# Performances of the PS$^2$ Parallel Storage and Processing System for Tomographic Image Visualization

V. Messerli, B. Gennart, R.D. Hersch
Ecole Polytechnique Fédérale, Lausanne
{messerli,gennart,hersch}@di.epfl.ch

**Abstract.** *We propose a new approach for developing parallel I/O- and compute-intensive applications. At a high level of abstraction, a macro data flow description describes how processing and disk access operations are combined. This high-level description (CAP) is precompiled into compilable and executable C++ source language. Parallel file system components specified by CAP are offered as reusable CAP operations. Low-level parallel file system components can, thanks to the CAP formalism, be combined with processing operations in order to yield efficient pipelined parallel I/O and compute intensive programs.*
*The underlying parallel system is based on commodity components (PentiumPro processors, Fast Ethernet) and runs on top of WindowsNT. The CAP-based parallel program development approach is applied to the development of an I/O and processing intensive tomographic 3D image visualization application. Configurations range from a single PentiumPro 1-disk system to a four PentiumPro 27-disk system. We show that performances scale well when increasing the number of processors and disks. With the largest configuration, the system is able to extract in parallel and project into the display space between three and four 512x512 images per second. The images may have any orientation and are extracted from a 100 MByte 3D tomographic image striped over the available set of disks.*

## 1 Introduction

Breaking the I/O bottleneck in parallel processing systems requires more than a RAID disk array hooked on one or several SCSI strings. In order to build parallel applications with high I/O bandwidth requirements, multiprocessor file systems are needed, which decluster files over many disks [2,9,10,7,8,3]. Processes running on any of the available processors may independently access and process parts of the declustered file.

However the availability of a multiprocessor file system does not ensure the development of efficient parallel applications. In order to be truly efficient and to avoid data transfer overheads, processing operations should be located as close as possible to the disks containing the required data file parts. There is an inherent contradiction between a multiprocessor file system hiding the location of data file parts and the knowledge required in order to implement efficient I/O intensive parallel programs.

In order to address this problem, Huber et al. [6] developed a portable parallel file system offering "malleable access", enabling application processes to control the data layout of files over servers and the server's prefetching policies. The developers of the Bridge multiprocessor file system [2], also aware of the problem, introduced the notion of tools. Tools are applications which can be embedded into the parallel file system. Tools may use low-level file system calls giving them information about the locations of data file parts and accordingly create processes on processing nodes close to the disks where file parts are effectively located.

One further difficulty when developing parallel I/O intensive operations involving different processes running on different processors is the necessity to define and implement application specific protocols in order to exchange parameters and data between different processors which do not share common memory. This considerably slows down the development of parallel applications.

We present in this contribution a novel parallel storage and processing system (PS$^2$) which enables the combination of storage and processing operations. The underlying hardware architecture consists of a number of interconnected *storage/processing nodes*[1] (per node: one PentiumPro processor connected to several disks) and a *client node* (one PentiumPro processor with a display

1. The nodes are called storage/processing nodes, since the same processor runs both the ExtentServer threads executing disk access operations and the ComputerServer thread executing processing operations.

Proc. 1997 Int'l Conference on Parallel and Distributed Systems, Seoul, Korea, Dec. 1997, 514-522

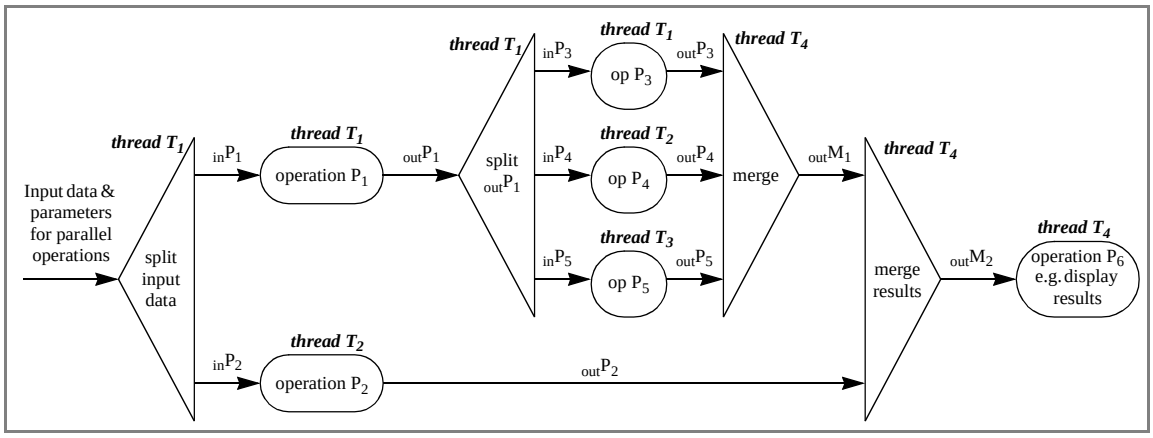



**FIGURE 1. Graphical CAP macro dataflow specification**

interface). The client node and the storage/processing nodes are interconnected by a high-speed network (100Mbits/s Fast Ethernet). The parallel file system is built on top of the native WindowsNT local file system running in each storage/processing node. Global files are declustered into local files residing in the different storage/processing nodes. A computer-aided parallelization tool (CAP) is used to specify at a high level of abstraction, sequences of processing and input-output operations to be executed in a pipelined-parallel manner. CAP automatically compiles the parallel target application, given the sequential code of the contributing operations and a formal description of the macro dataflow between the different operations executable on the available processors.

