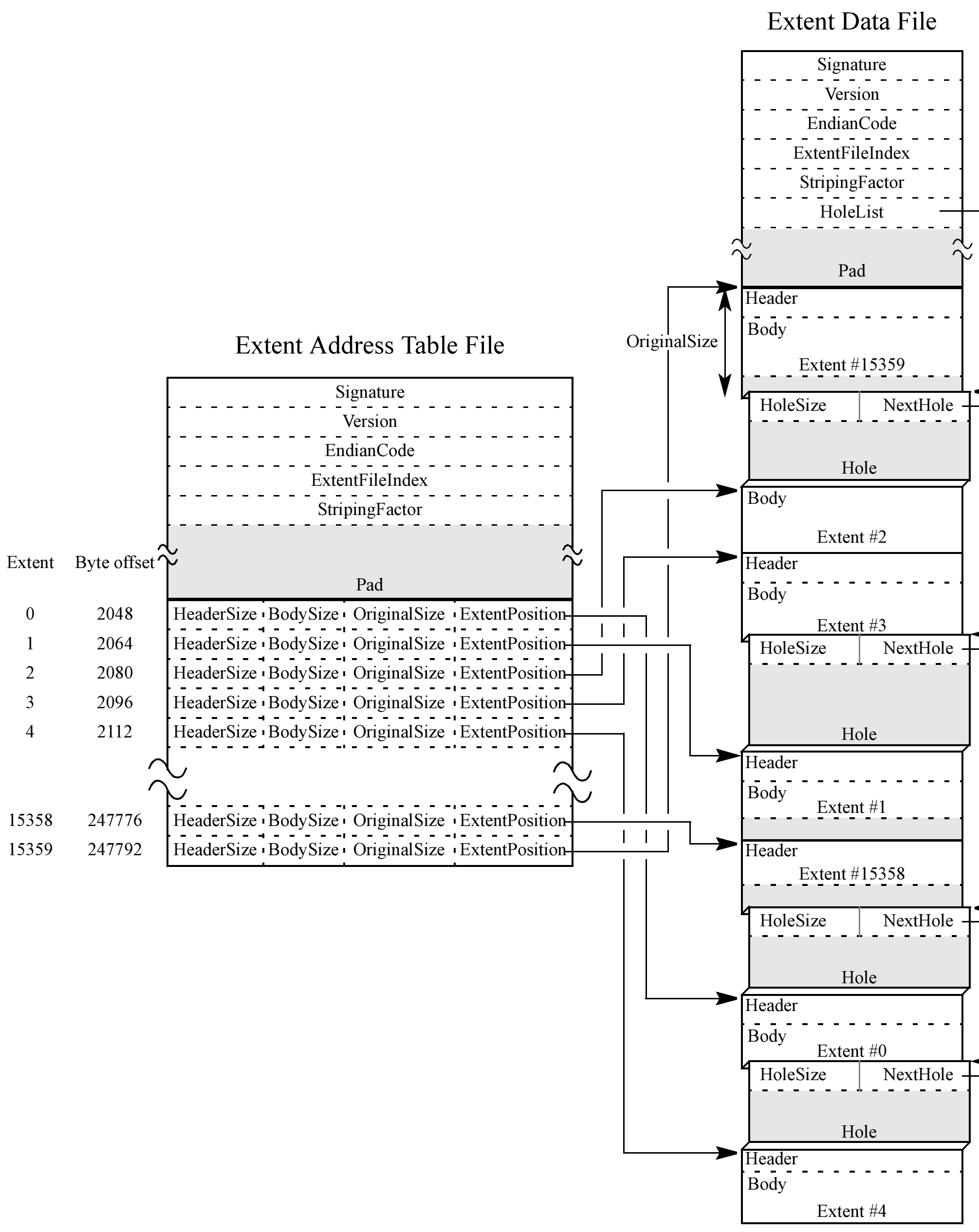


Figure 5-11. Internal structure of an extent file

5.10.2 Internal organization of EFS

For each open extent file there is a write-back 320KB[1] extent address cache keeping in memory most recently used extent address entries. EFS divides an extent address table into segments of 1024 entries, i.e. by chunk of 16 KBytes. Each time an extent is accessed (read, written or deleted), the extent address cache is consulted and if the required entry is not found, then the whole 16KB segment containing the required entry is read from the extent address table file into the extent address cache. When the extent address cache becomes full, the least recently used 16KB segment (LRU replacement policy) is replaced and the segment is possibly written back to the extent address table file in the case that at least one entry was modified (write-back cache). In the current release of EFS, reading and writing segments from extent address table files are done synchronously, i.e. during an extent address miss, the *ExtentServer* thread is blocked reading (and also possibly writing) an extent address 16KB segment. Meanwhile the *ExtentServer* thread cannot serve other extent access requests directed to different virtual disks

1. If we assume, for example, that an extent file contains extents of 50 KBytes each, then an extent address cache of 320 KBytes enables to keep in memory 20’480 extent positions corresponding to an extent data file of 20’480 extents x 50 KBytes = 1’000 MBytes.

located on the same SP node. This feature can be seen when a parallel file is freshly opened and all the extent address caches are empty. When the application start issuing extent access requests to the *ExtentServer* threads (one *ExtentServer* thread per SP node), disks located on a same SP node are accessed sequentially (for reading extent address segments) and not in parallel. For an improved version of EFS, this issue should be addressed.

When an extent file is opened, EFS reads the extent data file's hole list and builds an in-memory list enabling to efficiently manage and query the list. When writing a new extent, before expanding the extent data file and writing the extent at the end of the extent data file, EFS first searches for a hole in the in-memory list that is greater or equal to the extent size (best fit algorithm). When deleting an extent, EFS inserts the newly created hole in the in-memory hole list and possibly coalesces contiguous holes. Of course, if one intensively writes and deletes various-sized extents from an extent file, the extent data file might become fragmented, i.e. it may contain many small holes that cannot be reused and consume lots of disk space. Therefore, this issue should be addressed in a future version of EFS.

5.11 PS^2 configuration files

To run a PS^2 application on distributed memory PC's, two configuration files must be provided. One for the CAP runtime system specifying the mapping of threads to Windows NT processes (Section 3.6) and one for PS^2 specifying the NTFS root directories of each virtual disk in the parallel storage and processing server.

Program 5-19 shows a CAP configuration file for a 3-SP-node PS^2 server configuration (Figure 5-8). The *InterfaceServer* thread is generally mapped onto the user's PC where the application's front end runs. Three *ComputeServer* threads are mapped per SP node so as to benefit from dual processor PC's and a single *DiskServer* hierarchical process is mapped per SP node, i.e. an *ExtentFileServer* thread and an *ExtentServer* thread.

```
configuration {
processes:
  A ( "user" ) ;
  B ( "128.178.75.65", "\\FileServer\SharedFiles\Ps2Executable.exe" ) ;
  C ( "128.178.75.66", "\\FileServer\SharedFiles\Ps2Executable.exe" ) ;
  D ( "128.178.75.67", "\\FileServer\SharedFiles\Ps2Executable.exe" ) ;

threads:
  "InterfaceServer" (A) ;
  "ComputeServer[0]" (B) ;
  "ComputeServer[1]" (B) ;
  "ComputeServer[2]" (B) ;
  "ComputeServer[3]" (C) ;
  "ComputeServer[4]" (C) ;
  "ComputeServer[5]" (C) ;
  "ComputeServer[6]" (D) ;
  "ComputeServer[7]" (D) ;
  "ComputeServer[8]" (D) ;
  "VirtualDiskServer[0].ExtentFileServer" (B) ;
  "VirtualDiskServer[0].ExtentServer" (B) ;
  "VirtualDiskServer[1].ExtentFileServer" (C) ;
  "VirtualDiskServer[1].ExtentServer" (C) ;
  "VirtualDiskServer[2].ExtentFileServer" (D) ;
  "VirtualDiskServer[2].ExtentServer" (D) ;
}; // end of configuration file
```

Program 5-19. CAP configuration file mapping the *InterfaceServer* thread onto the user's PC, three *ComputeServer* threads per SP node, one *ExtentFileServer* thread per SP node, and one *ExtentServer* thread per SP node

Program 5-20 shows the PS^2 configuration file for a 27 disk PS^2 server configuration. A PS^2 configuration file, often named the *ServerDisk.txt* file, gives the NTFS root directories of each virtual disk. Each path name corresponds to an NTFS directory of a virtual disk, which is accessible from the assigned *VirtualDiskServer* hierarchical process, i.e. from the *VirtualDiskServer* process to which this virtual disk is assigned. Remember that PS^2 evenly assigns the virtual disks to the *VirtualDiskServer* processes, i.e. the virtual disks 0 to 8 are assigned to the *VirtualDiskServer[0]* hierarchical process, the virtual disks 9 to 17 are assigned to the VirtualDiskServer[1] hierarchical process, and the virtual disks 18 to 26 are assigned to the *VirtualDiskServer[2]* hierarchical process.

Program 5-21 shows how to start a PS^2 application on a 3-SP-node PS^2 server with a CAP configuration file and a PS^2 *ServerDisk.txt* configuration file. The following switches are automatically defined for a PS^2 application:
- The '*-cnf*' switch specifies the CAP configuration file.
- The '*-nod*' switch specifies the number of virtual disks.
- The '*-ro*' switch specifies whether the virtual disks are read only or not.

```
H:\PsPs
I:\PsPs
J:\PsPs
K:\PsPs
L:\PsPs
M:\PsPs
N:\PsPs
O:\PsPs
P:\PsPs

H:\PsPs
I:\PsPs
J:\PsPs
K:\PsPs
L:\PsPs
M:\PsPs
N:\PsPs
O:\PsPs
P:\PsPs

H:\PsPs
I:\PsPs
J:\PsPs
K:\PsPs
L:\PsPs
M:\PsPs
N:\PsPs
O:\PsPs
P:\PsPs
```

Program 5-20. PS^2 configuration file specifying the NTFS root directories of each virtual disk

- The '*-nod*s' switch specifies the number of *VirtualDiskServer* processes that the CAP runtime system must create, i.e. the number of *ExtentFileServer* threads and the number of *ExtentServer* threads. PS^2 imposes that the number of *VirtualDiskServer* processes be multiple of the number of virtual disks (Program 5-3).
- The '*-nocs*' switch specifies the number of *ComputeServer* threads that the CAP runtime system must create. PS^2 imposes that the number of *ComputeServer* threads be multiple of the number of virtual disks (Program 5-4).
- The '*-sdps*' switch specifies the PS^2 *ServerDisk.txt* configuration file.

```
Ps2Executable.exe -cnf \\FileServer\SharedFiles\ConfigurationFile.txt -nod 27 -ro 1 -nods 3 -nocs 9
                  -sdps \\FileServer\SharedFiles\ServerDisk.txt
```

Program 5-21. Starting a PS^2 application with a CAP configuration file and a PS^2 configuration file at DOS prompt

Once all the threads are launched on a PS^2 server (Section 3.6), PS^2 parses the *ServerDisk.txt* configuration file and for each virtual disk, a *MountExtentFileSystem* request (Program 5-12) is sent to the appropriate *ExtentFileServer* thread for mounting the NTFS root directory onto the virtual disk.

5.12 Summary

This chapter has described our PS^2 framework for developing parallel I/O- and compute- intensive applications. We have seen that PS^2 is not a conventional parallel storage system intended to directly meet the specific need of every applications. Our approach has been to design an extensible parallel I/O-and-compute framework for developing C/C++ high-level libraries, each of which designed to meet the needs of a specific community of applications by providing specific abstractions of files, e.g. 2D images or matricies. The trend in parallel storage systems is to provide flexible I/O systems where library programmers are free to modify the declustering strategy, the prefetching mechanism, the caching strategy, and even to program their own I/O interfaces on declustered files. However PS^2 goes a step further by providing not only a flexible parallel storage system, but also an extensible parallel processing system where the two systems are deeply integrated into a same framework. This enables application programmers to rapidly and easily develop pipelined parallel I/O-and-compute operations on declustered files that are incorporated into C/C++ high-level libraries.

In this chapter we have seen that a PS^2 server architecture, i.e. the parallel architecture on which PS^2 applications execute, comprises a number of server PC's connected to a Fast Ethernet network and offering data storage and processing services to application threads located over the network on a number of client PC's. PS^2 breaks away the distinction between I/O nodes and compute nodes, by executing on the server PC's both extent access operations and processing operations. Therefore, in the PS^2 terminology, we call these server PC's, storage and processing nodes (SP nodes).

This chapter has presented the PS^2 two-dimensional extent-oriented parallel file structure giving library programmers the ability to control both how data is distributed across the disks, and the degree of parallelism exercised on every subsequent access. The extensible PS^2 server CAP process hierarchy has been described. It mainly consists of a set of storage threads offering low-level parallel file access components and a set of compute threads fully extensible by application programmers. Thanks to a CAP configuration file, these threads are mapped onto a PS^2 server architecture so as at least one storage thread and one compute thread run per SP node. Thanks to the CAP formalism, application programmers are able to extend the functionalities of the compute threads by incorporating application-specific processing operations. These application-specific processing operations execute on the SP nodes and can, therefore, perform computations on locally stored data thus avoiding superfluous data communication. In order to access disks through local native file system calls (NTFS), disk access requests are directed to the storage thread located on the SP node where data resides, i.e. on which the disk is hooked. Similarly, in order to process locally stored data, processing requests are directed to the compute thread located on the SP node where its companion storage thread executes, i.e. where data resides. Thanks to CAP, new parallel I/O- and compute- intensive operations can be conceived by combining the predefined low-level parallel file access components offered by the storage threads with the application-specific processing operations offered by the compute threads. These new processing operations on parallel files can be incorporated into high-level C/C++ libraries offering application-specific abstractions of files.

We have seen how parallel file directory operations are synthesized using CAP. PS^2 incorporates an interface server thread coordinating parallel file directory operations and resolving race conditions on parallel files.

The end of the chapter has presented the single-disk extent-oriented file system (EFS) providing PS^2 with a portable abstraction of extent file on top of native file systems, e.g. the Windows NT file system. EFS comprises a write-through cache of extents with a least recently used (LRU) replacement policy. We have described how parallel file access operations are programmed using the asynchronous extent access routines of EFS and the *capDoNotCallSuccessor* and *capCallSuccessor* CAP-library functions.

Finally, we have seen how to launch a PS^2 application with a CAP configuration file and a PS^2 configuration file. The CAP configuration file gives the mapping of threads to SP nodes. The PS^2 configuration file gives the (NTFS) root path names of each disk presents in the PS^2 architecture.

Chapter 6

Developing Parallel I/O- and Compute- Intensive Applications using PS^2

6.1 Introduction

This chapter presents examples of high-level libraries and applications based on the PS^2 framework. Section 6.2 describes how to develop a C/C++ high-level library offering the abstraction of 2D images along with parallel imaging operations. Section 6.3 demonstrates the applicability and the performances of the PS^2 tools on a real parallel 3D image slice extraction application. This application enables clients to specify, access in parallel, and visualize image slices having any orientation and position. The image slices are extracted from huge 3D images declustered across multiple disks distributed between several SP nodes. The end of Section 6.3 analyses the performances of the pipelined parallel slice extraction and visualization operation.

6.2 Synthesizing parallel imaging operations using PS^2

This section describes how to develop parallel imaging operations using PS^2. The presented examples demonstrate how the CAP tool enables application programmers to easily and efficiently combine library-specific operations with the reusable parallel file system components offered by the customizable parallel file system (PS^2) in order to yield pipelined parallel I/O and compute intensive imaging operations. The flexibility of the CAP tool is also shown. Indeed, CAP enables programmers to rapidly modify the schedule of operations, i.e. processing operations and I/O access operations, and to incorporate new operations in a reading or writing pipeline.

Section 6.2.1 presents a 2D image file declustering strategy dividing large 2D image files in rectangular tiles and storing independently these tiles on multiple disks. This declustering strategy ensures that for most imaging applications, disks and SP node accesses are close to uniformly distributed, i.e. the I/O load is balanced between disks and the computation load is balanced between SP node processors.

Parallel imaging operations can be divided in two categories: neighbourhood independent operations and neighbourhood dependent operations. A neighbourhood independent operation is an operation where processing an image part, i.e. a 2D tile, can be done without requiring neighbouring tiles. Such operations may include compression, zooming, linear filtering operations and saving the image to disks. On the other hand, neighbourhood dependent operations requires fetching the 8 tile borders. Non-linear filtering operations such as morphological operations (erosion, dilatation) are typically neighbourhood dependent.

Section 6.2.2 describes neighbourhood independent imaging operations using PS^2. First, a parallel write window access operation without processing operation is described. Thanks to the CAP formalism, a compression stage is easily incorporated into the parallel operation enabling to write compressed tiles. Finally, we present an improved version of the parallel compress-and-write window operation requiring less communication. Section 6.2.3 describes neighbourhood dependent imaging operation using PS^2. This example demonstrates the ability of the CAP tool and the PS^2 framework to program complicated parallel operations requiring communications between SP nodes.

6.2.1 2D image file declustering strategy

Large 2D images are often divided into square or rectangular tiles which are independently stored on multiple disks. Pixmap image tiling is routinely used for internal representation of images in software packages such as PhotoShop. Rectangular tiles enable programmers to access image windows efficiently and to apply parallel processing operations on large 2D images, e.g. zooming, rotating, filtering, etc, with good data locality [Hersch93]. A tile represents the unit of I/O access, i.e. a tile is stored as a contiguous disk block. A tile represents also the unit of processing, i.e. a parallel imaging operation is decomposed into sequential operations performing elementary processing on tiles which are then combined in a pipelined parallel manner. For example, a parallel image rotation operation consists of performing in parallel a rotation on each tile making up the 2D window specified by a user and of sequentially merging the rotated tiles into an application's buffer.

Concerning the size of the rectangular tiles a trade off between data locality when applying imaging operations and the effective sustained I/O throughput needs to be found. As far as data locality is concerned, a tile should be as small as possible so that only the required part of a 2D image file is read from disks, e.g. when zooming a visualization window only the image data covering the visualization window needs to be actually read from disks (Figure 6-1). On the other hand, in order to maximize the effective sustained I/O throughput, a tile should be as large as possible so as to reduce the time lost in head displacement, i.e. the disk latency becomes small compared to the disk data transfer time. Previous works [Gennart94] shows that a good tile size for a wide variety of parallel imaging applications can be achieved when the disk latency time is similar to the disk transfer time. For example, for the IBM DPES-31080 disk drive, which have a measured latency of 13.77 ms and a measured throughput of 3.5 MBytes/s (Section 4.2.1), the appropriate tile size is around 47 KBytes.

