CAP : a methodology for the hierarchical specification of pipelined parallel applications

Benoit A. Gennart and Roger D. Hersch
Peripheral Systems Laboratory, EPFL
{gennart,hersch}@di.epfl.ch

Abstract. Programming parallel shared- and distributed-memory architectures remains a difficult task. This contribution proposes a methodology for the hierarchical specification of pipelined parallel applications running on shared- as well as distributed-memory architecture. The methodology targets coarse to medium grain parallelism. The CAP methodology (Computer-Aided Parallelization) assumes that parallel hardware works as a factory producing cars. The important part of the analogy is the support for pipelining. Another important feature of the CAP methodology is its hierarchical and compositional nature. The methodology is supported by the CAP language extension to C++. The CAP extension translates to sequential C++ programs for application validation using conventional debuggers, to shared-memory parallel programs based on threads, and to distributed-memory parallel programs communicating using the PVM message-passing library. This contribution presents the CAP methodology, the CAP language extension, as well as an application of the CAP methodology to medical imaging. It also presents the current status of the CAP project.

1 Introduction

Designing and specifying parallel applications remains a hazardous task. Even simple parallel applications, requiring little or no communication between processing elements (dubbed *embarrassingly parallel* benchmarks) require considerable effort from the programmer. There is a clear need for a parallel application design environment. The requirements for a parallel application design environment are :

- portability and support for shared-memory and distributed-memory architectures.
- integration in an existing language environment, to be able to use existing language tools (compiler, debugger).
- automated parallel program generation and downloading of sequential functions to the various processing elements of the architecture.

The Computer-Aided Parallelization (CAP) framework in this contribution is based on decomposing high-level operations such as 2-D and 3-D image reconstruction, database queries, or mathematical computations into a set of sequential suboperations with clearly defined input and output data. The application programmer uses the CAP language to specify the flowchart of sequential suboperations required to complete a given high-level operation, and assigns each suboperation to a processing element. The CAP language is a C++ extension. The CAP preprocessor translates the CAP specification into a set of concurrent programs communicating through communication libraries such as MPI, PVM, and TCP/IP, or communicating through shared memory. The concurrent programs are run on the various intelligent processing elements of the parallel architecture. The CAP methodology targets coarse to medium grain parallelism.

Section 2 describes the multiprocessor multidisk architecture toward which the CAP methodology is targeted. Section 3 describes the CAP language extension and parallel-program development methodology, using a car factory example. Section 4 describes the design features of the CAP language. Section 5 compares the CAP methodology to other parallel-program design methodologies. Section 6 shows the CAP specification of a parallel medical-imaging application. Section 5 describes the current status of the CAP project.

2 Multiprocessor multidisk architectures

The CAP methodology targets coarse to medium grain parallelism, on parallel and distributed architectures, and in particular *multiprocessor multidisk* architectures. Multiprocessor multidisk architectures consists of several *intelligent disk-nodes* linked by a high-speed interconnection. Each intelligent disk-node consists of a processor and one or more secondary storage devices, such as magnetic disks or CD-Roms. Examples of multiprocessor multidisk architectures are (1) the GigaView architecture [2,3], consisting of transputers connected by serial links or a crossbar switch (Figure 1) ; shared-bus shared-memory multiprocessor UNIX workstations with multiple storage devices (Figure 2) ; single-processor single-disk workstations connected by a network (Ethernet, FDDI, ATM).