Applications which need to achieve the highest possible performance with a pipeline composed by parallel I/O and by overlapped parallel computations have to provide an application-specific mapping between *file stripe parts* and storage/processing nodes. File stripe parts, also called *extents*, are indivisible objects, stored as a continuous set of bytes on a single local file (single disk) and provide the basic data unit for processing operations. When operations need to be performed on file extents residing on disks, the application specifies the set of required file extents and provides a mapping function for locating the corresponding storage/processing nodes. CAP enables access and processing requests to be sent in a pipelined parallel manner to these storage/processing nodes.

As an application example, we consider parallel plane extraction from 3D tomographic images. We show how the application is created using the CAP formalism in conjunction with the available basic parallel file system operations. We measure and analyze the application's performances under various configurations and identify the system's bottlenecks.

## 2 The computer-aided parallelization framework

In order to speedup the development of parallel applications and to specify parallel I/O and processing operations at a high level of abstraction, we use the Computer-Aided Parallelization (CAP) tool. This tool enables application programmers to hierachically specify the *macro dataflow* between *operations* performed on extents (file stripe parts). Operations are segments of sequential code performed by a single execution thread and characterized by input value and output values. The input and output values of an operation are called *tokens*. In the context of this paper, tokens consist of extent data and additional application-dependent parameters. The macro dataflow specifies how tokens are routed between the operations of the parallel program. In addition, synchronization points (also used for merging intermediate results) specify which tokens must be available before the next operation can start (Figure 1).

In a graphical CAP specification, parallel operations are displayed as parallel horizontal branches, pipelined operations are operations located in the same horizontal branch. Figure 1 assumes a parallel program consisting of 4 threads $T_1$, $T_2$, $T_3$, and $T_4$. In the macro data flow graph of Figure 1, the input token enters the graph from the left. It is divided into two parts, ${}_{in}P_1$ and ${}_{in}P_2$, which undergo operations $P_1$ and $P_2$. Operation $P_1$ is performed by thread $T_1$. Operation $P_2$ is performed by thread $T_2$. The result of operation $P_1$ is ${}_{out}P_1$. ${}_{out}P_1$ is divided into three tokens ${}_{in}P_3$, ${}_{in}P_4$, ${}_{in}P_5$, which undergo operations $P_3$, $P_4$ and $P_5$ in parallel (threads $T_1$, $T_2$ and $T_3$). The results of operations $P_3$, $P_4$ and $P_5$ are merged into a single token, ${}_{out}M_1$, which is in turn merged with ${}_{out}P_2$ to form ${}_{out}M_2$. ${}_{out}M_2$ is fed to operation $P_6$. If several tokens enter the macro data flow

graph of Figure 1, they are processed in a pipelined fashion.

The CAP specification of a parallel program is described in a simple formal language, an extension of C++. This specification is translated automatically into a C++ source program. At program startup time, the CAP runtime allocates the program threads to the available processors, using the information stored in a configuration file [4]. The macro data flow model which underlies the CAP approach has also been used successfully by the creators of the MENTAT parallel programming language [5].

Thanks to the automatic compilation of the parallel application, the application programmer does not need to explicitly program the protocols to exchange data between parallel processes and to ensure their synchronization. Furthermore, predefined library operations are available, for example for parallel file storage and access operations. Combining CAP parallel disk access and processing operations enables the customization of the parallel file system according to the application's requirements.

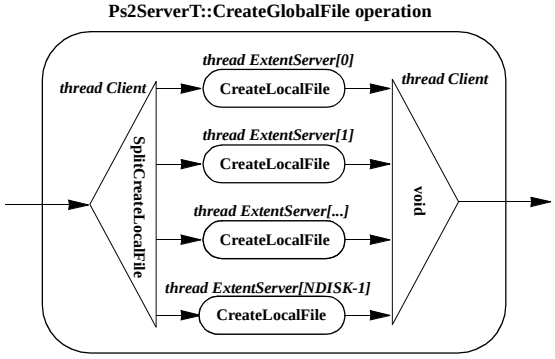
## 3 CAP-based synthesis of parallel file operations

In the context of this paper, the parallel storage and processing system (Figure 2) consists of a client thread running on the client node (line 3) and two sets of threads running on the storage/processing nodes (line 4, 5). The ExtentServer threads perform I/O operations and the ComputeServer threads perform computations on the extents extracted from the disks. Each storage/processing node comprises one ComputeServer thread and as many ExtentServer threads as disks.

```
1 process Ps2ServerT {
2 subprocesses:
3   ClientProcessT Client;
4   ExtentServerT ExtentServer[NDISK];    // NDISK:total nb of disks
5   ComputeServerT ComputeServer[NSTOR]; // NSTOR:total nb of
6 operations                              // storage/processing nodes
7   PlaneExtraction
8     in PlaneExtractionParametersT Input out PlaneT Output;
9   CreateGlobalFile
10    in GlobalNameT Input out void Output;
11  ...
12};
13
14process ExtentServerT {
15operations:
16  CreateLocalFile
17    in LocalNameT Input out void Output;
18  ReadExtent
19    in ExtentReadingRequestT Input out ExtentT Output;
20  WriteExtent
21    in ExtentWritingRequestT Input out void Output;
22  ...
23};
24
25process ComputeServerT {
26operations:
27  PlanePartExtraction
28    in ExtentT Input out PlanePartT Output;
29};
```