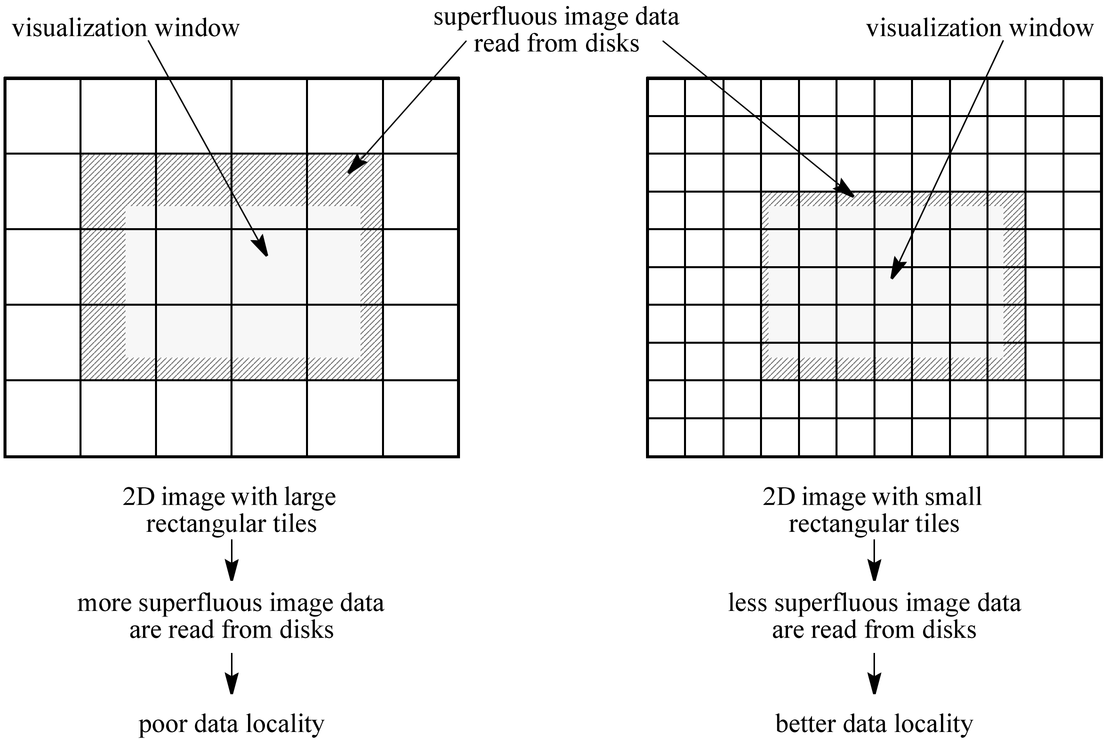

visualization window
superfluous image data read from disks
visualization window
2D image with large rectangular tiles
more superfluous image data are read from disks
poor data locality
2D image with small rectangular tiles
less superfluous image data are read from disks
better data locality

Figure 6-1. For a good data locality when applying an imaging operation on a tiled 2D image, the tile size should be as small as possible in order to reduce the amount of superfluous image data read from disks

The effective transfer time is given by Equation 6-1. An optimal tile size may be computed for a given window size. However this is not very critical since window sizes considerably vary.

$$\text{EffectiveDiskTransferTime} = \text{DiskLatencyTime} + \frac{TileSize}{\text{DiskThroughput}} \tag{6-1}$$

In order to decluster large 2D image files across several disks and to develop parallel imaging operations using the PS^2 customizable parallel file system, rectangular tiles of a 2D image are distributed between the extent files making up a parallel file. Tiles are stored in extents (one extent per tile) and the extents are distributed between the extent files residing on different virtual disks mapped to different physical disks (i.e. one extent file per physical disk). The declustering strategy is the function that distributes tiles of a 2D image between the extent files of a parallel file. This strategy must be devised so that when performing a parallel processing operation on a declustered 2D image file, the I/O load is balanced between the physical disks and the processing load is balanced between the SP nodes.

The distribution of tiles to extent files is made so as to ensure that direct 2D tile neighbours reside on different extent files located on physical disks hooked on different SP nodes. We achieve such a distribution by storing, in a round-robin manner, 2D tiles on successive extent files. Successive extent files reside on physical disks hooked on successive SP nodes. We introduce between two successive rows of 2D tiles an extent file offset (*ExtentFileOffset*$_Y$) which is prime to the number of extent files (*NExtentFiles*) making up the parallel image file. Since the extent files reside on different virtual disks mapped to different physical disks, the extent file offset corresponds to a physical disk offset and the number of extent files corresponds to the number of physical disks across which the parallel image file is declustered. Figure 6-2 shows an image divided in tiles, as well as a visualization window covering part of the image. Figure 6-2 also shows the distribution of the image tiles among the extent files, assuming the image is striped over 4 extent files (4 physical disks) and the extent file offset is 3 disks. The allocation index consists of the extent file index (*ExtentFileIndex*) as well as the local extent index (*ExtentIndex*) on that extent file. For example, the bottom right tile in Figure 6-2 is allocated on the extent file 1, and is the 17th tile (or extent) on that extent file.

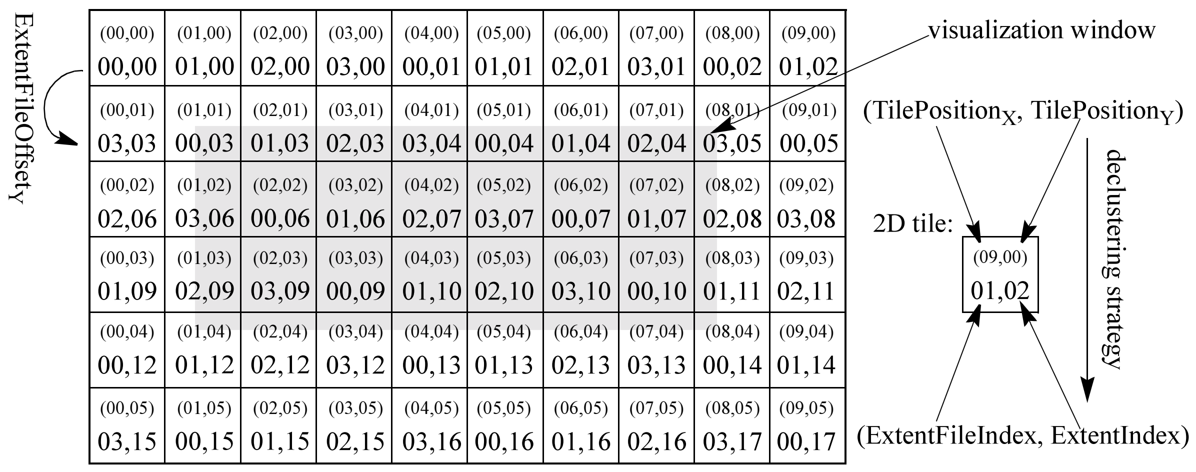


Figure 6-2. 2D image tiling and the declustering strategy

The declustering strategy shown in Figure 6-2 ensures that for most parallel imaging applications, disk and SP node accesses are close to uniformly distributed. Equation 6-2 gives the distribution of tiles to extent files (or physical disks) where *TilePosition*$_X$, and *TilePosition*$_Y$ are the tile coordinate, and *NTiles*$_X$ is the number of tiles per row. Equation 6-3 gives the distribution of tiles within a particular extent file computed using Equation 6-2 (all divisions are integer divisions). Therefore a particular tile (or extent) is uniquely identified with its *ExtentFileIndex* specifying on which extent file the tile resides and its *ExtentIndex* specifying the local extent index within the set of extents stored on that particular extent file.

$$ExtentFileIndex = (TilePosition_Y \cdot \mathrm{ExtentFileOffset_Y} + TilePosition_X) \ \mathrm{mod} \ \mathrm{NExtentFiles} \qquad \textbf{(6-2)}$$

$$ExtentIndex = TilePosition_Y \cdot \left(1 + \frac{\mathrm{NTiles_X} - 1}{\mathrm{NExtentFiles}}\right) + \frac{TilePosition_X}{\mathrm{NExtentFiles}} \qquad \textbf{(6-3)}$$

The C/C++ specification of the 2D image file declustering strategy (Equations 6-2 and 6-3) is shown in Program 6-1. The *EXTENT_FILE_OFFSET_Y* table (line 1) gives the *ExtentFileOffsetY* value according to the number of extent files across which the 2D image file is declustered.

6.2.2 Neighbourhood independent imaging operations

This section describes how we customize the PS^2 parallel storage and processing server (Section 5.8.1) to incorporate in it a new pipelined parallel operation that writes a 2D image window into a parallel image file. For the sake of simplicity, the operation described in this section only enables windows that span multiple tiles to be written, i.e. window boundaries correspond with tile boundaries (Figure 6-3).

```
int EXTENT_FILE_OFFSET_Y[] = // table giving the ExtentFileOffsetY according to
  {0,                        // the number of extent files making up the parallel image file
   1,   /* 1 extent file */
   1,   /* 2 extent files */
   1,   /* 3 extent files */
   ...
  };

void DeclusteringFunction(const int NExtentFiles,
                          const int NTilesX,
                          const int NTilesY,
                          const int TilePositionX,
                          const int TilePositionY,
                          int& ExtentFileIndex,
                          int& ExtentIndex)
{
  int ExtentFileOffsetY = EXTENT_FILE_OFFSET_Y[NExtentFiles];

  ExtentFileIndex = (TilePositionY * ExtentFileOffsetY+ TilePositionY) % NExtentFiles;
  ExtentIndex = TilePositionY * (1+(NTilesX-1)/NExtentFiles) + TilePositionX/NExtentFiles;
} // end DeclusteringFunction
```

Program 6-1. Declustering function converting a tile coordinate (*TilePositionX*, *TilePositionY*) into an extent location (*ExtentFileIndex*, *ExtentIndex*)

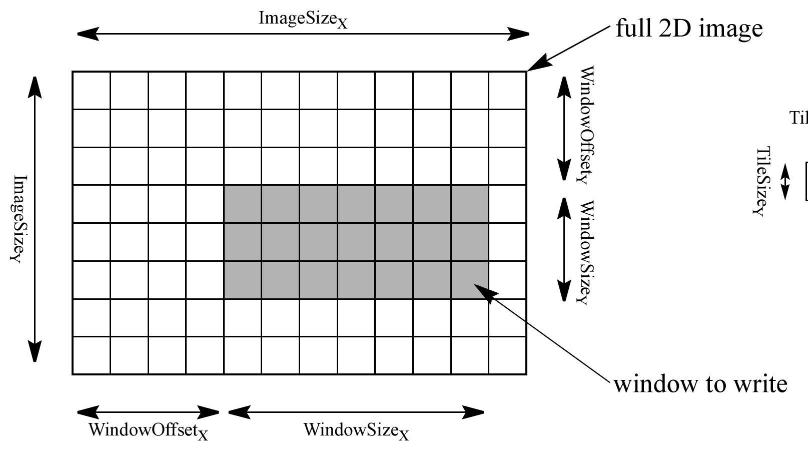
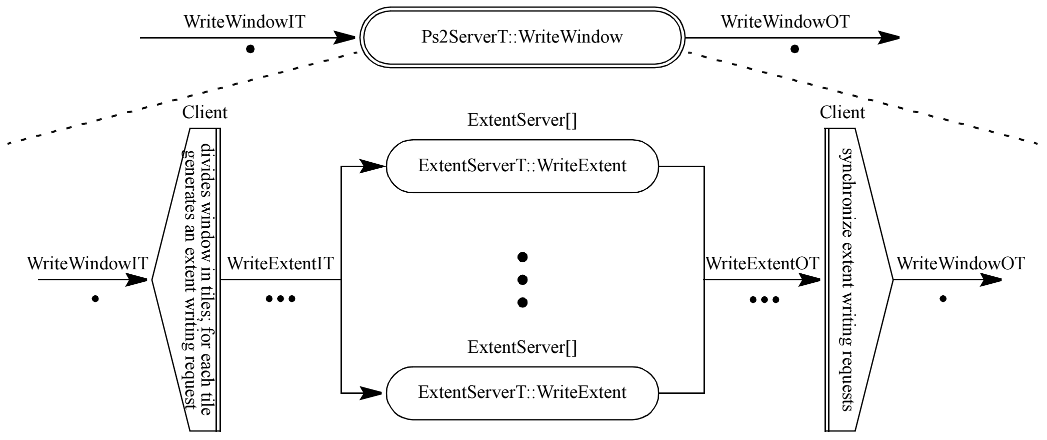


Figure 6-3. Parameters needed for writing a window into a full declustered 2D image file

Figure 6-4 shows the DAG of the parallel *Ps2ServerT::WriteWindow* operation. The operation consists in dividing the 2D image window in tiles and for each tile sending an extent writing request to the appropriate *ExtentServer* thread based on the declustering strategy shown in Section 6.2.1. The operation completes when all the tiles making up the 2D image window are written, i.e. when all *ExtentServerT::WriteExtent*'s output tokens are merged. Both the split and the merge functions are performed by the *Client* thread, i.e. the thread who initiates the parallel *PS2ServerT::WriteWindow* operation.



WriteWindowIT
Ps2ServerT::WriteWindow
WriteWindowOT
Client
ExtentServer[]
Client
divides window in tiles; for each tile generates an extent writing request
ExtentServerT::WriteExtent
synchronize extent writing requests
WriteWindowIT
WriteExtentIT
WriteExtentOT
WriteWindowOT
ExtentServer[]
ExtentServerT::WriteExtent

Figure 6-4. Graphical CAP specification of the parallel *Ps2ServerT::WriteWindow* operation

Program 6-2 shows the declaration of *Ps2ServerT::WriteWindow*'s input and output tokens. The input token (lines 7-20) comprises the pointer to the window data buffer (line 9), the size of the full 2D image in tiles (lines 10-11), the size of a tile in bytes (lines 12-13), the size of the image window to write in tiles (lines 14-15), the position of the image window from the upper left corner of the full 2D image (lines 16-17), the indices of the virtual disks where the extent files making up the parallel image file reside (line 18), and their extent file descriptors (line 19). The output token (lines 22-25) merely comprises an error code indicating whether the operation completes successfully or not (line 24).

```
 1 class BufferT
 2 {
 3 public:
 4   char* BufferP;
 5 }; // end class BufferT
 6
 7 token WriteWindowIT
 8 {
 9   BufferT WindowP;
10   int ImageSizeX;
11   int ImageSizeY;
12   int TileSizeX;
13   int TileSizeY;
14   int WindowSizeX;
15   int WindowSizeY;
16   int WindowOffsetX
17   int WindowOffsetY;
18   ArrayT<int> VirtualDiskIndex;
19   ArrayT<int> ExtentFileDescriptor;
20 }; // end token WriteWindowIT
21
22 token WriteWindowOT
23 {
24   int ErrorCode;
25 }; // end token WriteWindowOT
```

Program 6-2. Declaration of the input and output tokens of the parallel *Ps2ServerT::WriteWindow* operation

Program 6-3 shows the CAP specification of the parallel *Ps2ServerT::WriteWindow* operation. In order to extend the functionalities of the PS^2 server, i.e. to incorporate a new library-specific parallel operation into the *Ps2ServerT* hierarchical process, we use an additional CAP operation declaration (Program 3-7). Lines 42-44 show the declaration of the new parallel *Ps2ServerT::WriteWindow* operation outside the scope of the *Ps2ServerT* process declaration (Program 5-6).

The parallel *Ps2ServerT::WriteWindow* operation (Program 6-3) consists of a single flow-controlled *indexed parallel* CAP construct (lines 50-58) that sends an extent writing request to the appropriate *ExtentServer* thread (line 57) for each tile of the 2D window, i.e. *WindowSizeX* tiles per *WindowSizeY* tiles (lines 52-53). The *GenerateExtentWritingRequest* split function (lines 1-28) first converts an absolute tile coordinate (lines 11-12) into an extent coordinate (lines 13-14), i.e. a local extent index within an extent file, using the declustering strategy in Program 6-1. Using the *VirtualDiskIndex* mapping table (Section 5.7), the index of the virtual disk where the selected extent file, i.e. whose index is *ExtentFileIndex*, resides is retrieved (line 18). The *CopyTile* function (lines 22-27) copies the current tile from the data window buffer into the extent writing request. Once an extent writing request is generated by the *GenerateExtentWritingRequest* split function, it is redirected to the appropriate *ExtentServer* thread (line 57) using the *ToVirtualDiskServerIndex* macro (Program 5-3) converting a virtual disk index into a virtual disk server index. The *ExtentServerT::WriteExtent*'s output tokens are merged by the *MergeWriteExtentAnswer* merge function (lines 30-39) that merely propagates the first extent writing error.

As already mentioned in Section 5.2, the PS^2 framework is mainly used for developing high-level C/C++ libraries providing application-specific abstraction of files, e.g. matrices, images, etc, with appropriate parallel storage and processing operations on that abstraction, e.g. multiplying two out-of-core matrices or zooming a visualization window from a huge 2D image.

Program 6-4 shows how the parallel *Ps2ServerT::WriteWindow* operation (Program 6-3) is incorporated into a high-level C/C++ image library using the *call* CAP instruction (Program 3-8). The first argument of the *WriteWindow* function (line 25) is the descriptor of the image returned when the image file was first opened using the *OpenImage* function (lines 13-14). The *OpenImage* function opens the corresponding parallel file (line 18), reads the extent containing the image metadata from the first extent file (line 21), i.e. the size of the image, the size of a tile, etc, and fills a new *ImageDescriptorT* (lines 1-9) entry of the *ImageFileTable* table (line 11). The second and third arguments of the *WriteWindow* function (lines 26-27) are the window sizes in tiles and the next two arguments (lines 28-29) are the positions of the window from the upper left corner of the full 2D image. The last argument (line 30) is the window data buffer. The *WriteWindow* function first creates the *WriteWindowIT* input token (line 35) and fills the token with the user's provided arguments and the image's metadata (lines 36-46). The parallel *Ps2ServerT::WriteWindow* operation is called at line 48. At line 52, the error code of the parallel operation (line 49) is returned back to the caller.

```
 1 void GenerateExtentWritingRequest(WriteImageIT* inputP,
 2                                  WriteExtentIT* &subtokenP,
 3                                  int TileRelativePositionY,
 4                                  int TileRelativePositionX)
 5 {
 6   int ExtentFileIndex, ExtentIndex;
 7
 8   DeclusteringFunction(inputP->VirtualDiskIndex.Size(),
 9                        inputP->ImageSizeX,
10                        inputP->ImageSizeY,
11                        inputP->WindowOffsetX + TileRelativePositionX,
12                        inputP->WindowOffsetY + TileRelativePositionY,
13                        ExtentFileIndex,
14                        ExtentIndex);
15
16   subtokenP = new WriteExtentIT;
17
18   subtokenP->VirtualDiskIndex = inputP->VirtualDiskIndex[ExtentFileIndex];
19   subtokenP->ExtentFileDescriptor = inputP->ExtentFileDescriptor[ExtentFileIndex];
20   subtokenP->ExtentIndex = ExtentIndex;
21
22   CopyTile(subtokenP->Extent,
23            inputP->WindowP,
24            inputP->TileSizeX,
25            inputP->TileSizeY,
26            TileRelativePositionX,
27            TileRelativePositionY);
28 } // end GenerateExtentWritingRequest
29
30 void MergeWriteExtentAnswer(WriteWindowOT* outputP,
31                             WriteExtentOT* subtokenP,
32                             int TileRelativePositionY,
33                             int TileRelativePositionX)
34 {
35   if ( (subtokenP->ErrorCode != 0) && (outputP->ErrorCode == 0) )
36   {
37     outputP->ErrorCode = subtokenP->ErrorCode;
38   } // end if
39 } // end MergeWriteExtentAnswer
40
41
42 operation ps2ServerT::WriteWindow // Extending the functionalities of the Ps2Server:
43   in WriteWindowIT* InputP        // declaring an additional parallel operation
44   out WriteWindowOT* OutputP;
45
46 operation ps2ServerT::WriteWindow // Definition of the additional parallel operation
47   in WriteWindowIT* InputP
48   out WriteWindowOT* OutputP
49 {
50   flow_control(100)
51   indexed
52     (int TileRelativePositionY = 0; TileRelativePositionY < thisTokenP->WindowSizeY; TileRelativePositionY++ )
53     (int TileRelativePositionX = 0; TileRelativePositionX < thisTokenP->WindowSizeX; TileRelativePositionX++ )
54   parallel
55     (GenerateExtentWritingRequest, MergeWriteExtentAnswer, Client, local WriteWindowOT Result() )
56   (
57     VirtualDiskServer[ToVirtualDiskServerIndex(thisTokenP->VirtualDiskIndex)].ExtentServer.WriteExtent
58   ); // end indexed parallel
59 } // end operation ps2ServerT::WriteWindow
```

Program 6-3. CAP specification of the parallel *PS2ServerT::WriteWindow* operation

Thanks to the CAP formalism and to the customizable PS^2 framework, it is easy to combine the PS^2 reusable parallel file system components (e.g. *WriteExtent* in Program 6-3, line 57) in a pipelined parallel manner. For example, in the parallel *Ps2ServerT::WriteWindow* operation (Program 6-3) we can elegantly insert an extent compression stage within the writing pipeline, i.e. just before writing the extent. Figure 6-5 shows the modified DAG of the parallel *Ps2ServerT::WriteWindow* operation incorporating a new compression stage. Note that the compression operation is performed on the SP nodes where the extents are actually written so as to perform the compression in parallel on several SP nodes.