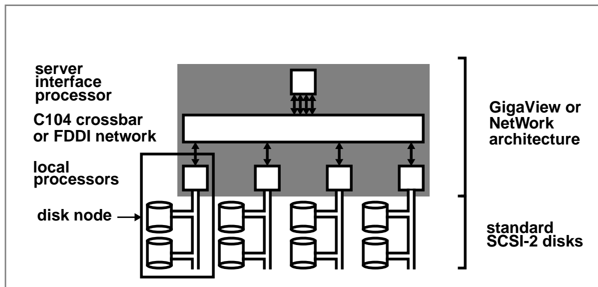

FIGURE 1. Distributed-memory architecture

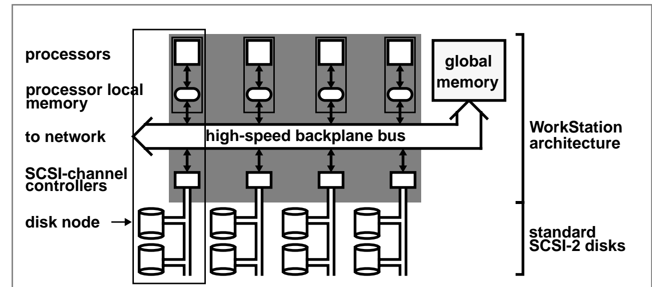

FIGURE 2. Shared-memory architecture

The computation model for a multiprocessor multidisk architecture assumes (1) that data is distributed in *tiles* on the various nodes of the architecture, and (2) that an operation on distributed data can be divided in suboperations to be executed on each data tiles by the intelligent storage node processors, followed by a merge operation gathering the results of the suboperations into a single result.

3 The factory analogy

Consider a simplified factory which produces cars out of various parts : body, engine, frame, interior. The factory is divided in several *stations* producing each of the parts, and all working in parallel. Once all parts are produced, they are put together to complete the car. While a body, engine, frame and interior are produced, the body, engine, frame, and interior of the previous production run are put together to produce a car.

```
process FactoryT {
processes :
  BodyStationT          BodyStation ;
  EngineStationT        EngineStation ;
  FrameStationT         FrameStation ;
  InteriorStationT      InteriorStation ;
  IntegrationStationT IntegrationStation ;
operations :
  ProduceACar   <-: RawMaterialsT Input
                >-: CarT Output ;
} ;
```

PROGRAM 1. FactoryT specification

```
operation FactoryT::ProduceACar
    <-: RawMaterialsT Input
    >-: CarT Output
{
    parallel
      ( BodyStation.ProduceBodyParts
      , EngineStation.ProduceEngine
      , FrameStation.ProduceFrame
      , InteriorStation.ProduceInterior
      ) >->
    IntegrationStation.IntegrateCarParts ;
}
```

PROGRAM 2. ProduceACar operation specification

The CAP specification of the `FactoryT::ProduceACar` operation (Programs 1 and 2) consists of the FactoryT process specification and the ProduceACar operation specification.

The **`process`** construct specifies the stations in the factory : body, engine, frame, interior and integration section. The **`process`** construct also specifies the name, input and output of the operations the `FactoryT` can perform. The factory can perform one operation, namely `ProduceACar`. To produce a car, the

FactoryT needs raw materials (RawMaterialsT Input), which will be split among the various stations. The output of the ProduceACar operation is a car (CarT Output).

The CAP **operation** construct specifies textually the *flowchart* for the ProduceACar operation. The RawMaterialsT are divided and given to the appropriate stations, namely the body, engine, frame and interior stations, which produce body parts, engine, frame and interior respectively. Once all four stations have completed their work, they send their parts to the integration station. The integration station integrates the parts to produce the final car.

An important issue is how the RawMaterialsT are divided between the various stations. In this example, the RawMaterialsT are a box containing four packets of raw materials. The body station takes the first packet, the engine station the second and so on.

Operation inputs and outputs model the data sets that move from one station to the other. Stations have input queues containing *orders*. An order consists of an input data set, an operation to be performed on the input data set and information about the output data set destination. The output data set destinations are derived automatically from the operation flowchart specifications.

Pipelining is the basis of the semantics of the CAP language. Assuming that (1) several cars are produced and (2) all stations produce or integrate their parts in roughly the same time, the timing diagram corresponding to a production run of several cars is shown in Figure 3.