**FIGURE 2. Parallel storage and processing system threads**

Figure 3 shows as an example the graphical and formal specification of the *Ps2ServerT::CreateGlobalFile* parallel operation. The input of the dataflow graph is a *GlobalNameT* token, containing the name of the global file to be created. The *SplitCreateLocalFile* routine divides the input token into *LocalNameT* subtokens, containing the name of each of the local files making up the global file. The *LocalFileT* subtokens are routed to the appropriate ExtentServer threads, which create the local files using the API Win32 CreateFile call. The *ExtentServerT::CreateLocalFile* sequential operation performed by each ExtentServer thread returns a void token used for synchronization purposes. This behavior is modeled in CAP using a parallel loop (index parallel construct) initialized by the *SplitCreateLocalFile* routine which generates the *LocalNameT* tokens. These tokens are sent in parallel mode to *ExtentServerT::CreateLocalFile* sequential operations. The *ExtentServerT::CreateLocalFile* sequential operations are performed by the ExtentServer threads running on the storage/processing nodes.




```
void SplitCreateLocalFile
  (GlobalNameT* FromP, LocalNameT* ThisP, int ExtSrvIndex)
{ // C++ code
}

leaf operation ExtentServerT::CreateLocalFile
  in LocalNameT Input out void Output
{ // C++ code
  WindowsNT::CreateFile(Input.Path, ...);
}

operation Ps2ServerT::CreateGlobalFile
  in GlobalNameT Input
  out void Output
{ // CAP code
  indexed
    (int ExtSrvIndex = 0; ExtSrvIndex < NDISK; ExtSrvIndex++)
  parallel (SplitCreateLocalFile, void, Client, void Result)
    (ExtentServer[ExtSrvIndex].CreateLocalFile);
};
```

**FIGURE 3. CAP specification of the parallel Ps2ServerT::CreateGlobalFile operation**

From now on, the parallel operation *Ps2ServerT::CreateGlobalFile* is known and can be used in CAP programs. This operation can also be incorporated

into a C++ parallel file system library (Figure 4) and be called from any C++ program.

```
Ps2ServerT Ps2Server;

void ps2CreateGlobalFile(const char* PathP, ...)
{ // create input token and call the Ps2 server
  GlobalNameT* InputP = new GlobalNameT(PathP, ...);
  void* OutputP; // Ptr to output token
  call Ps2Server.CreateGlobalFile in InputP out OutputP;
}
```

**FIGURE 4. CAP operation incorporated into a C++ library**

## 4 The parallel file system support

The parallel file system used for parallel access and processing applications differs from other parallel file systems [6,7,1] by the fact that its parallel operations are programmed in the CAP formalism and that it offers low-level operations such as *ReadExtent* or *WriteExtent* as CAP operations, usable as building blocks for the definition and automatic synthesis of parallel I/O and compute intensive programs.

As in other parallel file systems, global files are striped over several local files, each of which is stored on a different disk by the native local file system, i.e. NTFS. For each global file, there is one local file per contributing disk. A global directory maintains the information for accessing individual local files which have the same name as the global file. Each local file contains the set of extents stored on its corresponding disk and a table whose entries specify the size and the storage location of extents, i.e. their byte address within the corresponding local file.

Extent server sequential operations (Figure 2, line 16-21) such as *ExtentServerT::ReadExtent* or *ExtentServerT::WriteExtent* run on extent server threads: there is one extent server thread per disk. Extent server threads may run in the same or in distinct address spaces, depending whether their corresponding disks are hooked onto the same shared memory system or on systems interconnected by the communication network.