In the DAG of Figure 6-5, pipelining is achieved at three levels

- the *ExtentServer* thread writes a compressed extent while its companion *ComputeServer* thread located on the same SP node compresses the next extents,
- The *ComputeServer* thread compresses an extent while the same *ComputeServer* thread asynchronously receives from the network the next extents to be compressed and written,
- the *Client* thread asynchronously sends an extent to a *ComputeServer* thread while the same *Client* thread continues to divide the 2D window in rectangular tiles and to generate the next extent writing requests.

And the parallelization occurs at two levels:

- several extents are written simultaneously from different disks; the number of disks can be increased to improve I/O throughput,

```
struct ImageDescriptorT // An ImageDescriptorT object contains all the metadata
{                       // related to an image file.
  int ImageSizeX;       // Number of tiles along the X axis
  int ImageSizeY;       // Number of tiles along the Y axis
  int TileSizeX;        // Size of a tile along the X axis (number of bytes)
  int TileSizeY;        // Size of a tile along the Y axis (number of bytes)
  ArrayT<int> VirtualDiskIndex;     // Virtual disks across which the image is striped
  ArrayT<int> ExtentFileDescriptor; // Descriptors of the extent files making up the image file
}; // end ImageDescriptorT

ImageDescriptorT ImageFileTable[MAX_OPEN_IMAGES]; // Image descriptor of each open image

int OpenImage(PathNameT pathName,    // in:  Path name of the image to open
              int& imageDescriptor)  // out: Image descriptor, i.e. an index within the ImageFileTable
{
  // ...
  // Opens the parallel image file
  call Ps2Server.OpenParallelFile in ... out ...;
  // ...
  // Reads the extent containing the image metadata in the first extent file
  call Ps2Server.VirtualDiskServer[VirtualDiskIndex[0]].ExtentServer.ReadExtent in ... out ...;
  // ...
} // end OpenImage;

int WriteWindow(int imageDescriptor,  // in: Image descriptor, i.e. the index within the ImageFileTable
                int windowSizeX,      // in: Size of the window along the X axis (number of tiles)
                int windowSizeY,      // in: Size of the window along the Y axis (number of tiles)
                int windowOffsetX,    // in: Position of the window from the upper left corner of the image
                int windowOffsetY,    // in: Position of the window from the upper left corner of the image
                BufferT windowP)      // in: Window data buffer
{
  WriteWindowIT* InputP;
  WriteWindowOT* OutputP;

  InputP = new WriteWindowIT;
  InputP->WindowP = windowP;
  InputP->ImageSizeX = ImageFileTable[imageDescriptor]->ImageSizeX;
  InputP->ImageSizeY = ImageFileTable[imageDescriptor]->ImageSizeY;
  InputP->TileSizeX = ImageFileTable[imageDescriptor]->TileSizeX;
  InputP->TileSizeY = ImageFileTable[imageDescriptor]->TileSizeY;
  InputP->WindowSizeX = windowSizeX;
  InputP->WindowSizeY = windowSizeY;
  InputP->WindowOffsetX = windowOffsetX;
  InputP->WindowOffsetY = windowOffsetY;
  InputP->VirtualDiskIndex = ImageFileTable[imageDescriptor]->VirtualDiskIndex;
  InputP->ExtentFileDescriptor = ImageFileTable[imageDescriptor]->ExtentFileDescriptor;

  call Ps2Server.WriteWindow in InputP out OutputP; // Synchronously calls the parallel WriteWindow operation
  ErrorCode = OutputP->ErrorCode;
  delete OutputP;

  return ErrorCode;
} // end WriteWindow
```

Program 6-4. Incorporating the parallel *Ps2ServerT::WriteWindow* operation into a C/C++ image processing library

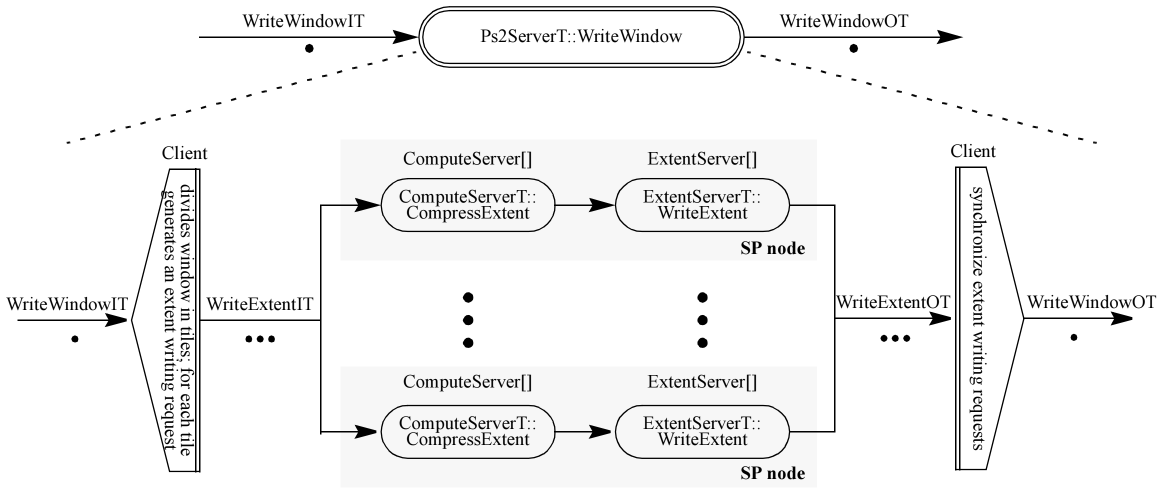

WriteWindowIT
Ps2ServerT::WriteWindow
WriteWindowOT
Client
ComputeServer[]
ExtentServer[]
Client
ComputeServerT:: CompressExtent
ExtentServerT:: WriteExtent
SP node
divides window in tiles; for each tile generates an extent writing request
synchronize extent writing requests
WriteWindowIT
WriteExtentIT
WriteExtentOT
WriteWindowOT
ComputeServer[]
ExtentServer[]
ComputeServerT:: CompressExtent
ExtentServerT:: WriteExtent
SP node

Figure 6-5. Thanks to the CAP formalism is very easy to incorporate a processing stage within the writing pipeline, e.g. an extent compression operation

- compression of extents is done in parallel by several processors; the number of processors per SP node and the number of SP nodes can be increased to improve the extent compression performance.

The CAP specification of the pipelined parallel *Ps2ServerT::WriteWindow* operation corresponding to the DAG of Figure 6-5 is shown in Program 6-5. At lines 1-3, thanks to the CAP additional operation declaration (Program 3-7), we elegantly extend the functionalities of the *ComputeServer* threads by declaring an additional library-specific extent compression operation. The C/C++ definition of the sequential *ComputeServerT::CompressExtent* operation is provided at line 5. Finally, this custom processing operation is efficiently incorporated into the writing pipeline of the parallel *Ps2ServerT::WriteWindow* operation at lines 19-20. Thanks to the CAP formalism, a minor modification of Program 6-3 enables the new enhanced *Ps2ServerT::WriteWindow* operation (Figure 6-5 and Program 6-5) to be generated.

```
 1 leaf operation ComputeServerT::CompressExtent  // Extending the functionalities of the ComputeServer threads:
 2   in WriteExtentIT* InputP                     // declaring an additional sequential operation
 3   out WriteExtentIT* OutputP;
 4
 1 leaf operation ComputeServerT::CompressExtent  // Definition of the additional sequential operation
 2   in WriteExtentIT* InputP
 3   out WriteExtentIT* OutputP
 4 {
 5   // C/C++ code (compresses the extent)
 6 } // end leaf operation ComputeServerT::CompressExtent
 7
 8 operation ps2ServerT::WriteWindow
 9   in WriteWindowIT* InputP
10   out WriteWindowOT* OutputP
11 {
12   flow_control(100)
13   indexed
14     (int TileRelativePositionY = 0; TileRelativePositionY < thisTokenP->WindowSizeY; TileRelativePositionY++ )
15     (int TileRelativePositionX = 0; TileRelativePositionX < thisTokenP->WindowSizeX; TileRelativePositionX++ )
16   parallel
17     (GenerateExtentWritingRequest, MergeWriteExtentAnswer, Client, local WriteWindowOT Result() )
18   (
19     ComputeServer[ToComputeServerIndex(thisTokenP->VirtualDiskIndex)].CompressExtent
20     >->
21     VirtualDiskServer[ToVirtualDiskServerIndex(thisTokenP->VirtualDiskIndex)].ExtentServer.WriteExtent
22   ); // end indexed parallel
23 } // end operation ps2ServerT::WriteWindow
```

Program 6-5. Thanks to the CAP formalism, a minor modification of Program 6-3 enables compressed tiles to be written

In Program 6-5, the selection of the *ComputeServer* thread performing the extent compression operation (line 19) is based on the *ToComputeServerIndex* macro (Program 5-4). The *ToComputeServerIndex* macro ensures that the selected *ComputeServer* thread performing the extent compression at line 19 is in the same address space as the *ExtentServer* thread writing the compressed extent at line 21 (Section 5.7).

Programs 6-3 and 6-5 feature a major drawback. Numerous extent writing answers, i.e. *WriteExtentOT* output tokens, have to be sent from the SP nodes to the client PC creating an unnecessary load on the network interfaces and processors of both the client and the SP nodes (although the *WriteExtentOT* tokens are tiny). A solution reducing the amount of messages transferred between the SP nodes and the client consists in performing a partial merge on each SP node and to send only a single extent writing answer to the client per SP node. Again, thanks to CAP, a minor modification of Program 6-5 enables the programmer to generate the new optimized parallel *PS2ServerT::WriteWindow* operation (Figure 6-6 and Program 6-6).

Comparing the first version in Program 6-5 with the improved specification of Program 6-6[1], only 4 lines (lines 25-28) have been added to the pipelined-parallel *Ps2ServerT::WriteWindow* operation. For each *ComputeServer* thread located on a different SP node (line 26), the *WriteWindowIT* request is duplicated by the *DuplicateWriteWindowRequest* routine and then split into slice writing requests by the *GenerateWritingRequest* routine as in the first version of the program (Program 6-5). Since there is a *GenerateWritingRequest* split routine

1. For CAP specialists. In Program 6-6 note the keyword '*local*' at line 34 instructing the CAP runtime system to initialize the *indexed parallel* CAP construct's output token in the current address space, i.e. in the client address space, and to send the *WriteWindowOT* output token in the address space where the merge routine is executed, i.e. in the *ComputeServer[ComputeServerIndex]* thread's address space (line 34). On the other hand, if we had used the keyword '*remote*', the *indexed parallel* CAP construct's input token, i.e. the *WriteWindowIT* input token (line 22), would have been sent to the *ComputeServer[ComputeServerIndex]* thread's address space for initializing the *indexed parallel* CAP construct's output token leading to transfer an unnecessary large amount of data, i.e. the window data buffer (Program 6-2, *WriteWindowIT* token), from the client address space to the *ComputeServer* thread address spaces.

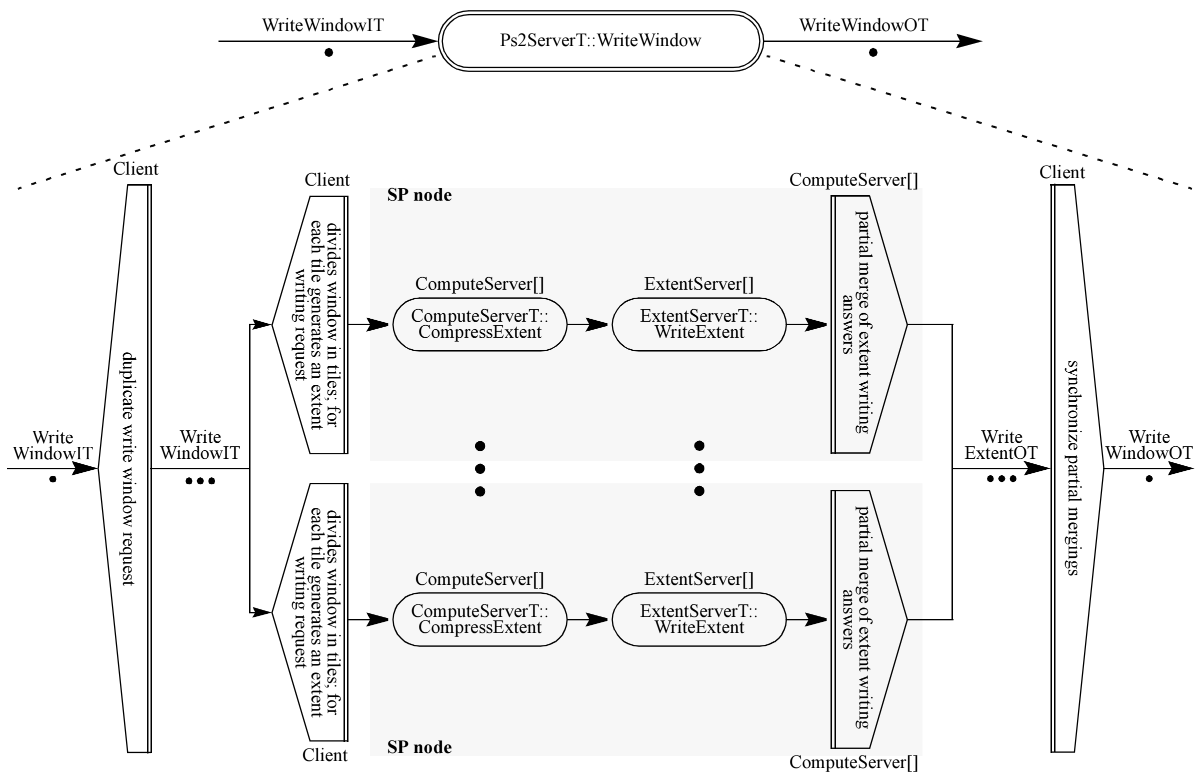

WriteWindowIT
Ps2ServerT::WriteWindow
WriteWindowOT
Client
duplicate write window request
Write WindowIT
Client
divides window in tiles; for each tile generates an extent writing request
SP node
ComputeServer[]
ComputeServerT:: CompressExtent
ExtentServer[]
ExtentServerT:: WriteExtent
ComputeServer[]
partial merge of extent writing answers
Write ExtentOT
Client
synchronize partial mergings
Write WindowOT

Figure 6-6. Graphical representation of the improved parallel *PS2ServerT::WriteWindow* operation requiring much less communication between the SP nodes and the client node

```
void GenerateExtentWritingRequest(WriteImageIT* inputP,
                                  WriteExtentIT* &subtokenP,
                                  int TileRelativePositionY,
                                  int TileRelativePositionX,
                                  int ComputeServerIndex)
{
  int ExtentFileIndex, ExtentIndex;

  DeclusteringFunction( ... );

  // Only generates an extent writing request for that particular compute server thread
  if ( ToComputeServerIndex(inputP->VirtualDiskIndex[ExtentFileIndex]) == ComputeServerIndex )
  {
    subtokenP = new WriteExtentIT;
    ...
  } else {
    subtokenP = (WriteExtentIT*) NULL;
  } // end if
} // end GenerateExtentWritingRequest

operation ps2ServerT::WriteWindow
  in WriteWindowIT* InputP
  out WriteWindowOT* OutputP
{
  indexed
    (int ComputeServerIndex = 0; ComputeServerIndex < NumberOfComputeServers; ComputeServerIndex++ )
  parallel (DuplicateWriteWindowRequest, SynchronizePartialMerging, Client, local WriteWindowOT Result() )
  (
    flow_control(3, 10)
    indexed
      (int TileRelativePositionY = 0; TileRelativePositionY < thisTokenP->WindowSizeY; TileRelativePositionY++ )
      (int TileRelativePositionX = 0; TileRelativePositionX < thisTokenP->WindowSizeX; TileRelativePositionX++ )
    parallel
      (GenerateWritingRequest, MergeWriteAnswer, ComputeServer[ComputeServerIndex], local WriteWindowOT Result() )
     (
      ComputeServer[ComputeServerIndex].CompressExtent
      >->
      VirtualDiskServer[ToVirtualDiskServerIndex(thisTokenP->VirtualDiskIndex)].ExtentServer.WriteExtent
    ) // end indexed parallel
  ); // end indexed parallel
} // end operation ps2ServerT::WriteWindow
```

Program 6-6. Improved CAP specification requiring less communication between the SP nodes and the client node

dividing the 2D window in tiles per *ComputeServer* thread, this split routine should only generate tile writing requests for tiles that are written on the virtual disks hooked on the SP node where the *ComputeServer* thread runs (line 12).

6.2.3 Neighbourhood dependent imaging operations

This section describes how the PS^2 parallel storage and processing server can be customized for developing neighbourhood dependent operations, i.e. a pipelined-parallel operations requiring to fetch the 8 borders of a tile for processing that tile. For example, 2D image filtering operations are neighbourhood dependent operations.

