Parallelizing I/O Intensive Image Access & Processing Applications

V. Messerli, O. Figueiredo, B. Gennart, R. D. Hersch
École Polytechnique Fédérale, Lausanne, Switzerland
{gennart,hersch}@di.epfl.ch}

Abstract. We propose a new approach for developing parallel I/O- and compute-intensive applications on distributed memory PC. Using the CAP Computer-Aided Parallelization tool, application programmers create separately the serial program parts and express the parallel behavior of the program at a high level of abstraction. This high-level parallel program description (CAP) is preprocessed into a compilable and executable C++ source parallel program. 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. These programs may run on multiple PC servers offering their access and processing services to clients located over the network.

The applicability of the CAP tools on a real application 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. The image slices are extracted from a 14 GByte color 3D tomographic image striped over the available set of disks. On a 5 Bi-Pentium Pro PC 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 file caching is disabled, an aggregate I/O disk bandwidth of 104 MBytes/s has been obtained. When caching is enabled, an I/O throughput of up to 240 MBytes/s is obtained.

1 Introduction

Image oriented access and processing operations are often both compute and I/O intensive. Making use of a large number of commodity components working in parallel, i.e. parallel processing on several PC's and parallel access to many 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. Programming a parallel application on top of the native operating system (e.g. WindowsNT) or with a message passing system (e.g. MPI [Gropp94] or Fortran M [Foster95, chapter 6]) yields generally synchronous parallel programs, where communications and I/O operations do not overlap with computing operations. Creating parallel programs with completely asynchronous communications and I/O accesses is possible [MPI97], 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 rather expensive SMP computers, i.e. computers whose processors interact via shared memory and synchronization semaphores (for example the SGI Origin 2000 multiprocessor system). 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 disks at a time located on different computers requires appropriate parallel file system support, i.e. means of striping a global file into a set of disks located on different computers and of providing metainformation specifying where the file stripes are located and how the striping is being done.

This contribution presents methods and tools to build parallel applications based on commodity components (PC's, SCSI disks, Fast Ethernet, Windows NT). The use of the presented tools is illustrated by a real 3D medical image server application, the Visible Human Slice Server.

We describe a computer aided parallelization tool (CAP) and parallel file system components that have been developed for facilitating the development of parallel server applications running on distributed memory multi-processors, e.g. PC's connected by Fast-Ethernet. We show how parallel server applications may be automatically generated from a high-level description of threads, operations and of the macro dataflow between operations. Due to the macro data flow nature of CAP, the generated parallel application is completely asynchronous. Each thread incorporates an input token queue ensuring that communication occurs in parallel with computation. In addition, disk access operations are executed asynchronously. The call-back function associated with each disk access operation ensures that the result token is forwarded to the next operation.

IEEE Concurrency, Vol. 7, No. 2, April-June 1999, 28-37

We apply the CAP tool to synthesize the parallel Visible Human Slice Server application (section 2). Thanks to CAP, the generated application is very flexible and additional optional processing steps can be easily added to the parallel program. Evaluation of the Visual Human image slice access times shows that performances are close to the best performance obtainable by the underlying hardware.

2 The Visible Human Slice Server application

Let us introduce the Visible Human Slice Server application which has been synthesized using the Computer-Aided Parallelization tool (CAP) described in section 3.

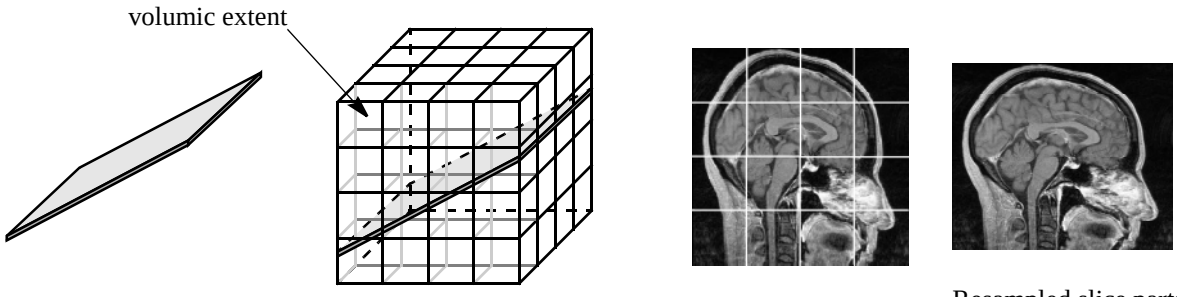
The 24-bit Visible Human Male created by the National Library of Medicine at Bethesda Maryland USA [Ackerman95, Spitzer96] reaches a size of 2048x1216x1878 voxels, i.e. 13 GBytes RGB. The width (X) and height (Y) resolutions are three pixels per millimeter. The axial anatomical images (Z) where obtained at one millimeter 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 server PCs. We achieve such a distribution by storing volumic extent rows on successive disks, successive disks being hooked on successive server PCs, and by introducing between two successive rows and between two successive planes of volumic extents disk offsets ($DiskOffset_Y$ and $DiskOffset_Z$) which are prime to the number of disks (*NDisks*). This ensures that for nearly all extracted slices, disk and server PC accesses are close to uniformly distributed. Equation 1 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 2 gives the distribution of volumic extents within a particular disk computed using Equation 1 (all divisions are integer divisions). Therefore a particular volumic extent is uniquely identified with its *DiskIndex* specifying on which disk the extent resides and its *LocalExtentIndex* specifying the local extent index within the set of extents stored on that particular disk.