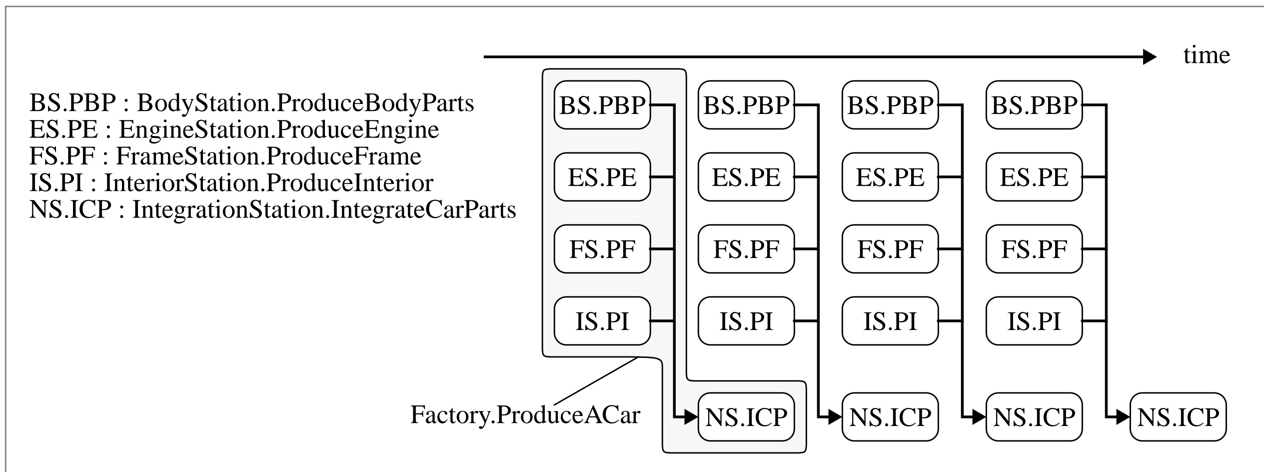

FIGURE 3. Timing diagram for a pipelined car-production run

4 The CAP language design

CAP simplifies the task of programming explicit message-based systems by providing a parallel-program model based on the factory analogy. Each process is seen as a station in a factory, with an order input queue. Each station can execute *orders*, consisting of input data, an operation to perform, and a destination for the output data. Composite operations are specified as a *flowchart* of orders. A CAP specification consists of (a) station specifications (process construct) ; (b) operation specification (operation construct) ; (c) architecture specification ; and (d) mapping between stations and architecture components. The CAP preprocessor transforms the CAP specification into a set of programs running on each of the processor in the target architecture. The CAP compiler takes over memory allocation and communication. The set of programs generated by the CAP preprocessor are based on the task-parallel explicit-message-passing paradigm.

CAP is geared toward coarse-grain parallelism and pipelining of operations. It takes into account the transfer of data to the processing elements, to the client and to secondary storage. CAP takes advantage of

the fact that communication can be executed in pipeline with sequential operation. Provided that communication takes less than execution time, communication can be made completely transparent.

Rather than specifying the allocation of *variables* onto processing elements, the CAP language specifies the allocation of *operations* onto processing elements, and automatically handles the transfer of data between processing elements. Pipelining is the base of the semantics of the CAP language. The CAP compiler generates automatically a set of programs with explicit-channel-communication.

The CAP language supports both shared-memory architectures and distributed memory architectures. In a distributed memory architecture, operation inputs and outputs are used for synchronisation and data transfer. In a shared-memory architecture, operations inputs and outputs are used for synchronisation, whereas shared variables are used for communication. Shared-variables are protected by restricting the number of processes which can access it.

The CAP runtime environment uses the features supported by the underlying architecture. Conversely, it does not make up for the features missing in an architecture. In particular, in a distributed memory-architecture, if the programmer specifies access to a shared variable, the CAP preprocessor will report an error. Program 3 shows the specification of a CAP operation using a shared variable (WindowT window).