We consider the situation where the source image is declustered across multiple virtual disks using the strategy described in Section 6.2.1 and where the virtual disks are evenly distributed between the SP nodes. The target image, i.e. the result of the neighbourhood dependent image processing operation, is written to disks using exactly the same distribution and the same virtual disks. Before filtering can be performed on a tile, tile sides must be fetched, i.e. a tile must receive pixels from its 8 neighbouring tiles. The width of the fetched borders is defined in the border reading request and depends on the filtering operation (Figure 6-7).

The declustering strategy in Section 6.2.1 (Equations 6-2 and 6-3) ensures a proper load-balancing, i.e. disk and SP node accesses are close to uniformly distributed. Figure 6-7 shows the distribution of tiles across 4 virtual disks (*ExtentFileIndex*). Since consecutive virtual disks are mapped to physical disks hooked on different SP nodes, adjacent tiles on a same row are processed by different SP nodes.

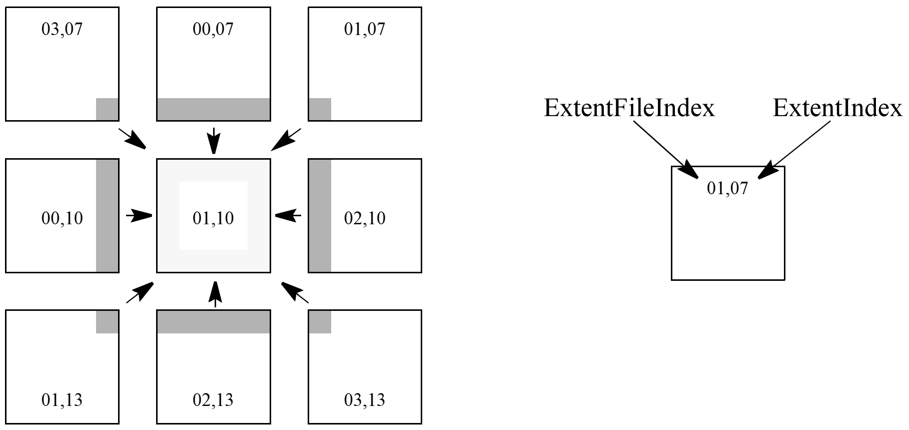


Figure 6-7. During a neighbourhood dependent imaging operation the borders of a tile are fetched from the neighbouring SP nodes

In the proposed execution schedule, all SP nodes work in parallel, i.e. each SP node independently filters tiles residing on its local virtual disks. Tiles are scanned as a serpentine[1] so as to make a better use of the extent caches [Gennart98a]. Each SP node performs a four-step pipeline for filtering a particular tile. The first step consists of reading the tile from a local virtual disk (or from the corresponding local extent cache) and, in parallel, the 8 neighbouring tile's borders from the other SP nodes. During the second pipeline step, the SP node computes the central part of the tile. During the third pipeline step, the SP node computes the border of the tile after having received the 8 neighbouring tile's borders. During the fourth pipeline step, the SP node writes the computed tile back to the virtual disk. The tile central part is defined as the part of the tile that is not affected by the neighbouring tile sides. The tile border is defined as the part of the tile that is affected by the 8 neighbouring tile sides. This four-step pipeline is repeatedly performed on all tiles in a pipelined manner.

Figure 6-8 shows the graphical CAP specification of the parallel *Ps2ServerT::NeighbourhoodDependent* image processing operation. The process window request, i.e. the *ProcessImageIT* input token, is first duplicated by the *Client* thread and sent to all SP nodes, i.e. *ComputeServer* threads. Each SP node iteratively traverses its tiles, i.e. the tiles intersecting the image window and residing on the local virtual disks, and for each tile generates a tile processing request. This request is redirected to the parallel *PS2ServerT::ProcessTile* operation where the tile and its 8 borders are read and filtered. The *Ps2ServerT::ProcessTile*'s output token, i.e. the filtered tile, is redirected to the *ExtentServerT::WriteExtent* operation writing the tile back to the target image file. Once all the tiles making

1. Serpentine scanning: left to right on first row, right to left on second row, left to right on third row, etc.

up the window part of a SP node are filtered, an answer is sent back to the *Client* thread comprising an error code, i.e. a *WriteExtentOT* output token. When the *Client* thread has received an acknowledge from all the SP nodes, i.e. the full 2D image has been filtered, the *ProcessImageOT* output token is passed to the next operation.

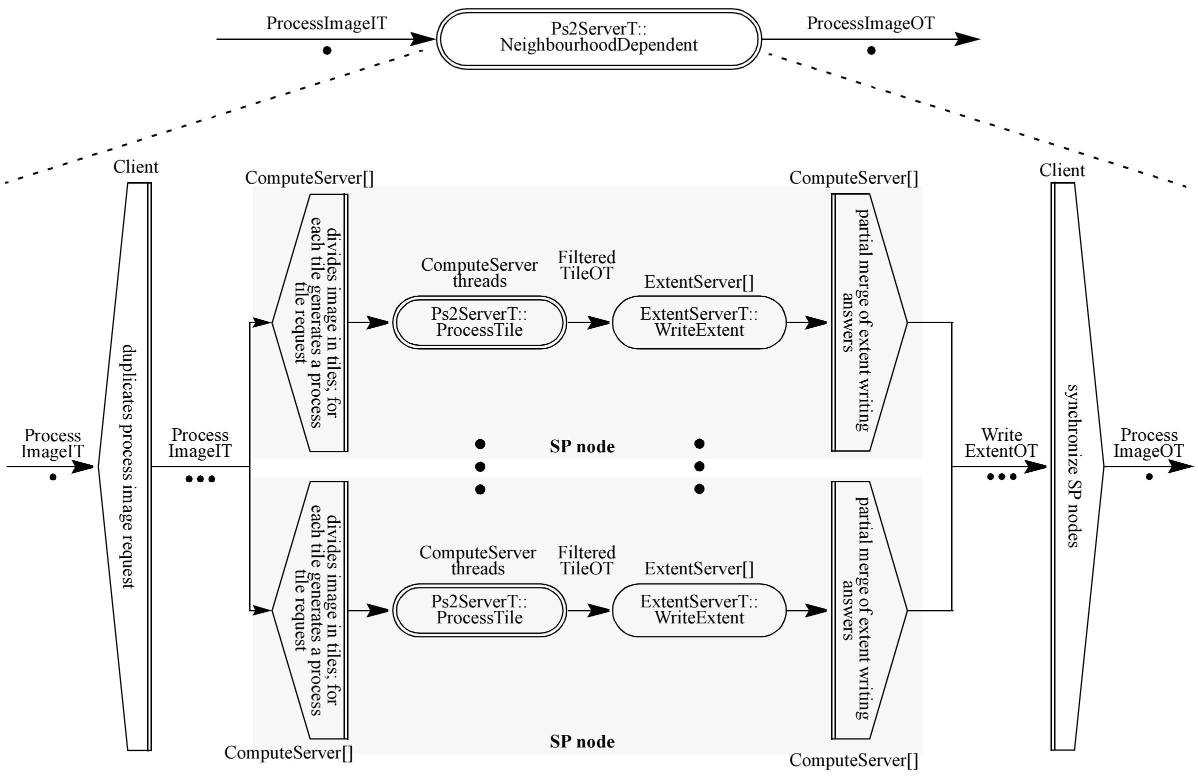


Figure 6-8. Graphical CAP specification of the neighbourhood dependent imaging operation

In the DAG of Figure 6-8, pipelining is achieved at two levels:

- the *ExtentServer* thread writes a filtered tile while its companion *ComputeServer* thread located on the same SP node processes the next tiles,
- the *ComputeServer* thread processes a tile[1], while the same *ComputeServer* thread continues to divide its image part into tiles and to generate tile processing requests.

And the parallelization occurs at two levels:

- several tiles are read and written simultaneously to and from different disks; the number of disks can be increased to improve I/O throughput,
- tile filtering is done in parallel by several processors; the number of processors per SP node (processors sharing a common memory) and the number of SP nodes can be increased to improve the tile filtering performance.

Figure 6-9 shows the graphical CAP specification of the parallel *Ps2ServerT::ProcessTile* operation. This pipelined-parallel operation consists in reading in parallel the tile and its 8 border tiles from multiple virtual disks. Once a border tile is read (*if* CAP construct) by an *ExtentServer* thread, its companion *ComputeServer* thread, i.e. the *ComputeServer* thread residing in the same address space as the *ExtentServer* thread that read the border tile, extracts the required border from the tile using the *ComputeServerT::ExtractBorderFromTile* sequential operation. Then, the 8 borders are possibly transfered across the network (from one SP node to another SP node) to the *ComputeServer* thread where the central tile resides. Finally, the central tile and its 8 borders are filtered by the *ComputeServer* thread using the *ComputeServerT::FilterTile* sequential operation. Once the tile is filtered, an extent writing request (*WriteExtentIT* token) is generated and redirected to the next operation. Remark that during the whole process the central tile is never transfered across the network. Only the 8 borders are possibly transfered from the neighbouring SP nodes to the SP node where the central tile resides.

1. Remember that processing a tile involves reading asynchronously the tile and its 8 borders from other SP nodes.

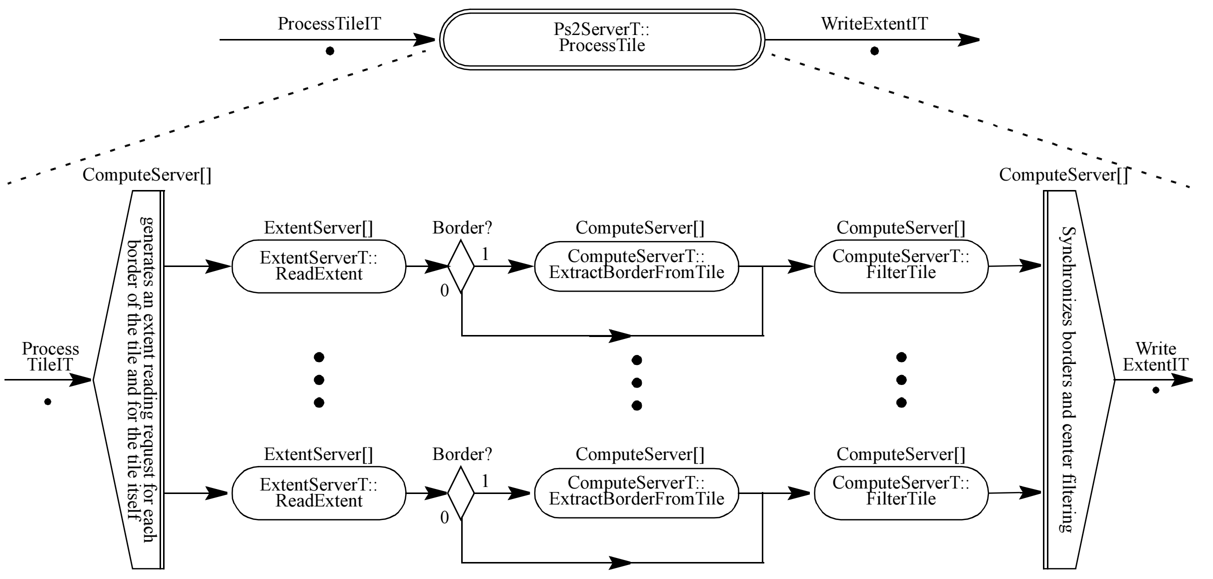

ProcessTileIT
Ps2ServerT:: ProcessTile
WriteExtentIT
ComputeServer[]
ComputeServer[]
generates an extent reading request for each border of the tile and for the tile itself
Process TileIT
ExtentServer[]
ExtentServerT:: ReadExtent
Border?
1
0
ComputeServer[]
ComputeServerT:: ExtractBorderFromTile
ComputeServer[]
ComputeServerT:: FilterTile
Synchronizes borders and center filtering
Write ExtentIT

Figure 6-9. Graphical CAP specification of the parallel *Ps2ServerT::ProcessTile* operation

Program 6-7 shows the CAP specification of the pipelined-parallel *Ps2ServerT::NeighbourhoodDependent* operation (lines 25-51). The input token is an image description (parallel image file descriptor, image size, tile size) and a filter description (filter size, filter values) but no data. The output token merely comprises an error code, since the filtered image is directly stored on the virtual disks without producing any output. The first *indexed parallel* CAP construct (lines 29-31) duplicates the image processing request for each *ComputeServer* threads (line 30) located on different SP nodes. Then, each *ComputeServer* thread (line 33) independently and in parallel performs the second *indexed parallel* CAP construct that scans tiles of the 2D image in serpentine (lines 37-42). The first loop (lines 37-39) iterates on successive image tile rows and the second loop (lines 40-42) iterates on all tiles of one input image tile row. For each local tile, i.e. for tiles stored on virtual disks hooked on the SP node where the *ComputeServer[ComputeServerIndex]* thread at line 33 resides, a tile processing request is generated by the *GenerateTileProcessingRequest* routine which is redirected to the parallel *Ps2ServerT::ProcessTile* operation. The output of the *Ps2ServerT::ProcessTile* operation (line 46), which is a filtered tile, is redirected to the *ExtentServerT::WriteExtent* operation (line 48) performed by the companion *ExtentServer* thread residing in the same address space as the *ComputeServer[ComputeServerIndex]* thread (line 33).

The parallel *Ps2ServerT::ProcessTile* operation is made of a single *indexed parallel* CAP construct (lines 9-12) that generates a border processing request (*GenerateBorderProcessingRequest* routine) for the central tile and its 8 borders (line 10). The 9 processing requests are redirected to the appropriate *ExtentServer* threads reading in parallel the 9 tiles (line 14). Then, if it is a border (line 16), the border is extracted from the read tile (line 18) by the companion *ComputeServer* thread, i.e. the *ComputeServer* thread residing in the same address space as the *ExtentServer* thread that read the tile at line 14. The extracted borders and the central tile are redirected (and possibly transfered across the network) to the *ComputeServer* thread (line 21) who initiates the parallel *Ps2ServerT::ProcessTile* operation, i.e. where resides the central tile. The *ComputeServerT::FilterTile* sequential operation is performed on the central tile and its 8 borders. Finally, once the 9 processing requests are merged by the *MergeBorderProcessingAnswer* routine, i.e. the entire tile is filtered, the extent writing request, i.e. the *WriteExtentIT* output token (line 7), is redirected to the next operation.

By being able to direct at execution time the *ExtentServerT::ReadExtent* and *ComputeServerT::ExtractBorderFromTile* operations to the SP node where the tiles reside, operations are performed only on local data and superfluous data communications over the network are completely avoided.

6.3 The Visible Human slice server application

This section demonstrates the applicability and the performances of the CAP tool and the PS^2 framework on real applications with a parallel 3D image slice extraction server application. This application, developed using the PS^2 methodology, enables clients to specify, access in parallel, and visualize image slices having any desired

```
enum NeighbourhoodT {CENTER, NORTH_EAST, NORTH, NORTH_WEST, EAST, WEAST, SOUTH_EAST, SOUTH, SOUTH_WEST };

operation ps2ServerT::ProcessTile(int computeServerIndex,
                                  int tilePositionX,
                                  int tilePositionY)
  in ProcessTileIT* InputP
  out WriteExtentIT* OutputP
{
  indexed
    ( NeighbourhoodT Neighbour = CENTER; Neighbour >= SOUTH_WEST; Neighbour++ )
  parallel ( GenerateBorderProcessingRequest, MergeBorderProcessingAnswer(tilePositionX, tilePositionY),
             ComputeServer[computeServerIndex], local WriteExtentIT Result(*thisTokenP) )
  (
    VirtualDiskServer[ToVirtualDiskServerIndex(thisTokenP->VirtualDiskIndex)].ExtentServer.ReadExtent
    >->
    if ( Neighbour != CENTER )
    (
      ComputeServer[SelectComputeServer(tilePositionX, tilePositionY, Neighbour)].ExtractBorderFromTile(Neighbour)
    ) // end if
    >->
    ComputeServer[computeServerIndex].FilterTile(Neighbour)
  ); // end indexed parallel
} // end operation ps2ServerT::ProcessTile

operation ps2ServerT::NeighbourhoodDependent
  in ProcessImageIT* InputP
  out ProcessImageOT* OutputP
{
  indexed
    ( int ComputeServerIndex = 0; ComputeServerIndex < NumberOfComputeServers; ComputeServerIndex++ )
  parallel (DuplicateProcessImageRequest, SynchronizePartialMerging, Client, local ProcessImageOT Result() )
  (
    ComputeServer[ComputeServerIndex].{ }  // Redirect a process image request to each SP node
    >->
    flow_control(3, 10)
    indexed // doubly indexed
      ( int TilePositionY = FirstTilePositionY(thisTokenP);
        IsNotLastRow(TilePositionY, thisTokenP);
        TilePositionY++ )
      ( int TilePositionX = FirstTilePositionX(TilePositionY, thisTokenP);
        IsNotLastTileInRow(TilePositionX, TilePositionY, thisTokenP);
        TilePositionX = NextTilePositionX(TilePositionX, TilePositionY) )
    parallel ( GenerateTileProcessingRequest(ComputeServerIndex), MergeTileProcessingAnswer,
               ComputeServer[ComputeServerIndex], local ProcessImageOT Result() )
    (
      ProcessTile(ComputeServerIndex, TilePositionX, TilePositionY)
      >->
      VirtualDiskServer[ToVirtualDiskServerIndex(thisTokenP->VirtualDiskIndex)].ExtentServer.WriteExtent
    ) // end indexed parallel
  ); // end indexed parallel
} // end operation ps2ServerT::NeighbourhoodDependent
```

Program 6-7. CAP specification of the neighbourhood dependent imaging operation

position and orientation. The image slices are extracted from the 3D Visible Human Male declustered across multiple disks distributed on several SP nodes. Users located over the local area network interact with the underlying parallel image slice extraction server with a graphical interface developed in Borland Delphi. Moreover, in order to demonstrate the applicability of our tools (CAP and PS^2) to build parallel Web server on distributed memory PC's, we have interfaced this parallel image slice extraction engine to a Microsoft IIS Web server using the ISAPI protocol. A Java 1.1 applet runs on Web clients and enables users to specify slice position and orientation and generate image slice extraction requests. Replies of the Web server are compressed using the JPEG standard and send back to the Web client for display. The Web interface is operational at *http://visiblehuman.epfl.ch* [Vetsch98].

Such complex client parallel-server applications require several software components developed using different tools and technologies (CAP, PS^2, Microsoft Visual C/C++, DLL libraries, Borland Delphi, ISAPI protocol, Borland Java Builder, Microsoft IIS Web server) that must be properly assembled. For the Windows NT application, the parallel image slice extraction engine, developed using the PS^2 framework, has been interfaced to a Borland Delphi graphical interface using a dynamic linked library (DLL). For the Web application, the parallel image slice extraction engine (a Microsoft Visual C/C++ DLL library) has been interfaced to the Microsoft IIS Web server using an ISAPI dynamic linked library developed in Borland Delphi. This demonstrates the ability of PS^2-generated programs to be incorporated in a wide variety of applications.