The basic parallel file system functions are available to parallel application programmers as CAP operations usable as building blocks. The input and output tokens for these operations comprise the parameters described in Figure 5. Additional high-level parallel file system calls having the same semantics as traditional standard file system calls (*pfsCreateFile*, *pfsOpenFile*, *pfsSeek*, *pfsWriteFile*, *pfsReadFile*, ...) are also available as C++ library calls. However, since these file system calls hide the way data is distributed across the disks, parallel computation operations making use of these file system calls may induce considerable I/O access and communication overhead. In contrast, parallel applications described by CAP and making use of the basic parallel file system operations (Figure 5) know precisely the distribution of extents across disks, and can therefore create pipelined parallel data processing sequences where processing operations are executed on the same nodes as the corresponding data access operations.

```
Ps2ServerT::CreateGlobalFile
  in  { const char* PathP,
        int StripeFactor,
        const int* ExtentServerIndexTableP }
  out { void };
Ps2ServerT::OpenGlobalFile
  in  { const char* PathP,
        int OpenFlags }
  out { int GlobalFileDescriptor,
        int StripeFactor,
        const int* ExtentServerIndexTableP,
        const int* LocalFileDescriptorTableP };
Ps2ServerT::CloseGlobalFile
  in  { int GlobalFileDescriptor }
  out { void };
Ps2ServerT::DeleteGlobalFile
  in  { const char* PathP }
  out { void };
Ps2ServerT::CreateGlobalDirectory
  in  { const char* PathP }
  out { void };
Ps2ServerT::DeleteGlobalDirectory
  in  { const char* PathP }
  out { void };
Ps2ServerT::ListGlobalDirectory
  in  { const char* PathP }
  out { int ListSize,
        const DirEntryT* EntryTableP };
ExtentServerT::ReadExtent
  in  { int LocalFileDescriptor,
        int ExtentServerIndex,
        int LocalExtentIndex }
  out { int ExtentSize,
        char* ExtentP };
ExtentServerT::WriteExtent
  in  { int LocalFileDescriptor,
        int ExtentServerIndex,
        int LocalExtentIndex,
        int ExtentSize,
        const char* ExtentP }
  out { void };
```

**FIGURE 5. Library of reusable CAP operations**

When creating a global file, one must provide a path name, a stripe factor and a table of extent server indexes specifying the number of disks, i.e. local files, and the disks across which the global file is declustered. When opening a global file, the system returns the global file descriptor, the stripe factor, the table of extent server indexes and the table of local file descriptors so as to enable the *ExtentServerT::ReadExtent* or *ExtentServerT::WriteExtent* sequential operations to be called.

## 5 CAP specification of the tomographic image visualization application

In order to compare predicted and measured performances, we consider a real parallel and I/O intensive

application: the extraction of image planes from 3D tomographic images (Figure 6).

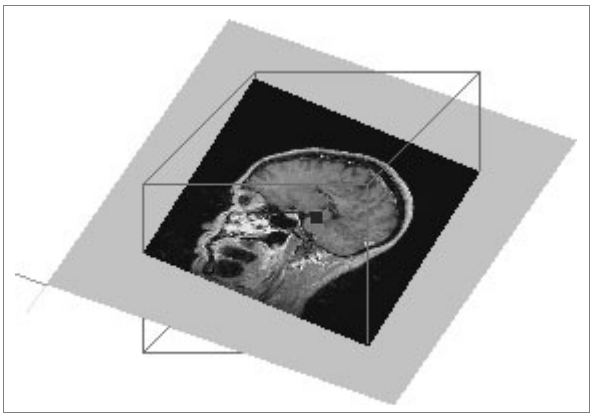


**FIGURE 6. Specification of freely oriented plane to be extracted from a 3D volume**

The considered 3D tomographic images each consist of a volume of size 512x512x384, requiring approximately 100MB storage space. In order to stripe a tomographic image across a set of storage/processing nodes, the image volume is divided into small cubes containing each $32^3$ pixels. These cubes form the extents, which are distributed across the set of available disks. The distribution of extents to disks is made so as to ensure that direct cube neighbors reside on different disks. We achieve such a distribution by introducing, between two successive rows of extents and between two successive planes of extents, offsets which are prime to the number of disks. This enables planes having any orientation to intersect extents which are close to uniformly distributed across the disks.

In order to visualize a plane having an arbitrary orientation, the following operations need to be performed: (1) the extents intersecting the desired plane are computed, (2) extents intersecting the plane are read from the disks, (3) their respective plane parts are extracted and projected into the screen space and (4) all projected plane parts are merged into the displayed image.

Figure 7 gives the graphical description of the CAP program specifying the parallel extraction of a plane from a 3D tomographic image and its display.

Pipelining is achieved at three levels:

- plane extraction and projection is performed by the compute server thread on one extent while extent server threads read the next extents
- an extracted and projected plane part is merged by the client thread into the final display buffer while the next plane part is being extracted
- a full plane is displayed by the client thread while the next full plane is being prepared (in the case the user has requested a series of successive planes)

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 plane parts from extents and projection operations can be done in parallel by several processors; the number of processors can be increased to improve the plane part extraction and projection performance.

With the CAP parallel program specification syntax, *ExtentServerT::ReadExtent* and *ExtentServerT::WriteExtent* sequential disk operations can easily be mixed with