A key issue in the design of CAP is flow control, or how to manage station queues without overflow. A window-based flow-control scheme, as used in communication protocols such as TCP/IP, optimizes the memory requirements of CAP-generated programs.

```
leaf operation ProcessT::Merge (WindowT window)
    <-: TileT Input
    >-: void Output
{
    // C++ code to merge a tile into
    // a visualization window
}
```

PROGRAM 3. CAP operation using a shared variable

The CAP environment supports the translation of a CAP specification into a sequential program, allowing to use a conventional debugger to check the algorithm being developed. Its main feature is of course the translation of CAP specifications into a set of separate programs. The code generated by the CAP preprocessor is C++ code calling various parallel libraries, such as PVM, MPI, or the GigaView parallel file system library [2,3].

The benefits of CAP are :

- Support for pipelining.
- Implicit communication between processes. The communication is generated automatically from the operation construct flowchart specification.
- Hierarchical specification of concurrent behavior. A flowchart specifies the set of suboperations required to achieve an operation. A suboperation can itself be specified as a flowchart or as a sequential program written in C++.
- Compositionality. This is a consequence of the hierarchical nature of CAP. The call to a C++ sequential function and the call to a CAP parallel operation are identical. The former will result in the sequential execution of the function by the current thread.The latter will launch a concurrent execution of the function using existing threads.
- Support for shared-memory and distributed-memory architectures.
- Emphasis on data transfers in general and parallel I/O in particular.

- Deadlock freedom. The specification of a CAP operation is a directed acyclic graph (DAGs), representing the ordering of suboperations required to achieve the operation. Such a specification is both completely general and cycle free : no algorithm requires the result of a latter step to start an earlier step. The lack of cycles ensures deadlock freedom, provided sufficient memory is available. This assertion remains true regardless of the allocation of suboperations to processes.
- No overhead due to thread management. Threads are initialized at the beginning of the parallel program. The allocation of threads to architecture components is static and depends on the architecture. During the execution of a CAP operation, the allocation of suboperation instances to threads is user-defined (derived from CAP operation flowcharts) and dynamic (thread selection can be a function of the suboperation-instance input-data).

5 Comparing CAP with other methodologies.

In this section, we refer for the classification of parallel programming languages and methodologies to the article published by Perrott in Concurrency : Practice and Experience [6].

There are three approaches for parallel programming language design : (1) extend an existing sequential language with features to represent parallelism ; (2) use a sequential language but rely on a compiler to detect which parts of the program can be executed in parallel ; (3) develop a completely new parallel language. The first approach benefits from the programmer's experience in an existing language. The drawback of the second approach is that the programmer must restructure the program for the compiler to generate efficient parallel code. Intimate knowledge of the compiler behavior is required. The third approach requires existing applications to be rewritten. The CAP language falls in the first category : it is an extension language to C++.

Parallel programming languages can broadly divided into two categories : imperative languages and declarative languages. The imperative language group is divided into procedural and object oriented languages. Procedural languages themselves are oriented toward array and vector processing on one hand, and multiprocessing/distributed processing on the other hand. The declarative language group is divided into logic and functional languages. CAP fits in the imperative procedural language category.

The approaches to parallelization in procedural languages are divided in three categories : implicit approach (compiler detection of parallelism) ; explicit approaches for array- and vector-processing ; explicit approach for multiprocessor and distributed systems. The CAP language supports an explicit approach for multiprocessor and distributed systems, based on message passing.

The explicit approaches based on message passing are derived from CSP [18], and supported by some programming languages, such as Ada [19], Occam [21], and SyncC++ [22]; and by most parallel programming libraries, such as PVM and MPI. Programming a parallel application in such a model is recognised to be a difficult task, requiring the programmer to explicitly program the communication channels and the communication pattern between processing elements.