Section 6.3.1 introduces the image slice extraction and visualization application. It presents the 14GByte Visible Human Male and how it is striped between multiple disks using the declustering strategy of Section 6.2.1. Section 6.3.2 presents the parallelization of the extraction algorithm using PS^2. In order to demonstrate the flexibility of CAP, we present a first naive solution featuring superfluous communications. Then, we easily modify the schedule of the first solution leading to an application requiring much less data transfer. Section 6.3.3

experimentally analyses the performances of the image slice extraction and visualization application for various PS^2 server configurations and demonstrates that the obtained image slice throughputs are close to the performances of the underlying hardware.

6.3.1 Description of the image slice extraction and visualization application

The 24-bit Visible Human Male created by the National Library of Medicine at Bethesda Maryland USA [Ackerman95, Spitzer96] reaches a size of 2048x1216x1878 voxels, i.e. 13 GBytes RGB. The width (X) and height (Y) resolutions are 3 pixels per millimetre. The axial anatomical images (Z) where obtained at 1.0 millimetre intervals. For enabling parallel storage and access, the 3D Visible Human Male data set is segmented into 3D volumic extents of size 32x32x17 voxels, i.e. 51 KBytes, distributed over a number of disks.

The distribution of volumic extents to disks is made so as to ensure that direct volumic extent neighbours reside on different disks hooked on different SP nodes. We achieve such a distribution by extending the declustering strategy presented in section 6.2.1 to a third dimension (Z), i.e. by introducing between two successive planes of volumic extents an extent file offset ($ExtentFileOffset_Z$) which is also prime to the number of extent files (*NExtentFiles*) making up the parallel 3D image file. This ensures that for nearly all extracted slices, disk and SP node accesses are close to uniformly distributed. Equation 6-4 gives the distribution of volumic extents to disks where $ExtentPosition_X$, $ExtentPosition_Y$ and $ExtentPosition_Z$ are the volumic extent coordinate, and $NExtent_X$ and $NExtent_Y$ are the numbers of volumic extents per row and per column. Equation 6-5 gives the distribution of volumic extents within a particular disk, or extent file, computed using Equation 6-4 (all divisions are integer divisions). Therefore a particular volumic extent is uniquely identified with its *ExtentFileIndex* specifying on which extent file the extent resides and its *ExtentIndex* specifying the local extent index within the set of extents stored on that particular extent file.

$$\begin{aligned} ExtentFileIndex = {} & (ExtentPosition_Z \cdot \mathrm{ExtentFileOffset_Z} + ExtentPosition_Y \cdot \mathrm{ExtentFileOffset_Y} + \\ & ExtentPosition_X) \ \mathrm{mod} \ \ \mathrm{NExtentFiles} \end{aligned} \tag{6-4}$$

$$\begin{aligned} ExtentIndex = {} & ExtentPosition_Z \cdot \left[\mathrm{NExtents_Y} \cdot \left(1 + \frac{\mathrm{NExtents_X} - 1}{\mathrm{NExtentFiles}}\right)\right] + \\ & ExtentPosition_Y \cdot \left(1 + \frac{\mathrm{NExtents_X} - 1}{\mathrm{NExtentFiles}}\right) + \frac{ExtentPosition_X}{\mathrm{NExtentFiles}} \end{aligned} \tag{6-5}$$

Visualization of 3D medical images by slicing, i.e. by intersecting a 3D tomographic image with a plane having any desired position and orientation is a tool of choice in diagnosis and treatment. In order to extract a slice from the 3D image, the volumic extents intersecting the slice are read from the disks and the slice parts contained in these volumic extents are extracted and resampled (Figure 6-10).

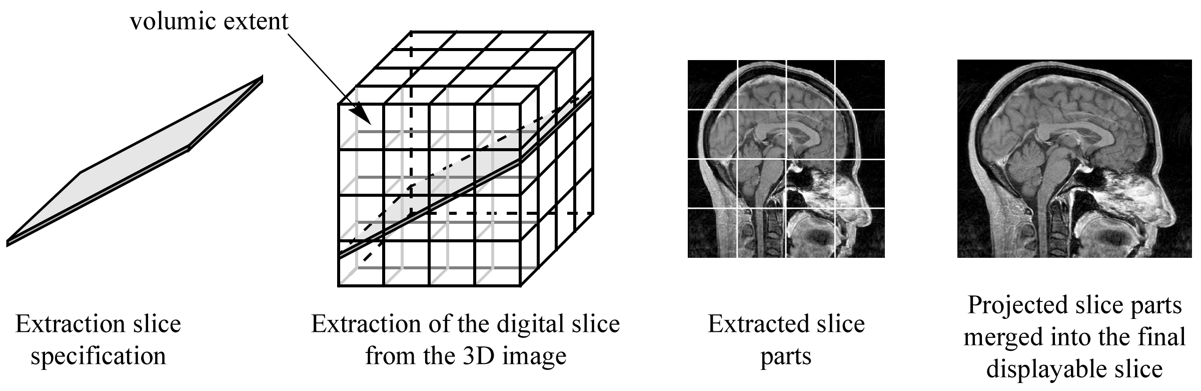

volumic extent
Extraction slice specification
Extraction of the digital slice from the 3D image
Extracted slice parts
Projected slice parts merged into the final displayable slice

Figure 6-10. Extraction of slice parts from volumic file extents

Since the resolution along the Z-axis is three times lower than the resolution in X and Y, non isometric 3D volume interpolation is applied for the extraction of slice parts. To avoid non-isometric interpolation across extent boundaries, i.e. dependences between interpolation operations which after parallelization, may be executed in different address spaces, the volumic extents we store on disks have overlapping boundaries in Z-direction, i.e. vertically adjacent extents have one axial anatomical image in common.

The sequential algorithm of the extraction of image slices from 3D tomographic images has been created by Oscar Figueiredo based on his research on discrete planes [Figueiredo99].

Users are asked to specify a slice located at a given position within a 3D image and having a given orientation. The client's user interface displays enough information (a miniaturized version of the 3D image, as in [North96]) to enable the user to interactively specify the desired image slice (Figure 6-11).

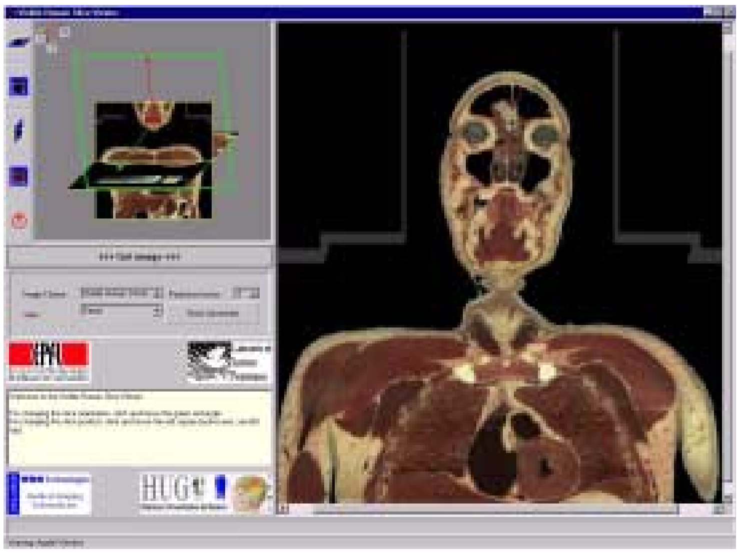


Figure 6-11. Selecting an image slice within a 3D tomographic image

The parameters defining the user selection are sent to the server application. The server application consists of a proxy residing on the client's site and of server processes running on the server's parallel processors. The proxy interprets the slice location and orientation parameters and determines the image extents which need to be accessed. It sends to the concerned servers (servers whose disks contain the required extents) the extent reading and image slice extraction and projection requests. These servers execute the requests and transfer the resulting slice parts to the server proxy residing on the client site, which assembles them into the final displayable image slice (Figure 6-12).

6.3.2 Parallelizing the image slice extraction and visualization application using PS^2

This section shows how the image slice extraction and visualization operation has been parallelized using the CAP Computer-Aided Parallelization tool and how it executes on a multi-PC environment using a configuration map specifying the layout of threads on the available PC’s.

The first step when creating a pipelined-parallel operation using CAP is to devise its macro data flow, i.e. to decompose the problem into a set of sequential operations, and to specify the operations input and output data types (tokens). This enables the application to be decomposed into basic independent sub-tasks that may be executed in a pipelined-parallel manner.

From Section 6.3.1, we identify the five basic independent sequential sub-tasks that compose the slice extraction and visualization application:

- compute the extents intersecting the slice from the slice orientation parameters and from the 3D image file distribution parameters,
- read an extent from a single disk (a predefined CAP operation available in a library of reusable components),
- extract a slice part from an extent and project it onto the display space (possibly combined with zoom),

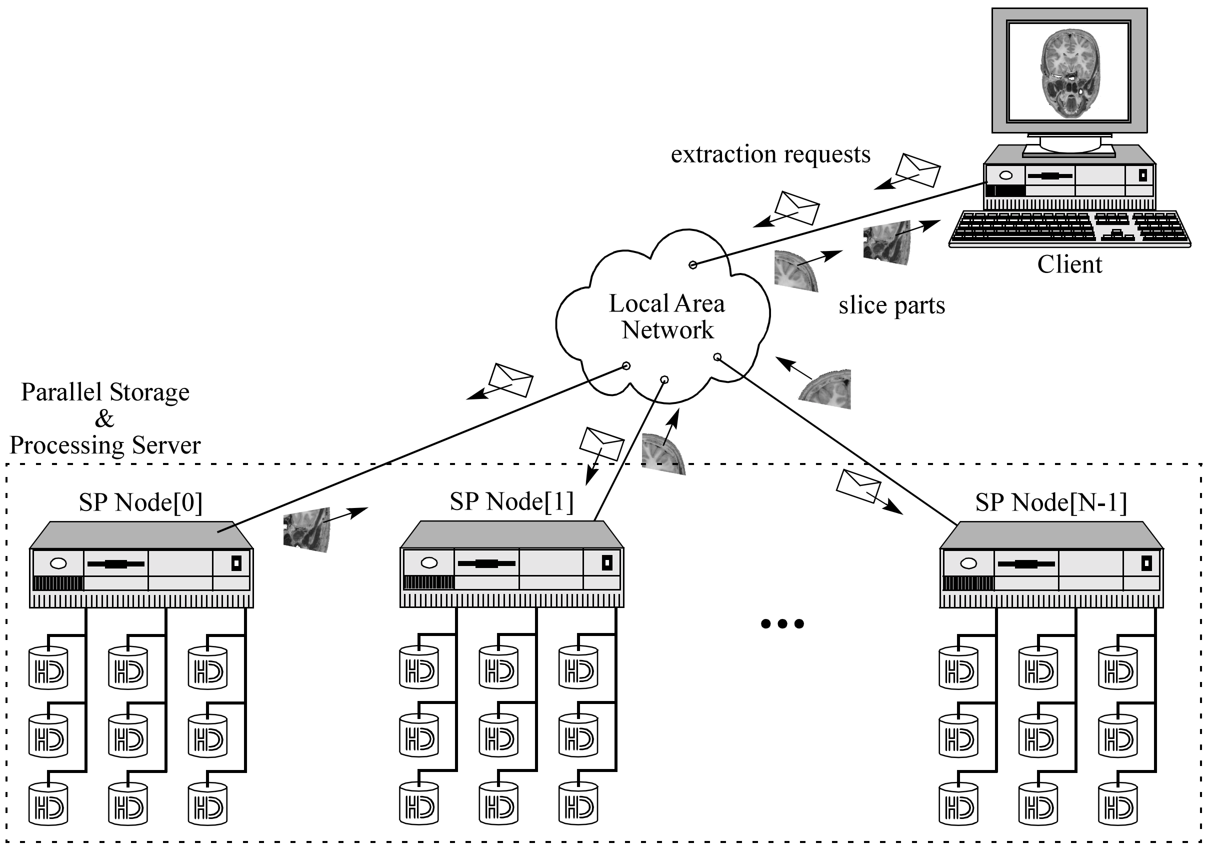


Figure 6-12. Sending the extraction requests and receiving the slice parts

- merge the extracted and projected slice parts into the full image slice,
- display the extracted full slice on the client computer.

The macro data flow specifying the schedule of these five basic operations is shown in Figure 6-13.

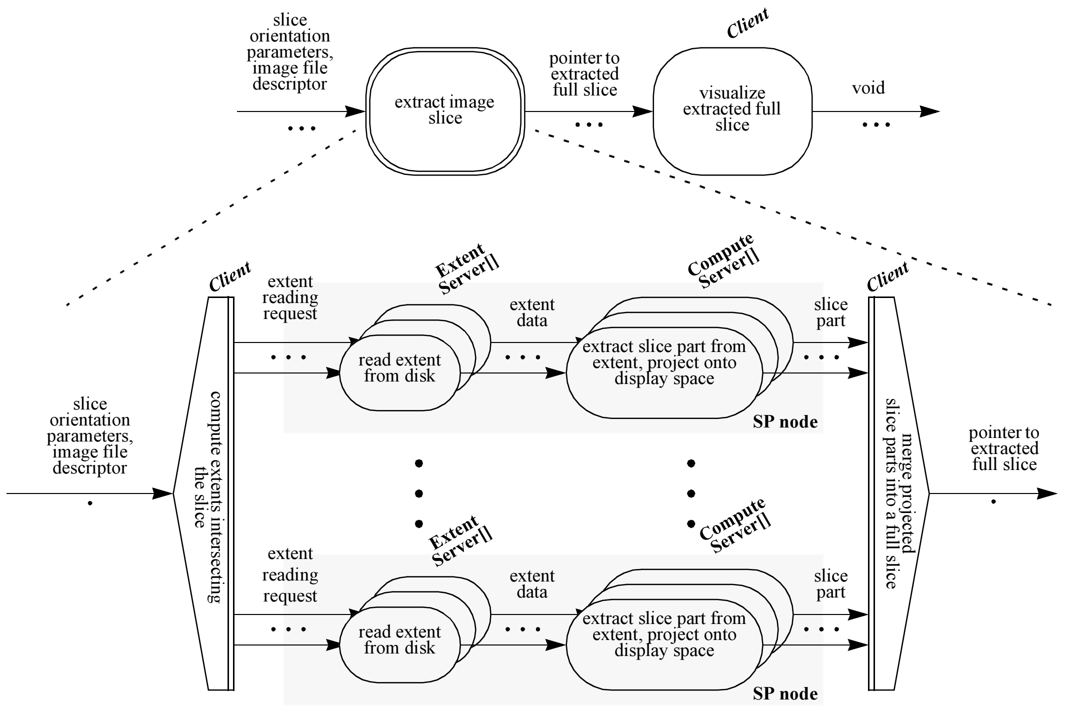

slice orientation parameters, image file descriptor
extract image slice
pointer to extracted full slice
Client
visualize extracted full slice
void
Client
compute extents intersecting the slice
extent reading request
Extent Server[]
read extent from disk
extent data
Compute Server[]
extract slice part from extent, project onto display space
slice part
SP node
Client
merge projected slice parts into a full slice
pointer to extracted full slice

Figure 6-13. Graphical representation of the pipelined-parallel slice extraction and visualization operation

The pipelined-parallel slice extraction and visualization operation is nothing else than the pipelining of the parallel *ExtractSlice* operation and the sequential *VisualizeSlice* operation (Figure 6-13). The input token of the first stage is a *SliceExtractionRequest* comprising the slice orientation parameters and the 3D tomographic image file descriptor. The output token is the extracted and projected *Slice* which is the input token of the second stage executed by the *Client* thread.

The *ExtractSlice* operation is defined as follows. Based on the *SliceExtractionRequest*, the *Client* thread first computes all the extents intersecting the slice. For each of these extents a reading request is sent to an *ExtentServer[]* located on the SP node where this extent resides[1]. Once an extent is read by the asynchronous *ReadExtent* operation, it is fed to the *ExtractAndProjectSlicePart* operation performed by a *ComputeServer[]* thread residing in the same computer as the *ExtentServer* thread. The resulting extracted and projected slice part is sent back to the *Client* thread to be merged into the full image slice. When all the slice parts are merged, the full image *Slice* is passed to the next operation.

Pipelining is achieved at four levels:

- slice extraction and projection is performed by the *ComputeServer* thread on one extent while the companion *ExtentServer* thread reads the next extents,
- an extracted and projected slice part is asynchronously sent to the *Client* thread while the *ComputeServer* thread extracts and projects the next slice parts,
- an extracted and projected slice part is merged by the *Client* thread into the full image slice while the next slice parts are being asynchronously received,
- a full image slice is displayed by the *Client* thread while the next full image slices are being prepared (in the case the user has requested a series of successive slices).

Parallelization occurs at two levels:

- several extents are read simultaneously from different disks; the number of disks can be increased to improve I/O throughput,
- extraction of slice parts from extents and projection operations are done in parallel by several processors; the number of processors can be increased to improve the slice part extraction and projection performance.

The CAP program of the pipelined-parallel slice extraction and visualization operation corresponding to the graph of Figure 6-13 is shown in Program 6-8.

The parallel *ExtractAndVisualizeSlice* operation (line 44) is decomposed into two sub operations *ExtractSlice* and *VisualizeSlice* that are executed in pipeline. The slice extraction request input argument comprises the slice orientation parameters and the 3D tomographic image file descriptor.

The input of the parallel *Ps2ServerT::ExtractSlice* operation (line 24) performed by the *Client* thread is a *SliceExtractionRequestT* request. Using a *parallel while* CAP construct, this request is divided by the *SplitSliceRequest* routine (line 13) that incrementally computes the extents intersecting the slice. Each time it is called, it generates an extent reading request and returns a boolean specifying that the current request is not the last request (end of the while loop). The *ExtentServer* threads with index *thisTokenP->ExtentServerIndex* running the *ReadExtent* operation (line 1) read the required 3D extents from the disks and feed the extent data to their companion *ComputeServer* threads. The *ComputeServer* threads running the *ExtractAndProjectSlicePart* (line 5) extract the slice parts from the received extents, project them into display space, and return them to the *Client* thread who originally started the operation. The *Client* thread merges the projected slice parts into a single *SliceT* full slice using the *MergeSlicePart* routine (line 20).

The sequential *ClientT::VisualizeSlice* operation (line 36) is a standard C/C++ sequential routine that displays the input 24-bit color bitmap on the screen using the Win32 application programming interface.

Each *ExtentServer* thread and its companion *ComputeServer* thread work in pipeline; multiple pairs of *ExtentServer* and *ComputeServer* threads may work in parallel if the configuration map specifies that different *ExtentServer*/*ComputeServer* threads are mapped onto different processes running on different computers.