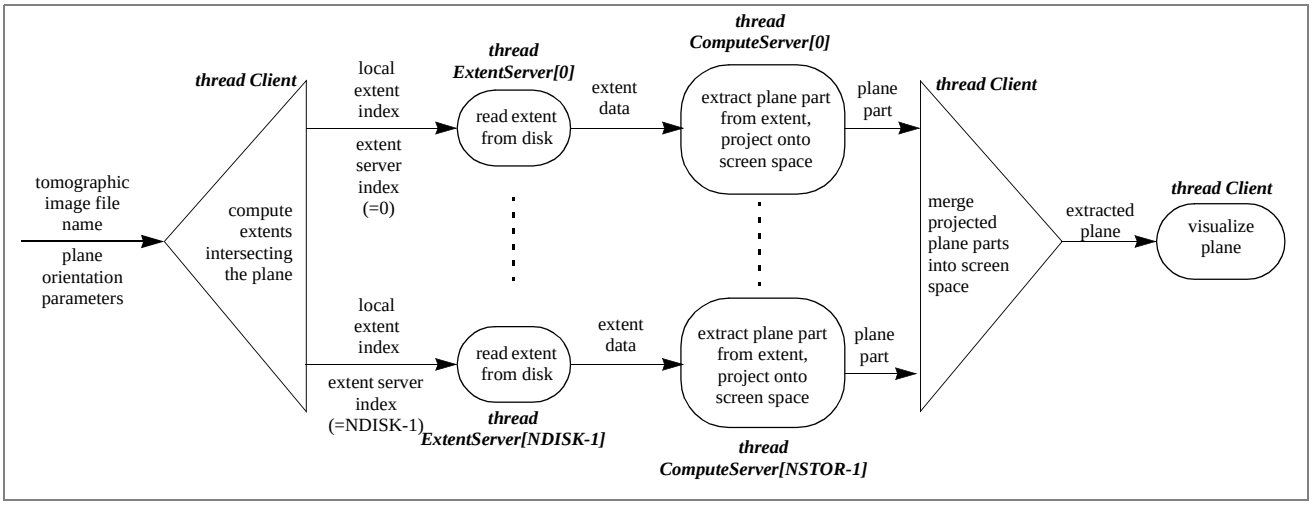



**FIGURE 7. Graphical representation of the pipelined parallel plane extraction and visualization application**

processing operations. For example, the pipelined-parallel plane extraction and visualization application (Figure 7) is specified by the CAP program detailed in Figure 8. The input of the parallel *Ps2ServerT::PlaneExtraction* operation performed by the Client thread is a *PlaneExtractionParametersT* token. This token is divided by the *SplitPlaneParameters* routine into several extent reading requests. The appropriate ExtentServer threads read the required 3-D extents from the disks and feed the extent data to their companion ComputeServer thread. The ComputeServer thread extracts sequentially plane parts from the extents received from the ExtentServer threads, and returns them to the Client thread who originally started the operation. The Client thread merges the plane parts into a single PlaneT token using the *MergePlaneParts* routine. In this specification, ExtentServer threads and their companion ComputeServer thread work in pipeline; ExtentServer and ComputerServer threads work in parallel.

```
int SplitPlaneParameters(PlaneExtractionParametersT* FromP,
                         ExtentReadingRequestT* ThisP)
{ // C++ code }

void MergePlaneParts(PlaneT* IntoP, PlanePartT* FromP)
{ // C++ code }

operation Ps2ServerT::PlaneExtraction
  in  PlaneExtractionParametersT Input
  out PlaneT Output
{
  pipeline(SplitPlaneParameters, MergePlaneParts, Client,
           PlaneT Output)
  (
    ExtentServer[thisTokenP->ExtentServerIndex].ReadExtent
    >->
    ComputeServer[thisTokenP->ExtentServerIndex*
                  NSTOR/NDISK].PlanePartExtraction
  );
}
```

**FIGURE 8. CAP specification of the pipelined-parallel plane extraction and visualization application**

## 6 Performances of PS$^2$ running on PC-based multiprocessor system

The PS$^2$ parallel storage and processing system is executed on top of a multiprocessor system made up of a network of 200MHz PentiumPro PCs, each node running the WindowsNT 4.0 operating system. The PCs are interconnected by a Fast Ethernet network (100Mbits/s). On the slave PCs (storage/processing nodes) we use the 3COM 3C595-TX Fast Ethernet PCI adapters and on the master PC (client node) we use the SMC 9332BDT Fast Ethernet PCI adapter[2].

A TCP-IP socket-based communication library called MPS implements the *SendMessage* and *ReceiveMessage* primitives enabling messages to be sent from an application program memory of one PC to an application program memory located on a second PC, with at most one intermediate memory to memory copy at the receiving site.

Each PC can incorporate up to 5 SCSI-2 strings each offering a maximum nominal throughput of 10MBytes/s. We use IBM-DPES 31080 disks which have a measured data transfer throughput of 3.3MBytes/s and a measured latency (seek time + rotation time) of 18.5ms. Throughput and latency values have been obtained by accessing blocks randomly distributed over the whole disk. When accessing 32KB blocks located at random disk locations, an effective throughput of 1.1 MBytes/s per disk is reached.

Our goal is to measure separatly the throughputs of the system's components (disks and bus/memory subsystem), to identify potential bottlenecks and to measure global performances for various system configurations. The experiment consists of 60 pipelined parallel 512x512 plane extractions during which 18'848 extents of 32KBytes, i.e. 589 MBytes, are read from disks and 22MBytes of plane parts are produced.

The plane extraction application incorporates the following 4 main steps:

1. Reading all extents intersecting the plane
2. Extracting the plane parts from the volumic extents and projecting them into the screen space (resampling operation)
3. Sending, possibly across the network, the plane parts to the processor executing the merging operation
4. Merging the plane parts into the final displayable screen image.