The methodology most similar to CAP is SADT, the Structured Analysis and Design Technique [4,5]. The SADT is a graphical flowchart representation of systems emphasizing data and operations. SADT is geared toward communication and lacks an execution semantics. CAP is a textual and executable representation of a system. CAP draws inspiration from queuing models and their simulation semantics (QNAP [20]). The execution semantics of dataflow languages also has similarities with the execution semantics of CAP : in single-assignment parallel languages, processes wait until required variables are assigned ; in CAP an operation is executed as soon as its input data is available, and its process is ready to execute it.

Other examples of automatic generation of explicit-communication message-based approaches are :

1. The implicit-communication task-parallel methodology combines sequential functions to achieve parallelism [12]. The combination of sequential operations is done either using a separate coordination language (Strand [12], PCN [16]) or a language extension (CC++ [15], FM [17]). All these languages use single assignment variables to enforce synchronisation and achieve data transfers. The semantics of these languages is also based on a single address space. In [12], Foster insists on compositionality, which is a feature of CAP.
2. Skeleton-based programming environments also help generate message-passing-based parallel programs [11,7,8,9,10]. Skeletons support the specification of processes, communication-channels between these processes, and algorithms using the processes and their communication channels.

6 CAP specification of a medical-imaging application

The medical imaging application we consider in this proposal is a browser through a 3-D image resulting from a CAT (computer-aided tomography) scan or MRI (magnetic resonance imaging). The 3-D image is divided in 3-D tiles stored on each of the storage node of the GigaServer. The application extracts 2-D visualization windows out of the 3-D data. The window can have any orientation and position inside the 3-D data.

To perform the window-extraction from the 3-D data, the storage server allocates room in shared memory for the visualization window, and distributes the visualization window description to all intelligent disk-nodes in the architecture. Each disk-node processor reads from its storage devices the tiles intersecting the visualization window, computes the intersection of the visualization window with each tile, and copies each tile intersection in the visualization window allocated in shared memory.

For the medical imaging application, we consider a storage server consisting of 4 intelligent disk-nodes. To pursue the analogy started in section 3, our “factory” consists of 8 processes : 4 disk-access stations, and 4 processing stations. One disk-access station and one processing station are allocated on each disk-node.Extracting the visualization window consists of broadcasting the window request to all disk-nodes in the server. Each disk-node performs a two-stage pipeline : the disk-access station reads tiles from the storage device(s) ; the processing station computes the intersection of the tiles with the visualization window, and writes the intersection in shared memory. Figure 4 illustrates the computation of the intersection of the visualization window with a tiled 3D image. In Figure 4, wired cubes represent tile boundaries ; triangles and hexagons represent the intersection of the visualization window with each tile. The color of the triangles and hexagons indicate which of the four disk-node computed the intersection of the tile with the visualization window. The white square represents the visualization window required by the client.

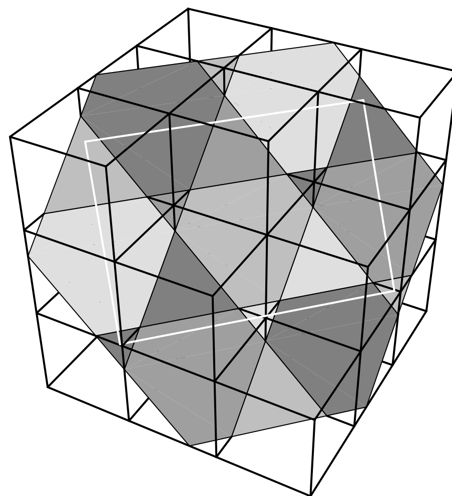

FIGURE 4. Intersection of a visualization window with a tiled 3D image

The CAP specification of the visualization-window extraction operation is shown in Programs 4 to 10. Program 4 shows the GigaServer stations : 4 disk-access stations, and 4 processing stations. The GigaServer can perform a read operation, taking as input a window request, and producing a visualization window as output. The Gigaserver read operation (program 5) is a pipelined operation. The GigaServer divides a window request into multiple tile requests and sends the requests to the appropriate disk-access station (`Disk[Tile.Disk]`). The disk-access station reads the tile, and sends it to the processing station on the same disk-node. The processing station computes the intersection of the visualization window with the tile, and writes it to the output window.

```
process GigaServerT {
processes :
  DiskAccessStationT    Disk[4] ;
  ProcessingStationT    Station[4] ;
operations :
  Read   <-: WindowRequestT Input
         >-: WindowT Output ;
} ;
```