1. When a parallel file is opened, the client thread obtains information how the global file is striped into subfiles and on which disk and processing node each subfile resides. This enables the index of the processing node whose disk contains the desired volumic file extent to be computed.

```
external leaf operation ExtentServerT::ReadExtent
  in ReadExtentIT* InputP
  out ReadExtentOT* OutputP;  // Taken from the library of reusable low-level parallel file system components

leaf operation ComputeServerT::ExtractAndProjectSlicePart
  in ReadExtentOT* InputP
  out SlicePartT* OutputP
{
  OutputP = new SlicePartT;
  ...C++ sequential code
}

bool SplitSliceRequest(SliceExtractionRequestT* FromP,
                       ReadExtentIT* PreviousP, ReadExtentIT* &ThisP) {
  ThisP = new ReadExtentIT;
  ...C++ sequential code
  return (IsNotLastExtentReadingRequest);
}

void MergeSlicePart(SliceT* IntoP, SlicePartT* ThisP) {
  ...C++ sequential code
}

operation Ps2ServerT::ExtractSlice
  in SliceExtractionRequestT* InputP
  out SliceT* OutputP
{
  parallel while (SplitSliceRequest, MergeSlicePart, Client, SliceT Output)
  (
    VirtualDiskServer[thisTokenP->VirtualDiskServerIndex].ExtentServer.ReadExtent
    >->
    ComputeServer[thisTokenP->ComputeServerIndex].ExtractAndProjectSlicePart
  );
}

leaf operation ClientT::VisualizeSlice
  in SliceT* InputP
  out capTokenT* OutputP
{
  OutputP = new capTokenT;
  ...C++ sequential code
}

operation Ps2ServerT::ExtractAndVisualizeSlice
  in SliceExtractionRequestT* InputP
  out capTokenT* OutputP
{
  ExtractSlice
  >->
  Client.VisualizeSlice;
}
```

split function

merge function

thread where *MergeSlicePart* is executed

output token of the parallel while construct

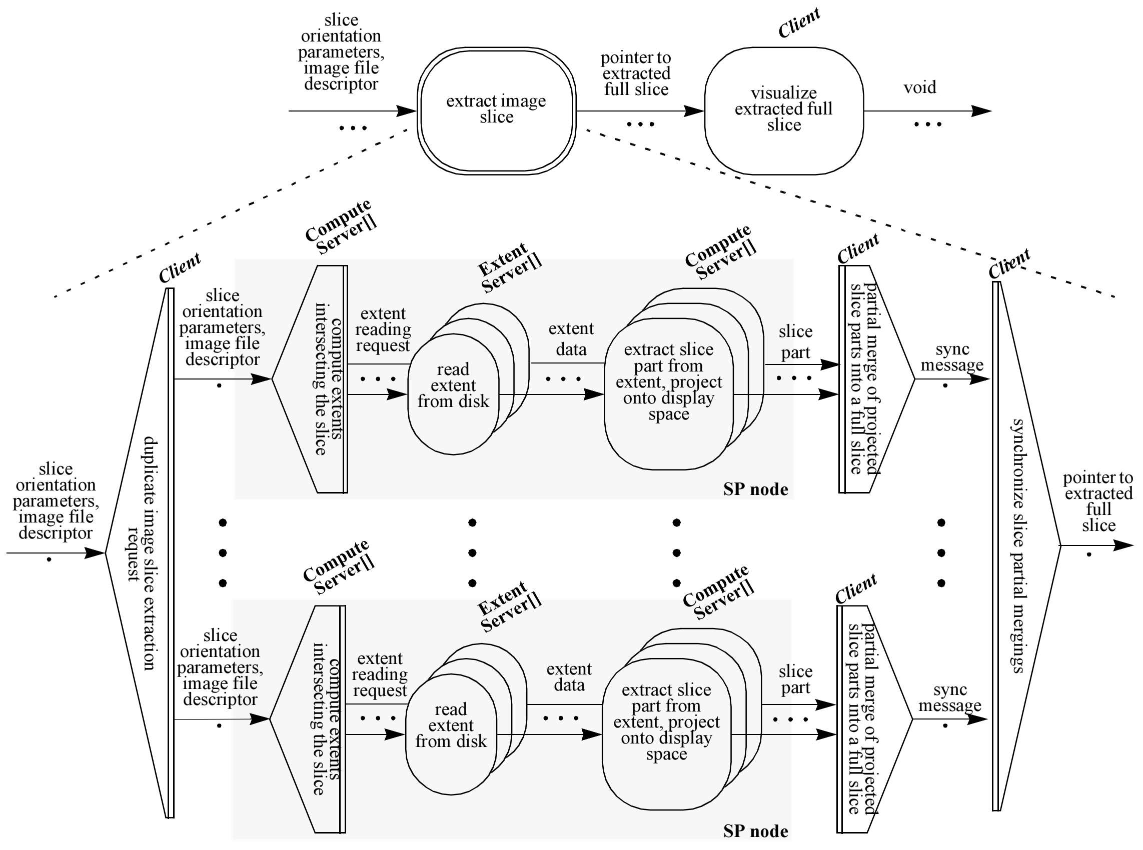
Program 6-8. CAP specification of the pipelined-parallel slice extraction and visualization operation

The program shown in Figure 6-13 features a major drawback. Numerous extent reading requests have to be sent from the client PC to the SP nodes creating an important load on both the client and server network interfaces and processors. To give a figure, with 5 SP nodes, each with one disk and an enabled extent cache, 4.8 512x512 image slices per second are visualized. Since each slice perpendicular to the volume's main diagonal intersects on average 437 volumic extents, extracting a slice requires on average 437 extent reading requests. The client PC must therefore sustain an output network throughput of 4.8 x 437 = 2'098 extent reading requests per second.

A solution reducing heavily the amount of messages transfered between the client and SP nodes consists in sending a full slice access request to all the SP nodes. The SP nodes segment themselves the slice access request into local extent reading requests. Thanks to CAP, a minor modification of the program shown in Figure 6-13 enables the new optimized application (Figure 6-14 and Program 6-9) to be generated.

Comparing the first version in Program 6-8 and the improved specification in Program 6-9, only 5 lines (line 13 through line 17) have been added to the pipelined-parallel *Ps2ServerT::ExtractSlice* operation. For each SP node (line 13), the *SliceExtractionRequestT* request is duplicated by the *DuplicateSliceExtractionRequest* routine (line 1) and sent to a *ComputeServer* thread located on that particular SP node (line 16) where the slice extraction request is divided into extent reading requests (lines 18 and 19) as in the first version of the program (Figure 6-8). Line 16 forces the *SliceExtractionRequestT* request to be sent from the client PC to a SP node where the *SplitSliceRequest* routine will be executed.

By being able to direct at execution time the *ExtentServerT::ReadExtent* and *ComputeServerT::ExtractAndProjectSlicePart* operations to the SP node where the extents resides, operations are performed only on local data and superfluous data communications over the network are completely avoided. Load-balancing is ensured by distributing volumic extents onto the disks according to Equations 6-4 and 6-5.



slice orientation parameters, image file descriptor
extract image slice
pointer to extracted full slice
Client
visualize extracted full slice
void
Client
duplicate image slice extraction request
slice orientation parameters, image file descriptor
Compute Server[]
compute extents intersecting the slice
extent reading request
Extent Server[]
read extent from disk
extent data
Compute Server[]
extract slice part from extent, project onto display space
slice part
Client
partial merge of projected slice parts into a full slice
sync message
Client
synchronize slice partial mergings
pointer to extracted full slice
SP node
SP node

Figure 6-14. Graphical representation of the improved operation requiring much less communication between the client PC and the SP nodes

```
void DuplicateSliceExtractionRequest(SliceExtractionRequestT* FromP,
                                     SliceExtractionRequestT* &ThisP,
                                     int ServerPC) {
  ThisP = new SliceExtractionRequesT(FromP);
}

void SynchronizeSPNodes(SliceT* IntoP, SliceT* ThisP) { ...C++ sequential code }

operation Ps2ServerT::ExtractSlice
  in SliceExtractionRequestT* InputP
  out SliceT* OutputP
{
  indexed (int SPNode = 0; SPNode < NSPNODES; SPNode++)
  parallel (DuplicateSliceExtractionRequest, SynchronizeSPNodes, Client, SliceT Output)
  (
    ComputeServer[ThisTokenP->ComputeServerIndex].{ }
    >->
    parallel
    while (SplitSliceRequest, MergeSlicePart, Client, SliceT Output)
    (
      VirtualDiskServer[ThisTokenP->VirtualDiskServerIndex].ExtentServer.ReadExtent
      >->
      ComputeServer[ThisTokenP->ComputeServerIndex].ExtractAndProjectSlicePart
    )
  );
}
```

Program 6-9. Improved CAP specification requiring much less communication between the client PC and the SP nodes

6.3.3 Performances and scalability analysis of the image slice extraction and visualization application

The server architecture we consider comprises 5 200MHz Bi-PentiumPro PC's interconnected by a 100 Mbits/s switched Fast Ethernet network (Figure 6-12). Each SP node runs the Windows NT Workstation 4.0 operating system, and incorporates 12 SCSI-2 disks divided into 4 groups of 3 disks, each hooked onto a separate SCSI-2 string. We use IBM-DPES 31080, IBM-DCAS 32160, CONNER CFP1080S, SEAGATE ST52160N and

SEAGATE ST32155N disks which have a measured mean read data transfer throughput of 3.5 MBytes/s and a mean latency time, i.e. seek time + rotational latency time, of 12.2 ms [Messerli97]. Thus, when accessing 51 KBytes blocks, i.e. 32x32x17 RGB extents, located at random disk locations, an effective throughput of 1.88 MBytes/s per disk is reached.

In addition to the SP nodes, one client 333MHz Bi-PentiumII PC located on the network runs the 3D tomographic image visualization task (Figure 6-12) which enables the user to specify interactively the desired slice access parameters (position and orientation) and interacts with the server proxy to extract the desired image slice. The server proxy running on the client sends the slice extraction requests to the SP nodes, receives the slice parts and merges them into the final image slice, which is passed to the 3D tomographic image visualization task to be displayed (Figure 6-11).

The present application comprises several potential bottlenecks: insufficient parallel disk I/O bandwidth, insufficient parallel server processing power for slice part extraction and projection, insufficient network bandwidth for transferring the slice parts from SP nodes to the client PC, insufficient bandwidth of the network interface at the client PC and insufficient processing power at the client PC for receiving many network packets, for assembling slice parts into the final image slice and for displaying the final image slice on the user's window.

To measure the image slice extraction and visualization application performances, the experiment consists of requesting and displaying successive 512x512 RGB image slices, i.e. 768 KBytes, orthogonal to one of the diagonals traversing the Visible Human's rectilinear volume.

Zoom factor 1, extent cache disabled

We first consider the case of a zoom factor of 1, i.e. each extracted image slice is displayed without reduction, where 1 slice extraction request of 120 Bytes is sent per SP node, 437 extents, i.e. 437 x 51KB = 22 MBytes, are read in average from disks and 437 slice parts of 3.8 KBytes each are sent to the client for each extracted 512x512 image slice. Figure 6-15 shows the performances obtained, in number of image slices per second, as a function of the number of contributing SP nodes and a function of number of disks per contributing SP node. In order to test the worst case behaviour, i.e. the general case where successive requests are directed towards completely different extents, no disk caching is allowed.

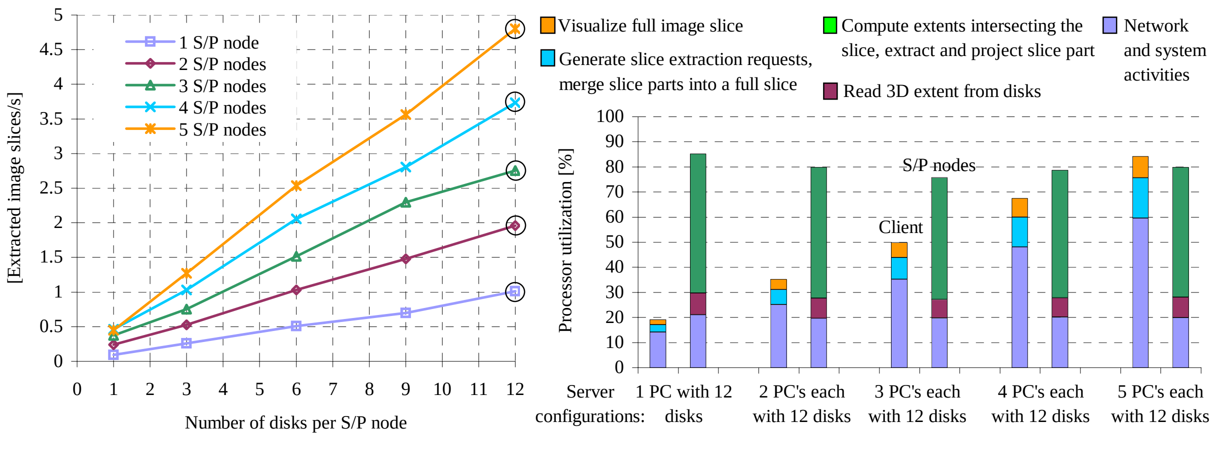


Figure 6-15. Performances at zoom factor 1, with the extent cache disabled on each SP node

When disabling the extent caches, with up to 12 disks per SP node, disk I/O bandwidth is always the bottleneck. Therefore increasing either the number of disks per SP node or the number of SP nodes (assuming each PC incorporates an equal number of disks) increases the number of disks and offers a higher extracted image slice throughput (Figure 6-15). With the 5 SP node 60 disk server configuration an aggregate disk I/O throughput of 4.8[image slices/s] x 437[extents/image slice] x 51[KBytes/extent] = 104 MBytes/s, i.e. 1.74 MBytes/s per disk, is reached. This shows that the disks are the bottleneck since they have a measured I/O mean throughput of 1.88 MBytes/s.

Each time you add a group of 3 disks to a single SP node the aggregate disk I/O throughput increases by ~5.2 MBytes/s, the server processor utilization increases by ~20%, the client processor utilization increases by ~4%, and the extracted image slice throughput increases by 5.2[MBytes/s] / 22[MBytes/slice] = ~0.24 image slices/s.

Each time the extracted image slice throughput increases by 1 image slice/s, the server processor utilization increases by 20[%] / 0.24[slices/s] = 83% and the client processor utilization increases by 4[%] / 0.24[slices/s] = 17%.

The maximum number of disks that a single SP node can handle is 15 disks per SP node and the extracted image slice throughput is (15/3) x 0.24[slices/s] = ~1.2 image slices/s. From 15 disks on, the server processor is the bottleneck. This represents the first scalability limit. On the other hand, the client PC can handle up to 5 SP nodes each with 15 disks and the extracted image slice throughput is (75/3) x 0.24[slices/s] = 6 image slices/s. From 5 SP nodes each with 15 disks the client processor is the bottleneck. This represents the maximum sustainable extracted and visualized image slice throughput assuming that only the disks, the server processors and the client processor are potential bottlenecks.

Zoom factor 1, extent cache enabled

In the present experiment, where we browse through successive image slices having identical orientations, there is a high probability that the next image slice requires data from the same volumic extents as the previous image slice. Therefore, when enabling the SP node's extent caches, the bottleneck shifts quickly from the disks to the limited processing power available on the SP nodes (Figure 6-16). This shows the efficiency of the extent caches when browsing through the Visible Human.

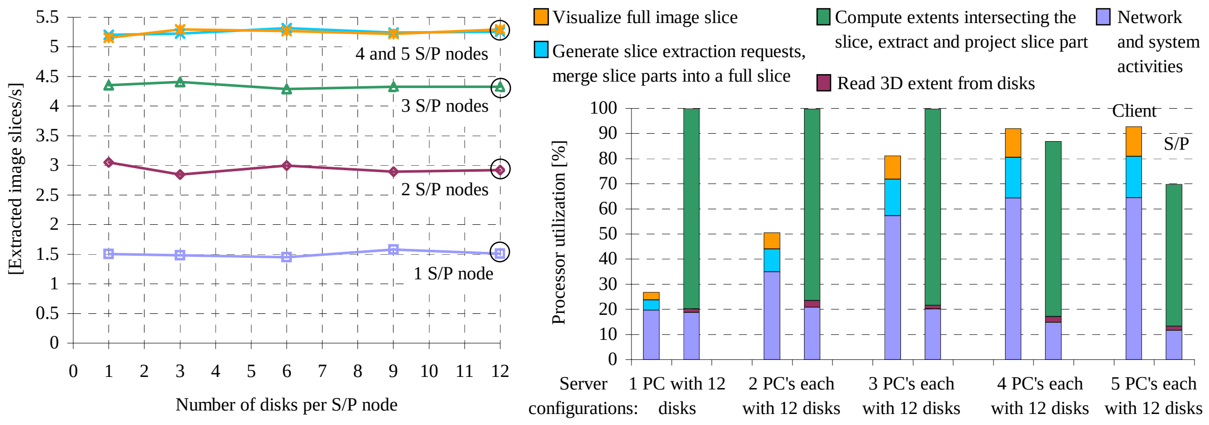


Figure 6-16. Performances at zoom factor 1, with an extent cache of 25 MBytes per SP node

Figure 6-16 shows that with up to 3 SP nodes the slice part extraction and projection operation running on the SP nodes is the bottleneck. Since the extent cache comprises 25 MBytes per SP node and the amount of extents that must be read for extracting a single 512x512 image slice is 22 MBytes, the maximum extent cache efficiency, i.e. the case when each extent is read only once from a disk, is already reached with a single SP node. At this maximum cache hit rate, 97.9% of the accessed extents are read from the extent caches, i.e. 1 of 50 extents is read from disks. This is correct since an extent contains along its vertical axis (Z) 17 x 3 = 51 slice parts.

A single SP node with a single disk and an enabled extent cache is able to extract and project 1.5 image slices/s. Without the extent cache, it sustains with 15 disks a maximum of 1.2 image slices/s. This difference is due to the fact that reading extents from disks require more processing power than reading extents from extent caches.

The next bottleneck is not the client processor but resides in the limited bandwidth of its Fast Ethernet network interface (Figure 6-16, 4 and 5 SP nodes), which saturates at the reception of 5.3[image slices/s] x 437[slice parts/image slice] = 2'316 slice parts/s corresponding to a network throughput of 2'316[slice parts/s] x 3.8[KBytes/slice part] = 8.6 MBytes/s.

Zoom factor 2, extent cache disabled

At a zoom factor of 2, the server processors extract slice parts from their corresponding disk extents, and resample them at a two by two times lower resolution, i.e. each slice part is reduced by a factor of 4 and sent to the client PC. For each 512x512 visualized image slice, a 1024x1024 slice is extracted from the 3D volume where 1'615 extents, i.e. 1'615 x 51KB = 80 MBytes, are read in average from disks and 1'615 slice parts of 1 KBytes each are sent to the client.

As figure 6-17 shows, with a zoom factor of 2, disk I/O throughput is the bottleneck for all considered configurations. Comparing with the zoom factor 1, this assertion is correct, since 4 times more extents need to be read for each extracted image slice.