$$DiskIndex = (ExtentPosition_Z \cdot \mathrm{DiskOffset_Z} + ExtentPosition_Y \cdot \mathrm{DiskOffset_Y} + ExtentPosition_X) \mod \mathrm{NDisks} \quad (1)$$

$$\begin{aligned} LocalExtentIndex = {} & ExtentPosition_Z \cdot \left[\mathrm{NExtents_Y} \cdot \left(1 + \frac{\mathrm{NExtents_X} - 1}{\mathrm{NDisks}} \right) \right] + \\ & ExtentPosition_Y \cdot \left(1 + \frac{\mathrm{NExtents_X} - 1}{\mathrm{NDisks}} \right) + \frac{ExtentPosition_X}{\mathrm{NDisks}} \end{aligned} \quad (2)$$

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 1).



Extraction slice specification | Extraction of the digital slice from the 3D image | Extracted slice parts | Resampled slice parts merged into the final displayable slice

FIGURE 1. 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.

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 2).

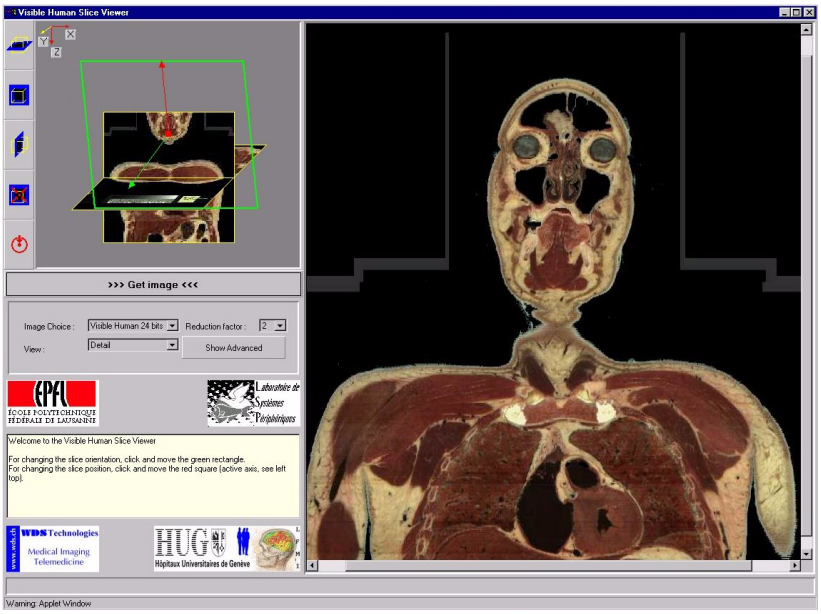


FIGURE 2. Selecting an image slice within a 3D tomographic image

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 resampling requests. These servers execute the requests and transfer the resulting slice parts to the server proxy residing on the client site, which asssembles them into the final displayable image slice (Figure 3).