PROGRAM 4. GigaServerT specification

```
operation GigaServerT::Read
    <-: WindowRequestT Input
    >-: WindowT Output
{
  pipeline (DivideWindow)
    ( Disk[Tile.Disk].ReadTile
    , Station[Tile.Disk].Intersect(Output)
    ) ;
}
```

PROGRAM 5. Read operation specification

In Programs 6 and 7, the interface of the disk-access and processing stations are specified. The disk-access station can perform a read-3D-tile operation. The read-3D-tile operation is a sequential operation written in C++ using the GigaServer file-system programming interface. The processing station can compute the intersection of a 3-D tile with a planar window, and produce a 2-D tile as output. In Program 8, the interface of the DivideWindow function is specified. This is a sequential C++ function. It is used by the pipeline construct to divide a global planar window request into 3D-tile window requests.

```
process DiskAccessStationT {
operations :
  ReadTile  <-: Tile3DRequestT Tile
            >-: Tile3DT Output ;
} ;
```

PROGRAM 6. DiskAccessStationT specification

```
process ProcessStationT {
operations :
  Intersect (WindowT Window)
    <-: Tile3DT Tile
    >-: Tile2DT Output ;
} ;
```

PROGRAM 7. ProcessStation specification

```
void DivideWindow
  ( WindowRequestT* windowRequestP
  , Tile3DRequestT* previousTileP
  , Tile3DRequestT* nextTileP
  ) ;
```

PROGRAM 8. DivideWindow intreface specification

The body of the ProcessStationT::Intersect operation is specified in Program 9. The **`leaf operation`** keywords indicate that the body of the operation is specified in C++. The operation initialization parameter (`WindowT window`) makes the window output of the GigaServerT::Read operation visible inside the ProcessStationT::Intersect operation body. The difference between initialization parameters and inputs is that inputs carry a synchronization semantics. An operation is started as soon as its input is available. The initialization parameter are assumed to be available when the operation is started.

```
leaf operation ProcessStationT::Intersect
      ( WindowT window )
    <-: Tile3DT Tile
    >-: Tile2DT Output

{
    // a lot of sequential C++ code here
}
```

PROGRAM 9. Intersect operation specification

```
#include "gigaserver.h"

GigaServerT GigaServer ;

main ()

{ WindowRequestT windowRequest (...) ;
  Window window ;
  call GigaServer.Read
    <-: windowRequest
    >-: window ;
}
```

PROGRAM 10. Main program specification

The CAP specifications are translated into separate C++ programs running on the different nodes of the GigaServer architecture. To execute one of the GigaServerT operations, the application must instantiate a GigaServerT, and call one of the GigaServerT operations using the CAP call statement and the familiar member function notation (Program 10).