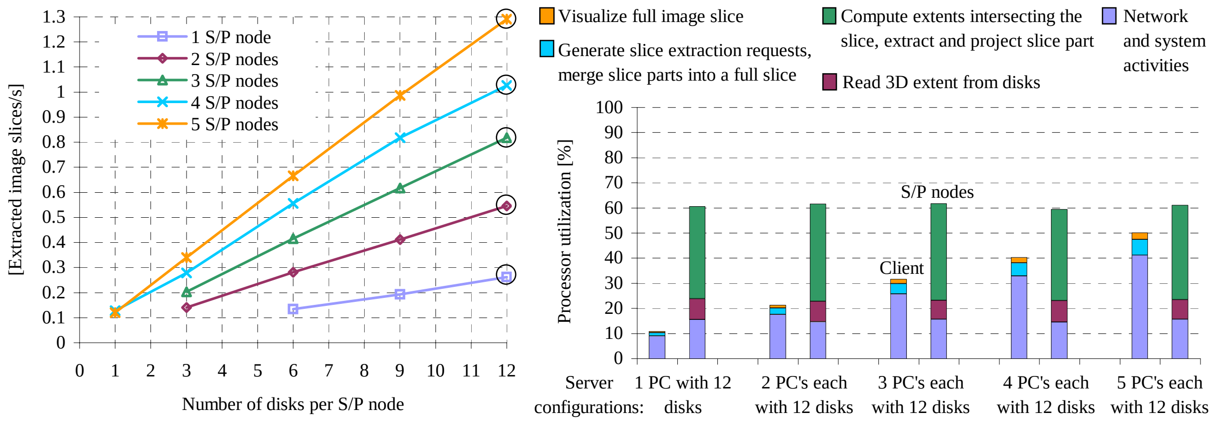


Figure 6-17. Performances at zoom factor 2, with the extent cache disabled on each SP node

As expected, the aggregate disk I/O throughputs are the same as with zoom factor 1 (Figure 6-15). For example, with the 5 SP node 60 disk server configuration, a global disk I/O bandwidth of 1.3[image slices/s] x 1'615[extents/image slice] x 51[KBytes/extent] = 105 MBytes/s is obtained.

Each time you add a group of 3 disks to a single SP node the aggregate disk I/O throughput increases by ~5.2 MBytes/s (same as with zoom 1), the server processor utilization increases by ~16%, the client processor utilization increases by ~2.5%, and the extracted image slice throughput increases by 5.2[MBytes/s] / 80[MBytes/slice] = ~0.065 image slices/s.

Each time the extracted image slice throughput increases by 1 image slice/s, the server processor utilization increases by 16[%] / 0.065[slices/s] = 246% and the client processor utilization increases by 2.5[%] / 0.065[slices/s] = 38%. Comparing these values with the zoom factor 1 experiment (Figure 6-15), the server processor utilization is 246[%] / 83[%] = ~3 times higher since 3.7 times more extents need to be read and a 2x2 reduction needs to be performed on the extracted and projected slice parts. The client processor utilization is 38[%] / 17[%] = ~2 times higher since the received slice parts are smaller (1 KBytes vs. 3.8 KBytes), and their number is larger (1'615 vs. 437). The same amount of data packed into 3.7 times more tokens, requires more than twice as much processing power. Receiving many small TCP/IP packets incurs a higher overhead.

The maximum number of disks that a single SP node can handle is 18 disks, the server processor utilization is 6 x 16[%] = ~96%, and the extracted image slice throughput is 6 x 0.065[image slices/s] = ~0.39 image slices/s. From 18 disks the server processor is the bottleneck. On the other hand, the client processor can handle up to 6 SP nodes each with 18 disks, the client processor utilization is 36 x 2.5[%] = 90%, and the extracted image slice throughput is 36 x 0.065 = ~2.3 image slices/s. From 6 SP nodes, each with 18 disks, the client processor is the bottleneck. This represents the maximum sustainable extracted and visualized image slice throughput assuming that only the disks, the server processors and the client processor are potential bottlenecks.

Zoom factor 2, extent cache enabled

As with zoom factor 1 (Figure 6-16), when enabling the extent caches the bottleneck shifts quickly from the disks to the limited processing power available on the SP nodes (Figure 6-18).

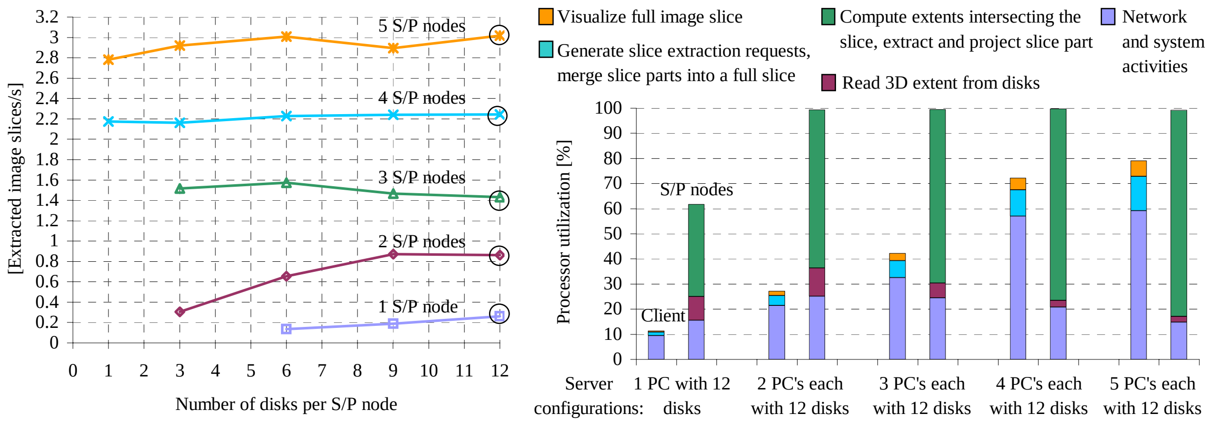


Figure 6-18. Performances at zoom factor 2, with an extent cache of 25 MBytes per SP node

The extent caches become effective only from 2 SP nodes. With a single SP node we obtain exactly the same performances as when the cache was disabled. This is due to cache thrashing.

As with zoom factor 1, i.e. reading extents from disks require more processing power than reading extents from extent caches. A single SP node with an enabled extent cache sustains a maximum throughput of 3[image slices/s] / 5[SP nodes] = 0.6 image slices/s. With a disabled extent cache, it sustains only 0.39 image slices/s.

The efficiency of the extent caches reach their maximum, i.e. the case when each extent is read only once from a disk, when their aggregate sizes is greater than 80 MBytes. This corresponds to the amount of extents that must be read for each 512x512 image slice at zoom factor 2. 80 MBytes aggregate extent cache is available from 4 SP nodes (each with 25 MBytes extent cache). 97.9% of accessed extents are read from the extent caches (as expected, same hit rate as zoom factor 1). With 5 SP nodes a global I/O throughput of up to 3[slices/s] x 80[MBytes/slice] = 240 MBytes/s is obtained.

A surprising but deliberate effect that must be mentioned is that, for the same visualized image slice throughput, the client processor is less loaded when the extent caches are enabled (Figure 6-18, 79[%] / 3[slices/s] = 26% for a rate of 1 slice/s) than disabled (Figure 6-17, 50% / 1.3[slices/s] = 38% for a rate of 1 slice/s). This explains why with the extent caches disabled, the client processor saturates at a rate of only 2.3 image slices/s (see above). This difference is due to the fact that the *SendToken* primitive coalesces several tokens into a same TCP/IP packet so as to reduce the overhead of sending and receiving many small tokens, e.g. 1 KByte slice parts (Section 4.5.3. The efficiency of this technique depends on the output token rate which is higher when extent caches are enabled. Without this clever technique, the client processor would have rapidly become the bottleneck.

6.4 Summary

In this chapter, we have shown that the CAP computer-aided parallelization tool and the PS^2 framework offer very flexible and efficient tools for developing parallel I/O- and compute- intensive applications or high-level libraries. A high-level library provides applications with specific, therefore appropriate and efficient, file abstractions and parallel processing operations on that abstractions, e.g. an image library provides the abstraction of images along with parallel imaging operations, an out-of-core linear algebra library provides the abstraction of out-of-core matrices along with parallel linear operations on matrices (multiplication, LU decomposition, etc).

The compositionality of CAP, i.e. the hierarchical specification of a parallel operation, has been demonstrated in the neighbourhood dependent imaging operation (Section 6.2.3). A first parallel operation processing a single tile has been independently devised. And then, thank to the compositionality of CAP, this parallel *ProcessTile* operation has been incorporated into the schedule of another parallel operation processing a whole tiled 2D image, i.e. performing the parallel *ProcessTile* operation on each tile and writing the processed tiles to disks. This approach of developing hierarchical operations is similar to the approach of procedural languages where a routine is treated as a black box performing a task that can be incorporated into another routine performing a more complex task.

Section 6.2 has described how to develop parallel imaging operations and how to incorporate them in a high-level image library. We have demonstrated that with our new tools (CAP and PS^2), we are able to rapidly and efficiently develop neighbourhood independent parallel imaging operations as well as neighbourhood dependent parallel imaging operations.

In Section 6.3, the applicability and the performances of the CAP tool and the PS^2 framework on real applications has been demonstrated with a parallel image slice extraction server application. This application enables clients to specify, access in parallel, and visualize image slices having any desired position and orientation. The image slices are extracted from the 14 GByte Visible Human Male striped over the available set of disks. We have shown that on a 5 Bi-Pentium Pro PC PS^2 server comprising 60 disks, the system is able to extract in parallel, resample and visualize 4.8 512x512 colour image slices per second. At the highest load and when extent caches are disabled, an aggregate I/O disk bandwidth of 104 MBytes/s has been obtained. When extent caches are enabled, one obtains an I/O throughput of up to 240 MBytes/s. The obtained image slice access times shows that performances are close to the best performance obtainable by the underlying hardware.

The development of this impressive Visible Human Slice Web server has required the expertise of several people and students among them Dr. Benoit Gennart for his CAP tool and for being the head of the GigaServer research group at the LSP laboratory, Vincent Messerli for having parallelized the algorithm and measuring the performances, Oscar Figueiredo for his sequential image slice extraction algorithm, Samuel Vetsch for having interfaced the parallel image slice extraction engine to the Microsoft IIS Web server and for his Java-based browser, Laurent Bovisi for having developed an earlier parallel version of the application, WDS Technologies SA and Luc Bidaut at the Hôpital Cantonal Universitaire de Genève for their various contributions to the project.

Chapter 7

Conclusion

This thesis proposes a new approach for developing parallel I/O- and compute- intensive applications on distributed memory computers based on commodity components (e.g. PC's, Fast Ethernet, SCSI-2 disks, Windows NT). As mentioned in the introduction, there are many applications from different fields (e.g. scientific applications, multimedia applications, information server applications) that need huge amounts of computing power and high-performance storage systems. To fulfil these parallel I/O- and compute- intensive applications, not only do we need parallel machines[1] incorporating tens to hundreds of high-speed processors with hundreds of independent disks connected by fast networks, but also we need appropriate systems[2], libraries, and tools for developing such consuming parallel applications.

Indeed, these new generation of parallel computers possess enormous raw computing powers and huge aggregate disk I/O throughputs meeting the needs of most parallel I/O- and compute- intensive applications. Without adequate parallel programming tools, parallel applications do generally not provide the expected speedup, i.e. they do not utilize efficiently the underlying parallel architecture. Computing power and disk I/O bandwidth may be wasted. These wasted resources can be due to the application itself, for example, non-optimized code, superfluous data communications, superfluous disk accesses, synchronous communications, synchronous disk accesses, poor load-balancing, a bad granularity of parallelism, waitings, or inefficient parallelization. Moreover, depending on how an application interacts with a communication system or a parallel storage system, performances may vary (or rather suffer), e.g. accessing many tiny chunks of data distributed in a declustered file is likely to be inefficient, sending large messages may congest the communication system, etc.

This thesis proposes a computer-aided parallelization tool (CAP) based on a macro-dataflow computational model along with a parallel storage-and-processing framework (PS^2) for rapidly developing parallel I/O- and compute- intensive applications that are efficient, i.e. applications that optimize the resources of the underlying parallel hardware (disks, processors, network). The PS^2 framework consists of an extensible parallel storage-and-processing server along with a library of reusable low-level parallel file system CAP components. Application or library programmers can, thanks to the CAP formalism, extend the functionalities of the PS^2 framework, i.e. combine application-specific or library-specific processing operations with the predefined low-level parallel file access operations, to yield highly pipelined parallel I/O- and compute- intensive programs.

The strength and the originality of the proposed tools (CAP and PS^2) are the following:

- CAP alleviates the task of writing parallel programs on distributed memory architectures. Thanks to its macro-dataflow computational model, the parallel behaviour of a program is expressed at a high-level of abstraction, i.e. as a flow of data and parameters between operations running on the same or on different processors.
- CAP offers a clean separation between parallel program parts and sequential program parts. Therefore, CAP programs are easier to develop, debug, and maintain.
- CAP is a flexible tool enabling to easily modify the schedule of suboperations. This flexibility is needed when enhancing an application or when extending a pre-existing application.
- Thanks to a configuration file, a CAP program may run on a variety of hardware configurations without recompilation. This feature enables programmers to debug a CAP program first as a multi-thread single-process single-PC application, second as a multi-thread multi-process single-PC application, and finally as a multi-thread multi-process multi-PC application.
- CAP and PS^2 are first and foremost optimized for developing parallel I/O- and compute- intensive programs on distributed memory PC's (PC's, Windows NT, Fast Ethernet, SCSI-2 disks). Parallel architectures based on commodity components offer affordable leading-edge parallel hardware for small- and medium- sized companies.

1. We call them parallel I/O-and-compute systems.
2. Operating systems, parallel storage systems, communication systems.

- PS^2 is based on top of a native file system, e.g. the NT file system. The use of the NT file system does not affect performances. Therefore, PS^2 is fully compatible with other Windows NT applications, e.g. one can copy a PS^2 parallel file using the 'Explorer' application.
- With PS^2 processing operations are performed where data reside. There is no separation between I/O nodes and compute nodes. PS^2 uses the notion of storage and processing node (SP node) where computations and disk I/O accesses take place, thus avoiding superfluous data communications.
- PS^2 provides a customizable framework for developing high-level libraries. PS^2 is not a parallel storage system intending to directly meet the specific needs of every user by providing a general-purpose abstraction of declustered files. Rather, PS^2 has been designed as a framework for developing high-level libraries, each of which was designed to meet the needs of a specific community of users by providing appropriate file abstractions and I/O-and-compute interfaces.
- PS^2-based programs overlap disk accesses and network communications with computations thus optimizing the resources of the underlying parallel hardware (processors, disks, network).

By implementing a parallel image slice extraction server application, we have demonstrated the applicability of CAP and PS^2 on real complex applications requiring the interaction of different software components (graphical interfaces, Web server interfaces, parallel I/O- and compute- intensive engines, etc) developed with various programming environment (Microsoft Visual C/C++, Borland Delphi, Borland Java Builder, etc) in diverse programming languages (C/C++, Object Pascal, Java, etc).

A number of other applications have been successfully developed using CAP [Gennart96, Mazzariol97]. We expect CAP and PS^2 to offer a high potential for developing powerful parallel I/O- and compute- intensive applications, e.g. parallel Web servers and parallel I/O for distributed memory supercomputers running scientific applications.

7.1 Limitations and future improvements

This section discusses the current limitations of PS^2 as well as possible improvements and extensions that can be incorporated to CAP and PS^2.

(a) **Filling factors in flow-controlled split-merge CAP constructs.**
Performances of a CAP program highly depends on how well the flow-control mechanisms work, i.e. the number of tokens circulating within a particular split-merge CAP construct (filling factors). Too small filling factors reduce the parallelism. Too large filling factors raise a memory overflow condition. Based on my experience, it is difficult to compute these filling factors giving the best performances and consuming the least memory. Therefore, a solution should be found where CAP automatically adjusts, at execution time, the filling factor of a flow-controlled split-merge CAP construct by monitoring the execution time and the utilization of each stage within the split routine and the merge routine. Based on these informations, CAP may be able to compute automatically the optimal values of the filling factors.

(b) **Metafiles.**
At the present time, PS^2 does not rely on a metafile to locate the extent files making up a parallel file. Indeed, each time a parallel file is opened, all the virtual disks are interrogated in order to locate the extent files. This solution has two main drawbacks: it incorporates a potential scalability limit when increasing the number of virtual disks and it imposes that each virtual disk has the same extent file directory tree. To overcome these limitations and to increase the flexibility of PS^2, the solution would be to have a metafile per parallel file giving the (NTFS) path names of the extent files making up the parallel file. Therefore, when opening a parallel file, its metafile is read and PS^2 can dynamically assign the extent files making up the parallel file to the extent server threads, i.e. to decide which extent server thread accesses which extent file. Moreover, this solution ensures that PS^2 is able to access the extent files whatever the PS^2 server configuration is, i.e. the number of extent server threads, their locations on the cluster of PC's, the number of computer server threads, their locations on the cluster of PC's. Also this solution breaks away with the limitations imposed by the static allocation of extent server threads (Program 5-3) and compute server threads (Program 5-4) based on a virtual disk server index.

(c) **Simultaneously running multiple applications.**

Since PS^2 is not a resident parallel file system providing applications with parallel storage services (clients/server model), but an extensible parallel storage-and-processing framework for developing applications, each PS^2 application incorporates its own PS^2 server process hierarchy. Therefore, when executing several concurrent PS^2 applications on the same cluster of PC's, each PS^2 application runs its own interface server thread and its own set of compute server threads, extent file server threads, and extent server threads wasting computing resources and precluding any optimization of disk arm motions, extent caches, and extent address caches. The present version of PS^2 supports a simple parallel application that executes on a dedicated cluster of PC's. Such applications include, for example, parallel Web servers and scientific applications requiring all the computing resources of the parallel machine (memory, processors, network, disks). To offer support for multiple concurrently running applications, CAP should incorporate new mechanisms enabling a client CAP application to be connected to a pre-existing PS^2 server process hierarchy. This raises new open issues, such as how to dynamically extend the functionalities of the pre-existing compute server threads, and how to protect the PS^2 server from being corrupted.

Bibliography

Ackerman95 Michael J. Ackerman, *Accessing the Visible Human Project*, D-Lib Magazine: The Magazine of the Digital Library Forum, October 1995, http://www.dlib.org/dlib/october95/10ackerman.html.

Agerwala82 Tilak Agerwala, Arvind, “Data Flow Systems: Guest Editors’s Introduction,” *IEEE Computer*, vol. 15, no. 2, pp. 10-13, February 1982.

Almasi94 George S. Almasi, Allan Gottlieb, *Highly Parallel Computing*, Second Edition, The Benjamin/Cummings Publishing Company, 1994.

Amdahl67 G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” *Proceedings AFIPS 1967 Spring Joint Computer Conference*, pp. 483-485, Atlantic City, New Jersey, April 1967.

Bach86 Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall International Editions, 1986.

Beguelin90 Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Vaidy S. Sunderam, “A User’s Guide to PVM Parallel Virtual Machine,” *Oak Ridge National Laboratory Report ORNL/TM-11826*, July 1990.