These four steps are performed in pipeline. Let us consider the case where a physician is interested in visualizing a sequence of adjacent planes, in order to quickly visualize slices through a specific body region. In that case, the four steps of the pipeline are always full, and we can ignore pipeline startup time. The global throughput of this pipeline in terms of displayable number of bytes/s is generally limited by the slowest component which forms the bottleneck.

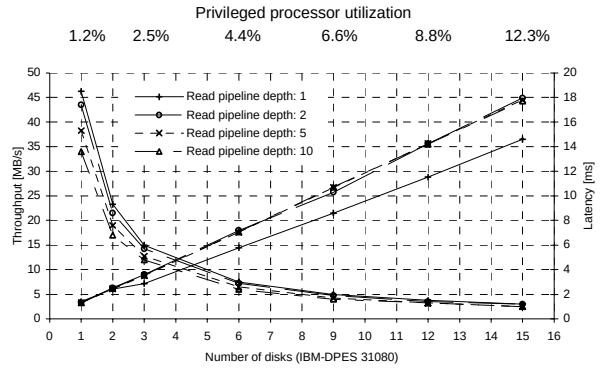
The components we take into consideration are the disks, the bus/memory subsystem, the network and the processors. Each of the four pipeline steps affects one or more of the hardware components. Step 1 (extent reading) affects the disks and the bus/memory subsystem. Steps 2 (plane extraction) and 4 (merging plane parts) affect the processors and the bus/memory subsystem. Step 3 (network transfer) affects the network, the bus/memory subsystem, and the processors (for packetization).

***Disks***. The bottleneck may be formed by the disks. In order to display one image plane of size 512x512 pixels, an average of 314 extents, each of size $32^3$ pixels (32KB), i.e. 9.8MB, need to be retrieved in parallel from the disks.

2. Our experiments show that the SMC Fast Ethernet adapter requires less CPU utilization than the 3COM one.

Disk arrays are described using two numbers: latency and throughput. The approach is to measure the delay when accessing in parallel randomly-distributed blocks striped over *k* disks for increasing block sizes, to linearize the delay using a least-square fit, and to get the formula of the type $DiskAccessTime = Latency + RequestSize / Throughput$.

Figure 9 reports the measured throughputs and latencies for disk array configurations of 1 to 15 disks hooked onto the same PentiumPro PC. The pipeline depth parameter corresponds to the maximum number of outstanding SCSI requests for a specific disk.




| Nb of disks | Nb of SCSI strings | Nb of disks | Nb of SCSI strings |
|---|---|---|---|
| 1 | 1 | 9 | 3 |
| 2 | 1 | 12 | 4 |
| 3 | 1 | 15 | 5 |
| 6 | 2 | | |

**FIGURE 9. Measured disk throughputs and latencies when accessing blocks striped over k disks hooked onto the same PentiumPro PC**

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 blocking disk access requests, i.e. a pipeline depth of a single request. With a pipeline depth of 2 or more requests, the throughput can be increased by a factor 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.

***Bus/Memory subsystem***.The bottleneck may also be formed by the PC's internal bus and memory sub-system. Since all accesses through the bus are read or write operations from or to memory, the bus/memory subsystem is considered as a single potential bottleneck component, simply called bus. If there are simultaneous data transfers from memory to memory (for computations), from the SCSI DMA interface to memory and from memory to the network interface, the internal bus located either within the storage/processing node PCs or the PC executing the merging operation may become a bottleneck. By measuring separately the maximal sustainable bus bandwidth for (1) memory read operations, (2) for disk to memory DMA transfers, one obtains the bus usage time per byte for each of the mentioned data transfers. When several data transfers occur simultaneously, their respective bus usage time per byte are added and the resulting effective bandwidth is computed as one over the total bus usage time per byte.

For memory read operations, bus usage time per byte is directly measured by executing a loop of memory read operations that saturates the bus/memory subsystem. The bus usage time for disk to memory transfers is indirectly measured by simultaneously performing memory read operations and disk to memory transfers and by verifying how much longer the combination of both operations lasts, when compared with simple memory read operations.

The same methodology is used to determine possible bottlenecks due to the combination of processing operations (plane part extraction and projection), disk to memory transfers and memory to network interface transfers. Therefore, bus usage times per byte are also measured for the typical processing operations of our application, i.e. (1) plane part extraction and projection and (2) merging the resulting projected plane parts into the final displayable image (Table 1).

| | |
|---|---|
| Memory read | 14.0ns per byte transferred from main memory to the cache |
| Disk read | 14.4ns per byte transferred from the disk to main memory |
| Plane part extraction & projection | 319.2ns per byte produced |
| Plane part merging | 36.2ns per byte merged |
| Send message | 56ns per byte transferred from main memory to the network interface |
| Receive message | 98ns per byte transferred from the network interface to main memory |

**TABLE 1. Bus usage times per byte for typical operations of our application**

In order to evaluate the worst-case bus usage on a slave PC we consider the 9 disk 3 SCSI string configuration where 9 disks sustain a throughput of approximately (Figure 9, blocking disk access requests, $32_{KB} / [2_{ms} + 32_{KB}/21.5_{MB/s}]$) 9.5 MBytes per second, i.e. 290 extents/s. The plane part extraction operation produces at most 347 KBytes per second (page 6, for 60 plane extractions, 18'848 extents are read and 22MBytes of plane parts are produced). According to the bus usage times in Table 1, during one second of slave processing time, the bus is used for 137ms (disk read) + 113ms (plane part extraction) + 20ms (send message) = 270ms.