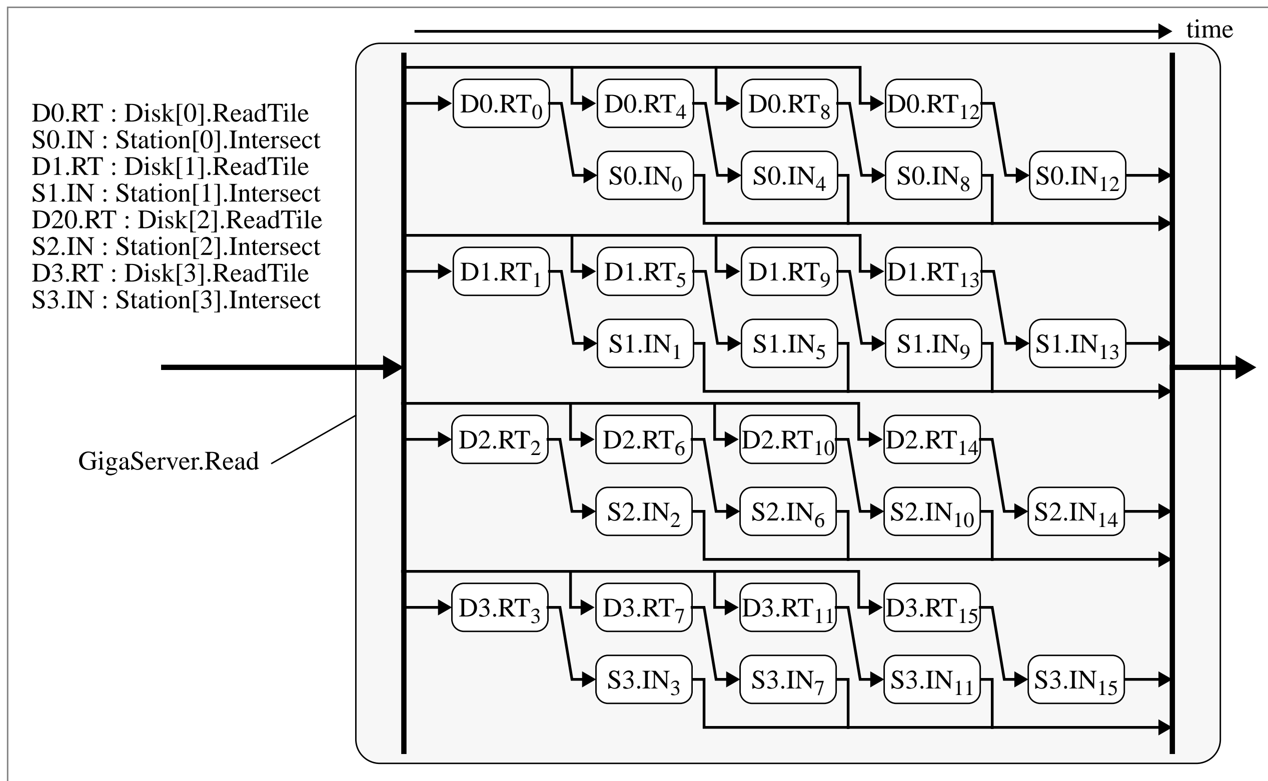

FIGURE 5. Timing diagram for a pipelined 2-D window extraction from a 3-D image (GigaServerT::Read operation)

A timing diagram resulting from the execution of the CAP specification shown in programs 4 to 10 is shown in Figure 6. The timing diagram assumes that all operations take roughly the same time, and that, although the DiskAccessStation and the ProcessStation processes run on the same processor, the actual disk access and an intersection operation can be performed concurrently.

7 CAP project status and conclusion

This contribution describes the CAP language extension to C++. The CAP extension supports the specification of distributed pipelined applications, using the model of a factory. Currently, the CAP prototype environment translates CAP extensions to sequential C++ code, and pseudo-parallel C++ code, for a few simple examples. The automatic translation to sequential code allows the use of existing C++ environment tools (compiler, debugger) to validate the CAP specification. The automatic translation to pseudo-parallel code allows to refine the design of the CAP language. Hand translations of CAP specifications exist already. They are based on the current version of the distributed MDFS file system running on a network of T800 transputers.

Under way is the automatic translation of CAP into a distributed application communicating through the PVM library, and the translation of CAP into a parallel application communicating through shared memory (file system on a Sparc20-51).

We have shown [3] that distributed architecture offer not only better scalability, but also better performance (with equal hardware) than shared-memory architectures with generic caching schemes. However,

distributed-memory applications are difficult to program. The CAP extension to C++ provides elements of solutions to the problem of specifying pipelined parallel applications on a distributed memory architecture.