Beguelin92 Adam Beguelin, Jack Dongarra, Al Geist, Robert Mancheck, Vaidy S. Sunderam, “HeNCE: Graphical Development Tools for Network-Based Concurrent Computing,” *Proceedings of SHPCC-92*, IEEE Computer Society Press, pp. 129-136, Los Alamitos, California, 1992.

Best93 Michael L. Best, Adam Greenberg, Craig Stanfill, Lewis W. Tucker, “CMMD I/O: A Parallel Unix I/O,” *Proceedings of the Seventh International Parallel Processing Symposium*, pp. 489-495, 1993.

Browne84a J. C. Browne, “Parallel Architectures for Computer Systems,” *Physics Today*, vol. 37, no. 5, pp. 28-35, May 1984.

Browne84b J. C. Browne, “Parallel Architectures for Computer Systems,” *Computer*, vol. 17, no. ??, pp. 83-87, July 1984.

Carretero96a J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso, “ParFiSys: A Parallel File System for MPP,” *ACM SIGOPS*, vol. 30, no. 2, pp. 74-80, April 1996.

Carretero96b J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso, “I/O Data Mapping in ParFiSys: Support for High-Performance I/O in Parallel and Distributed Systems,” *Proceedings of the 1996 EuroPar Conference*, Lecture Notes in Computer Science 1123, Springer-Verlag, pp. 522-526, August 1996.

Carriero89a Nicholas Carriero, David Gelernter, “Linda in Context,” *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, April 1989.

Carriero89b Nicholas Carriero, David Gelernter, “How to Write Parallel Programs: A Guide to the Perplexed,” *Journal of the ACM*, vol. 21, no. 3, pp. 323-357, September 1989.

Chen95 Huang-Jen Chen, *A Disk Scheduling Scheme and MPEG Data Layout Policy for Interactive Video Access from a Single Disk Storage Device*, Ph.D. thesis, Boston University, College of Engineering, 1995.

Cohen98 Aaron Cohen, Mike Woodring, *Win32 Multithreaded Programming*, O’Reilly & Associates, January 1998.

Corbett96	Peter F. Corbett, Dror G. Feitelson, “The Vesta Parallel File System,” *ACM Transactions on Computer Systems*, vol. 14, no. 3, pp. 225-264, August 1996.
Cormen93	Thomas H. Cormen, David Kotz, “Integrating theory and practice in parallel file systems,” *Proceedings of the 1993 DAGS/PC Symposium on Parallel I/O and Databases*, pp. 64-74, Hanover, NH, June 1993.
Dennis75	J. Dennis, “First Version of a Data Flow Procedure Language,” *MIT Technical Report TR-673*, MIT, Cambridge, Massachusetts, USA, May 1975.
Dibble88	Peter C. Dibble, Michael L. Scott, Carla Schlatter Ellis, “Bridge: A High-Performance File System for Parallel Processors,” *Proceedings of the 8th International Conference on Distributed Computer Systems*, IEEE Computer Society Press, pp. 154-161, San Jose, California, June 1988.
Dibble89	Peter C. Dibble, Michael L. Scott, “Beyond Striping: The Bridge Multiprocessor File System,” *Computer Architecture News*, vol. 17, no. 5, pp. 32-39, September 1989.
Feitelson95	Dror G. Feitelson, Peter F. Corbett, Sandra Johnson Baylor, Yarsun Hsu, “Parallel I/O Subsystems in Massively Parallel Supercomputers,” *IEEE Parallel & Distributed Technology*, pp. 33-47, Fall 1995.
Figueiredo99	Oscar Figueiredo, *Advances in Discrete Geometry Applied to the Extraction of Planes and Surfaces from 3D Volumes*, Ph.D. thesis 1944, Département d’Informatique, École Polytechnique Fédérale de Lausanne, February 1999.
Gennart94	Benoit A. Gennart, Bernard Krummenacher, Laurent Landron, Roger D. Hersch, “GigaView Parallel Image Server Performance Analysis,” *Proceedings of the World Transputer Congress, Transputer Applications and Systems*, IOS Press, pp. 120-135, Como, Italy, September 1994.
Gennart96	Benoit A. Gennart, Joaquin Tarraga, Roger D. Hersch, “Computer-Assisted Generation of PVM/C++ Programs using CAP,” *Proceedings of EuroPVM’96*, Lecture Notes in Computer Science 1156, Springer-Verlag, pp. 259-269, Munich, Germany, Octobre 1996.
Gennart98a	Benoit A. Gennart, Marc Mazzariol, Vincent Messerli, Roger D. Hersch, “Synthesizing Parallel Imaging Applications using CAP Computer-Aided Parallelization Tool,” *IS&T/SPIE’s 10th Annual Symposium, Electronic Imaging’98, Storage & Retrieval for Image and Video Database VI*, pp. 446-458, San Jose, California, USA, January 1998.
Gennart98b	Benoit A. Gennart, *The CAP Computer Aided Parallelization Tool: Language Reference Manual*, July 1998.
Ghezzi82	Carlo Ghezzi, Mehdi Jazayeri, *Programming Language Concepts*, John Wiley, 1982.
Ghezzi85	Carlo Ghezzi, “Concurrency in programming languages: A survey,” *Parallel Computing*, vol. 2, pp. 229-241, November 1985.
Gibson95	Garth A. Gibson, Daniel Stodolsky, Fay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, Jim Zelenka, “The Scotch Parallel Storage Systems,” *Proceedings of the 40th IEEE Computer Society International Conference (COMPCON 95)*, pp. 403-410, San Francisco, March 1995.
Grimshaw93a	Andrew S. Grimshaw, “The Mentat Computation Model – Data-Driven Support for Dynamic Object-Oriented Parallel Processing,” *Computer Science Technical Report CS-93-30*, University of Virginia, Charlottesville, Va., May 1993.
Grimshaw93b	Andrew S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, vol. 26, no. 5, pp. 39-51, May 1993.

Grimshaw96	Andrew S. Grimshaw, Jon B. Weissman, W. Timothy Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," *ACM Transactions on Computer Systems*, vol. 14, no. 2, pp. 139-170, May 1996.
Halstead85	Robert H. Halstead Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501-538, October 1985.
Hansen75	Per Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 199-207, June 1975.
Hatcher91	P. J. Hatcher, M. J. Quinn, A. J. Lapadula, B. K. Seevers, R. J. Anderson, R. R. Jones, "Data-Parallel Programming on MIMD Computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 377-383, 1991.
Hennessy96	John L. Hennessy, David A. Patterson, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.
Hersch93	Roger D. Hersch, "Parallel Storage and Retrieval of Pixmap Images," *Proceedings of the 12th IEEE Symposium on Mass Storage Systems*, IEEE Computer Society Press, pp. 221-226, Monterey, California, April 1993.
Huber95a	James V. Huber, Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, David S. Blumenthal, "PPFS: A High Performance Portable Parallel File System," *Proceedings of the 9th ACM International Conference on Supercomputing*, pp. 385-394, Barcelona, Spain, July 1995.
Huber95b	James Valentine Huber, Jr. *PPFS: An Experimental File System for High Performance Parallel Input/Output*, Ph.D. thesis, Graduate College of the University of Illinois, Urbana-Champaign, Illinois, February 1995.
Inmos85	Inmos Limited, *Occam Programming Manual*, 1985.
Jones85	Geraint Jones, "Programming in Occam, A Tourist Guide to Parallel Programming," *Programming Research Group, Technical Monograph PRG-13*, Oxford University Computing Laboratory, 1985.
Karpovich93	John F. Karpovich, Andrew S. Grimshaw, James C. French, "Breaking the I/O Bottleneck at the National Radio Astronomy Observatory (NRAO)," *Technical Report CS-94-37*, University of Virginia, Department of Computer Science, CharlottesVille, Virginia, September 1993.
Kotz91	David Kotz, Carla Schlatter Ellis, "Practical Prefetching Techniques for Parallel File Systems," *Proceedings of the Conference on Parallel and Distributed Information Systems*, IEEE Computer Society Press, pp. 182-189, December 1991.
Kotz94a	David Kotz, Nils Nieuwejaar, "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload," *Computer Science Technical Report PCS-TR94-211*, Department of Computer Science, Dartmouth College, Hanover, August 1994.
Kotz94b	David Kotz, "Disk-directed I/O for MIMD Multiprocessors," *ACM Transactions on Computer Systems*, vol. 15, no. 1, pp. 41-74, February 1997.
Kotz95a	David Kotz, "Exploring the use of I/O Nodes for Computation in a MIMD Multiprocessor," *Proceedings of the Workshop for I/O in Parallel and Distributed Systems at IPPS'95*, pp. 78-89, 1995.
Kotz95b	David Kotz, "Expanding the Potential for Disk-Directed I/O," *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, pp. 490-495, IEEE Computer Society Press, San Antonio, Texas, October 1995.

Kotz96	David Kotz and Nils Nieuwejaar, "Flexibility and Performance of Parallel File Systems," *ACM Operating Systems Review*, vol. 30, no. 2, pp. 63-73, 1996.
Krieger97	Orran Krieger, Michael Stumm, "HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 286-321, August 1997.
Loveman93	David B. Loveman, "High Performance Fortan," *IEEE Parallel & Distributed Technology*, vol. 1, no. 1, pp. 25-42, February 1993.
LoVerso93	Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, Richard Wheeler, "*sfs*: A Parallel File System for the CM-5," *Proceedings of the 1993 Summer USENIX Technical Conference*, pp. 291-305, Cincinnati, Ohio, June 1993.
Miller88	Phillip C. Miller, Charles E. St. John, Stuart W. Hawkinson, "FPS T Series Parallel Processor," in Robert G. Babb II, editor, *Programming Parallel Processors*, Addison-Wesley, 1988.
Mazzariol97	Marc Mazzariol, Benoit A. Gennart, Vincent Messerli, Roger D. Hersch, "Performance of CAP-Specified Linear Algebra Algorithms," *Proceedings of EuroPVM-MPI'97*, LNCS 1332, Springer Verlag, pp. 351-358, Krakow, Poland, November 1997.
Messerli97	Vincent Messerli, Benoit A. Gennart, Roger D. Hersch, "Performances of the PS^2 Parallel Storage and Processing System for Tomographic Image Visualization," *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, IEEE Computer Society Press, pp. 514-522, Seoul, Korea, December 1997.
Microsoft96	Microsoft Corporation, *Windows Sockets 2 Application Program Interface*, 1996.
Mockapetris87a	P. V. Mockapetris, *Domain Names: Concepts and Facilities*, Request For Comment 1034, November 1987.
Mockapetris87b	P. V. Mockapetris, *Domain Names: Implementation and Specification*, Request For Comment 1035, November 1987.
Mogul93	J. C. Mogul, "IP Network Performance," *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, Addison-Wesley, Reading, Massachusetts, pp. 575-675, 1993.
MPI94	Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *The International Journal of Supercomputer Applications and High Performance Compting*, vol. 8, 1994.
MPI97	Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," *Technical Report*, July 1997, http://www.mpi-forum.org.
Nagle84	J. Nagle, *Congestion Control in IP/TCP Internetworks*, Request For Comment 896, January 1984.
Nieuwejaar94	Nils A. Nieuwejaar, David Kotz, "A Multiprocessor Extension to the Conventional File System Interface," *Technical Report PCS-TR94-230*, Department of Computer Science, Dartmouth College, Hanover, September 1994.
Nieuwejaar96	Nils A. Nieuwejaar, *Galley: A New Parallel File System For Scientific Workloads*, Ph.D. thesis, Department of Computer Science, Dartmouth College, Hanover, November 1996.
Nitzberg92	Bill Nitzberg, "Performance of the iPSC/860 Concurrent File System," *Technical Report RND-92-020*, NAS Systems Divisions, NASA Ames Research Center, Moffett Field, California, December 1992.

North96 C. North, B. Shneiderman, C. Plaisant, “User-Controlled Overviews of an Image Library: A Case Study of the Visible Human,” *Proceedings of the 1996 ACM Digital Libraries Conference*, ACM Press, 1996.

Ousterhout85 John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson, "A Trace Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pp. 15-24, December 1985.

Papadopoulos93 C. Papadopoulos, G. M. Parulkar, “Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 429-440, August 1993.

Parasoft90 ParaSoft Corporation, *Express C User’s Guide (Version 3.0)*, 1990.

Partridge93 C. Partridge, S. Pink, “A Faster UDP,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 429-440, August 1993.

Patterson88 David A. Patterson, Garth Gibson, Randy H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID),” *Proceedings of the International Conference on Management of Data (SIGMOD)*, pp. 109-116, June 1988.

Patterson98 David A. Patterson, John L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

Patwardhan94 Tai Patwardhan, “ASA-II Extending the Power of Seagate’s Advanced SCSI Architecture,” *Seagate Advanced SCSI Architecture II Technology Paper*, October 1994, http://elm.eimb.rssi.ru/jwz/seagate/tech/asa2.shtml.

Pierce89 Paul Pierce, “A Concurrent File System for a Highly Parallel Mass Storage System,” *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, vol. 1, pp. 155-160, 1989.

Pool94 James Pool, “Preliminary survey of I/O intensive applications,” *Technical report CCSF-38*, Caltech Concurrent Supercomputing Facilities, January 1994.

Postel80 J. B. Postel, *User Datagram Protocol*, Request For Comment 768, August 1980.

Postel81a J. B. Postel, *Internet Protocol*, Request For Comment 791, September 1981.

Postel81b J. B. Postel, *Transmission Control Protocol*, Request For Comment 793, September 1981.

Postel94 J. B. Postel, *Internet Official Protocol Standards*, Request For Comment 1600, March 1994.

Reed95 Dan Reed, Charles Catlett, Alok Choudhary, David Kotz, Marc Snir, “Parallel I/O: Getting Ready for Prime Time,” *IEEE Parallel & Distributed Technology*, pp. 64-71, Summer 1995.

Rosario93 Juan Miguel del Rosario, Rajesh Bordawekar, Alok Choudhary, “Improved Parallel I/O via a Two-phase Run-time Access Strategy,” *Proceedings of the IPPS ’93 Workshop on Input/Output in Parallel Computer Systems*, pp. 56-70, 1993.

Seamons95 K. E. Seamons, M. Winslett, “A Data Management Approach for Handling Large Compressed Arrays in High Performance Computing,” *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computations*, pp. 119-128, February 1995.

Shu91 W. Shu, L. V. Kale, “Chare Kernel - A Runtime Support System for Parallel Computations,” *Journal of Parallel Distributed Computing*, vol. 11, pp. 198-211, 1991.

Silberschatz95	Abraham Silberschatz, Peter B. Galvin, *Operating System Concepts*, Fourth Edition, Addison-Wesley Publishing Company, January 1995.
Spitzer96	Victor Spitzer, Michael J. Ackerman, Ann L. Scherzinger, David Whitlock, "The Visible Human Male: A Technical Report," *Journal of the American Medical Informatics Association*, vol. 3, no. 2, pp. 118-130, March/April 1996.
Srini86	V. P. Srini, "An Architectural Comparison of Dataflow Systems," *IEEE Computer*, vol. 19, no. 3, pp. 68-88, March 1986.
Stevens90	W. Richard Stevens, *UNIX Network Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
Stevens96	W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley Professional Computing Series, Reading, Massachusetts, December 1996.
Sunderam90	Vaidy S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice & Experience*, vol. 2, no. 4, pp. 315-339, December 1990.
Sutton94	Tim Sutton, "Barracuda 2 2HP: Parallel Processing for Storage Devices," *Seagate Barracuda 2 Technology Paper,* October 1994, http://elm.eimb.rssi.ru/jwz/seagate/tech/cuda2.shtml.
Veen86	A. H. Veen, "Dataflow Machine Architecture," *ACM Computing Survey*, vol. 18, no. 4, pp. 365-396, December 1986.
Vetsch98	Samuel Vetsch, Vincent Messerli, Oscar Figueiredo, Benoit A. Gennart, Roger D. Hersch, Laurent Bovisi, Ronald Welz, Luc Bidaut, "A Parallel PC-based Visible Human Slice Web Server," *The Visible Human Project Conference Proceeding*, Bethesda, Maryland, USA, October 1998.
Watt90	David A. Watt, *Programming Language Concepts and Paradigms*, Prentice-Hall, 1990.
Weider92	C. Weider, J. K. Reynolds, S. Heker, *Technical Overview of Directory Services Using the X.500 Protocol*, Request For Comment 1309, March 1992.
Wilson96	Gregory V. Wilson, Paul Lu (Eds), *Parallel Programming Using C++*, The MIT Press, 1996.
Wright95	Gary R. Wright, W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley Professional Computing Series, Reading, Massachusetts, January 1995.

Biography

Vincent Messerli was born in the international city of Geneva, Switzerland, on October the 30th 1969. At the age of 14, he began the “École d’Ingénieur de Genève” where he obtained in June 1989 its diploma of technical engineer. After a 4-month military service, he left his natal city and went to “Le Brassus” a little village in “le Jura” where he worked for a small company for 1 year as a research and development engineer. He was assigned to a team of 10 engineers developing security equipments and modems. In October 1990 he began the “École Polytechnique Fédérale de Lausanne” where he obtained in April 1994 his diploma of computer science engineer. Then, he left Switzerland and went to Torquay in England for 7 months. He followed English courses. On December the 1st 1995, Vincent began his PhD at the Peripheral System Laboratory, Computer Science department, “École Polytechnique Fédérale de Lausanne”. He pursued 4 years research on tools for developing parallel I/O- and compute- intensive applications. His research interests include parallel file systems, parallel and distributed computing, computer-aided parallelization tool, network programming, and design pattern in C++.

Publications:

- Benoit Gennart, Vincent Messerli, Roger D. Hersch, “Performances of Multiprocessor Multidisk Architectures for Continuous Media Storage,” *Storage and Retrieval for Still Image and Video Databases IV*, SPIE Proc. 2670, Sethi, Jain Editors, pp. 286-299, San Jose, California, February 1996.
- Vincent Messerli, Benoit A. Gennart, Roger D. Hersch, “Performances of the PS^2 Parallel Storage and Processing System for Tomographic Image Visualization,” *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, IEEE Computer Society Press, pp. 514-522, Seoul, Korea, December 1997.
- Marc Mazzariol, Benoit A. Gennart, Vincent Messerli, Roger D. Hersch, “Performance of CAP-Specified Linear Algebra Algorithms,” *Proceedings of EuroPVM-MPI’97*, LNCS 1332, Springer Verlag, pp. 351-358, Krakow, Poland, November 1997.
- Benoit A. Gennart, Marc Mazzariol, Vincent Messerli, Roger D. Hersch, “Synthesizing Parallel Imaging Applications using CAP Computer-Aided Parallelization Tool,” *IS&T/SPIE’s 10th Annual Symposium, Electronic Imaging’98, Storage & Retrieval for Image and Video Database VI*, pp. 446-458, San Jose, California, USA, January 1998.
- Samuel Vetsch, Vincent Messerli, Oscar Figueiredo, Benoit A. Gennart, Roger D. Hersch, Laurent Bovisi, Ronald Welz, Luc Bidaut, “A Parallel PC-based Visible Human Slice Web Server,” *The Visible Human Project Conference Proceeding*, Bethesda, Maryland, USA, October 1998.