The same calculation is carried out for the master PC which has to receive and to merge from 3 different slave

PCs 1041 KBytes per second of extracted plane parts. Therefore for one second of master processing time, the bus is used for 104ms (receive message) + 39ms (plane part merging) = 143ms.

This clearly demonstrates that the bus and the memory subsystem cannot be a potential bottleneck for our specific application. Moreover, our 100 Mbits/s network cannot be a bottleneck either, since in the case of the full-blown configuration (3 storage/processing nodes with each 9 disks), it transfers only 8.1 Mbits/s (1041KBytes/s). Only the disks and the processors are potential bottlenecks.

## 7 Scalability analysis

Figure 10 gives the number of tomographic image planes extracted per second (or a proportional amount of overall disk throughput) with a single PentiumPro processor and a variable number of disks (local) or with one PentiumPro processor acting as the master assembling plane parts (client node) into the final image and one PentiumPro processor acting as the slave processor (storage/processing node) reading from the disks, extracting and resampling the plane parts. The processor utilization bars show that in all configurations, disk accesses represent the throughput's bottleneck, except in the configuration where the slave processor reads in parallel from 9 disks and sends the extracted and resampled extents to the master processor. In that case, the slave processor is 100% busy, with 30.0% for network and 6.6% for disk interface processing (privileged processor utilization).

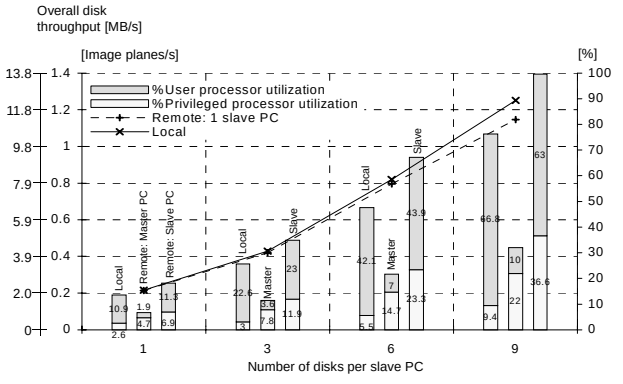



**FIGURE 10. Extracted image planes per second, overall disk throughput and processor utilization for a single processor configuration (local) and for a master - slave two processor system interconnected by Fast Ethernet (PentiumPro processors).**

Figure 11 shows how the system scales, when using more than one slave system (storage/processing node). Globally, the system scales linearly. However, since the disks used in the measures have a constant rotation rate and the number of sectors per track varies in function of the track's position, disk throughput may vary as a function of the position of the tomographic image file on the disks.

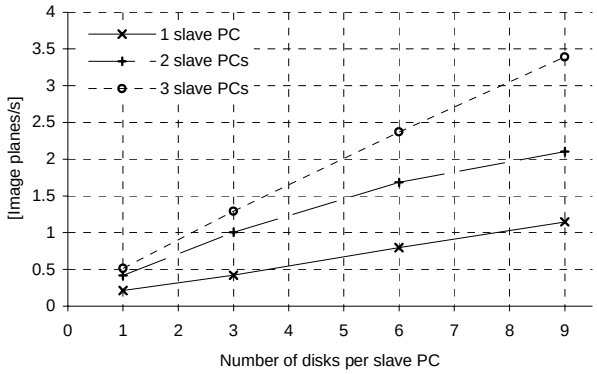



**FIGURE 11. Number of extracted image planes per second when varying both the number of disks per slave processor and the number of slave processors**

In all configurations, where slave processors incorporate 9 disks, slave processor utilization is between 90% and 100% (Figure 12). With 3 slave processors, the master processor is utilized 90% of its time.

Therefore, the current scalability limit of the global system is due to the limited processing power of the master node, which needs to receive plane parts from the slave processors and to assemble them into the final displayable image. At the highest rate of 3.4 image planes per second, the master processor receives on average from the Fast Ethernet network 1068 plane parts per second, i.e. 10.0 Mbits/s, and uses 62.1% of its processing time (privileged processor utilization) only for the network reception operations.

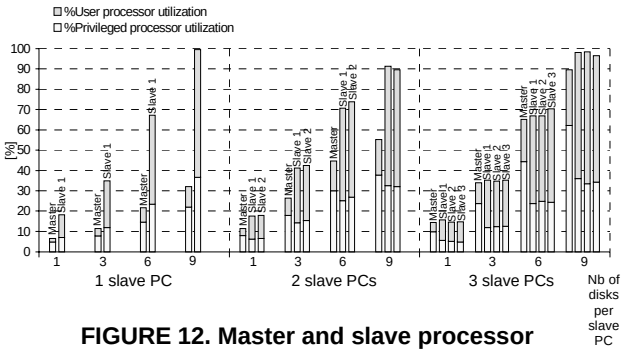



**FIGURE 12. Master and slave processor utilizations for different configurations**

Figure 13 shows that the system scales linearly when the number of slave storage/processing nodes is increased from 1 to 3 slaves. Augmenting the number of disks per

storage/processing node (1, 3, 6 and 9 disks) does not give a linear throughput increase as observed in Figure 9 due to the fact that in the present version of the parallel file system, single disk read extent requests are blocking.