References

[1] All L. Drapeau et al. RAID-II : A high-bandwidth network file server. In *Proc. 21th Symp. Computer Architecture*, pages 234-244, Chicago, Illinois, 1994.

[2] B. A. Gennart and R. D. Hersch. Multimedia performance behavior of the GigaView parallel image server. In *Proc. 13th IEEE Symposium on Mass Storage System*, IEEE press, p. 90-98, Annecy, 1994.

[3] B. A. Gennart and R. D. Hersch. Comparing multimedia storage architectures. In *Proc. Int. Conf. on Multimedia Computing and Systems*, IEEE Press, p. 323-329, Washington 1995.

[4] David A. Marca and Clement L. McGowan. *SADT : Structured Analysis and Design Technique*. McGraw-Hill, New-York, 1988.

[5] I.G.L. Technology. *SADT : un langage pour communiquer*. Eyrolles, Paris, 1989.

[6] R. H. Perrott. Parallell language developments in Europe : an overview. Concurrency : practice and experience, vol. 4(8), p. 589-617, December 1992.

[7] Helmar Burkhart, Robert Frank, Guido Hächler and Peter Ohnacker. Structured parallel programming : how informatics can help overcome the software dilemma. In *Proc. Priority Program Informatics Research, Information Conference Module 3, Massively parallel systems*, p. 129-138. November 1994, Zürich.

[8] Helmar Burkhart and Stephan Gutzwiller. Steps towards reusability and portability in parallel programming. In *Proc. IFIP Working Conference WG10.3 on Programming Environments for Massively Parallel Distributed Systems*, p. 147-157, Birkhäuser, 1994.

[9] Karsten M. Decker, Jiri J. Dvorak and René M. Rehmann. User-driven development of a novel programming environment for distributed-memory parallel programming systems. In *Proc. Priority Program Informatics Research, Information Conference Module 3, Massively parallel systems*, p. 40-47. November 1994, Zürich.

[10] C. Clémençon, K. M. Decker, A. Endo, J. Fritscher, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturker, B. J. N. Wylie, and F. Zimmermann. Application-driven development of an integrated tool environment for distributed-memory parallel processors. In Ramesh Rao and C. P. Ravikumar, editors, *Proc. of the First Int. Workshop on Parallel Processing* (Bangalore, India, December 27-30). IEEE, December 1994.

[11] M. Cole. *Algorithmic skeletons : structured management of parallel computation*. MIT Press, Cambridge, Massachussets, 1989.

[12] Ian Foster and Carl Kesselmann. Language constructs and runtime systems for compositional parallel programming. In *Proc. COMPAR94 - VAPP VI* (B. Buchberger and J. Volkert, Eds.). LCNS 854, Springer-Verlag, p. 5-16, Sep. 1994.

[13] Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming : the PCN approach. In *Scientific Programming*, Vol. 1, p. 51-66, 1992.

[14] C. Koelbel, D. Loveman, R. Schreiber, G. Steele and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[15] K. M. Chandy and C. Kesselman. CC++ : a declarative concurrent object-oriented programming notation. *Research directions in Object Oriented Programming*. MIT Press, 1993.

[16] K. M. Chandy and S. Taylor. *An introduction to parallel programming*. Jones and Bartlett (1992).

[17] I. Foster and K. M. Chandy. Fortran M : a language for modular parallel programming. In *Journal of Parallel and Distributed Programming*.

[18] C. A. R. Hoare. Communicating sequential processes. Prentice Hall, 1984.

[19] American National Standard Institute. The programming language Ada reference manual. Lecture Notes in Compute Science 614. Springer-Verlag, 1983.

[20] Simulog. QNAP2 User Manual.

[21] Inmos Ltd. *Occam Programming Manual*. Prentica-Hall, Englewood Cliffs, New Jersey.

[22] G. Call and A. Divin. Implementing real-time applications with concurrent objects. In Proc. 6th EuroMicro Workshop on Real-Time Systems, p. 92-97. June 1994.