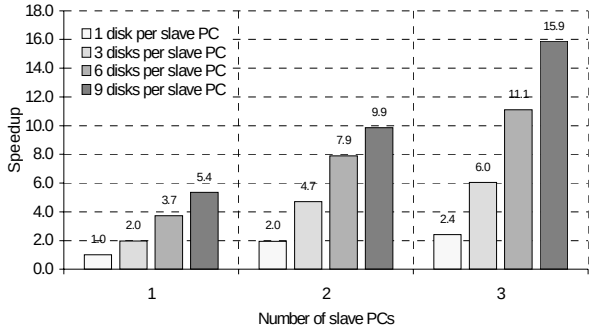



**FIGURE 13. Speedup**

## 8 Conclusions

The computer-aided parallelization tool (CAP) used in conjunction with $PS^2$ basic parallel file system operations enables to easily combine parallel computing and disk access operations, thereby making use of the pipelining and parallelization potential offered by the underlying hardware. We use CAP for creating a real tomographic image visualization application, where discrete planes are extracted in parallel from a large 3D tomographic image. The application runs on a parallel system made of commodity PC PentiumPro computers interconnected by a 100Mbit/s Fast Ethernet network. We measure the application's throughput and analyze for each configuration where the bottleneck resides.

In the case of a single PentiumPro PC connected to 9 disk nodes, the system is balanced: both processor utilization and disk node throughput are close to their maximal values and images can be extracted and displayed at a rate of slightly more than one image per second.

In the case of master-slave configurations, where the master PentiumPro PC (client node) receives plane parts from the network and assembles them into the display image, and where slave processors (storage/processing nodes) read plane parts from disks and project them into the display space, disk access throughput is the bottleneck with up to 9 disks per slave node.

In the case of the full-blown configuration consisting of one master node and 3 slave nodes with 9 disks each, the bottleneck resides both in the limited processing power of the slave nodes and of the master node. The master processing node receives plane parts from the network and assembles them into the displayable image at a speed of 3.4 displayable images per second. The network protocol itself requires approximately 62% of the master processor's computing power.

The next version of the system will incorporate two PentiumPro processors per processing system. With twice the processing power at the master node, we intend to scale the system to 60 disks. Since the slave nodes will also incorporate Bi-PentiumPro processors, they will be able to sustain up to twice the present number of disks. We foresee that a configuration of 12 disks per slave node, i.e. 4 SCSI strings, and 5 slave nodes hooked to a master node will offer at least twice the throughput of today's system, i.e. the ability to access, extract, project and display 7 image planes per second.

However, even the simplest single PentiumPro 9 disk architecture offers, due to parallel disk accesses and pipelining, a much better price/performance ratio than traditional UNIX-based tomographic visualization equipment, which requires preloading the complete tomographic image into main memory.

## References


[1] S. More, A. Choudhary, "MTIO: A multi-threaded parallel I/O system", *Proc. 11th International Parallel Processing Symposium*, April 97, Geneva, IEEE Press, 368-373

[2] P. Dibble, M.L. Scott, C. Ellis, "Bridge: A high-performance file system for parallel processor", *Proc. 8th International Conf. on Distributed Computing Systems*, June 1988, 154-161

[3] D. G. Feitelson, P.F. Corbett, J.P. Prost, "Performances of the Vesta parallel file system", IBM Research report, RC 19760, 1994

[4] B. Gennart, J. Tarraga, R.D. Hersch, "Computer-assisted generation of PVM/C++ programs using CAP", *Proc. Parallel Virtual Machine, EuroPVM'96*, LNCS 1156, Springer Verlag, 259-269

[5] A. S. Grimshaw, "Easy-to-use object-oriented parallel processing with Mentat", IEEE Computer, Vol. 26, No. 5, May 1993, 39-51

[6] J.V. Huber, C.L. Elford, D.A. Reed, A.A. Chien, D.S. Blumenthal, "PPFS: A high-performance portable parallel file system", *Proc. 9th ACM International Conference on Supercomputing*, Barcelona, Spain, 1995, 385-394

[7] D. Kotz, "Multiprocessor File System Interfaces", *Proc. IEEE Conf. on Parallel and Distributed Information Systems*, 1993, 194-201

[8] S.J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E.D. Milne, R. Wheeler, "SFS: A parallel file system for the CM-5", *Proc. Summer USENIX Conf*, June 1993, 291-305

[9] P. Pierce, "A concurrent file system for a highly parallel mass storage system", *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, 1989, 155-160

[10] T.W Pratt, J.C. French, P.M. Dickens, S. A. Janet, "A comparison of the architecture and performance of two paralllel file systems", *Proc. 4th Conf. on Hypercube Concurrent Computers and Applications*, 1989, 161-166