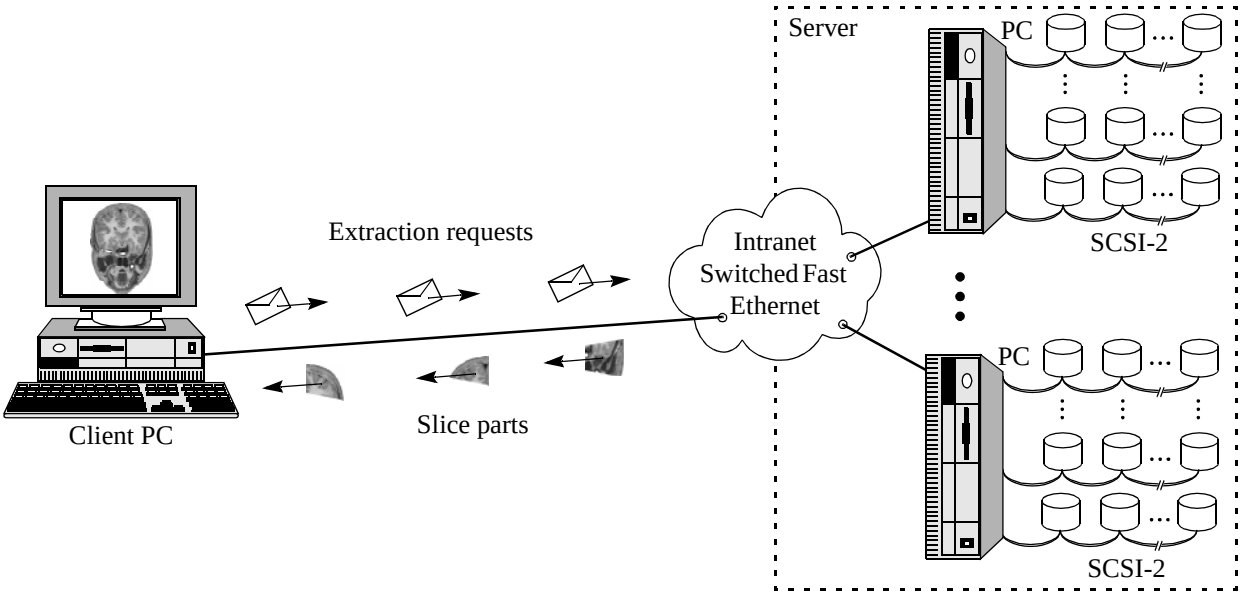


FIGURE 3. Sending the extraction requests and receiving the slice parts

3 Computer-aided parallelization of application programs

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 processors 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 preprocessor automatically generates parallel code that implements the required synchronizations and communications to satisfy the data dependencies specified by the user. CAP also handles a large part of the 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, transferred across the network, and unpacked. Communication occurs only when the output token of an operation is transferred to the input of another operation.

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 *directed* 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++ 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.

Each parallel CAP construct consists of a split function 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 its execution and passing its result to the higher level program after the arrival of all sub-results (Figure 4).

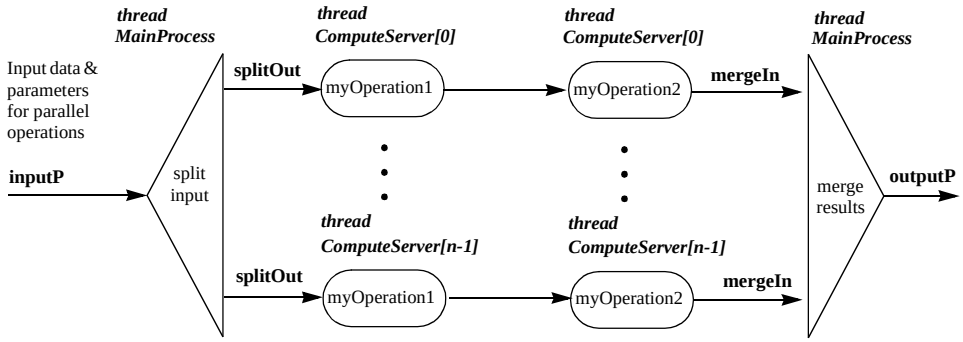


FIGURE 4. Graphical CAP specification: parallel operations are displayed as parallel horizontal branches, pipelined operations are operations located in the same horizontal branch

Parallel constructs are defined as high level operations and can be included in other higher level operations to ensure compositionality. The *parallel while* construct corresponding to the example of figure 4 has the syntax shown in Figure 5. Besides *split* and and *merge* functions, it incorporates two successive pipelined *operations* (*myOperation1* followed *by myOperation2*).

```
operation   ParallelServerT::ParallelComputation
 in inputParTokenType* inputP
  out mergeOutputTokenType* outputP
{
  parallel while (splitInput, mergeResults, MainProcess, mergeOutputTokenType Result)
    (ComputeServer[thisTokenP->index].myOperation1
) ;  >->ComputeServer[thisTokenP->index].myOperation2
    )
}
```

split function
merge function
thread executing merge function
result and its type
output of myOperation1 is directed to myOperation2

FIGURE 5. Syntax of a parallel while construct

The *split* function is called repeatedly to split the input data into subparts which are distributed to the different compute server thread operations (*ComputeServer[i].myOperation1*). Each operation running in a different thread (*ComputeServer[i]*) receives as input the subpart sent by the split function or by the previous operation, processes this subpart and returns its subresult either to the next operation or to the merge function. The parallel construct specifies explicitely in which thread the merge function is executed (often in the same thread as the split function). It receives a number of subresults equal to the number of subparts sent by the split function. Split and merge functions are

executed as many times as specified in the split function. For further information on the CAP preprocessor language syntax, please consult the CAP tutorial [CAP98].

A CAP program may incorporate any number of threads. The number of contributing threads is defined at compile time and chosen as large as required to run the parallel application on the largest available parallel platform. To overlap I/O and computations there is generally at least one thread for computing and one thread for disk I/O per processor. Load balancing at execution time is achieved by directing tokens to operations having the "smallest" token queue[1].

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 [Gennart96]. 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].

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 threads and to ensure their synchronizations. 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 correct pipelining is achieved, i.e. that data is transferred through the network or read from disks while previous data is being processed.

Furthermore, predefined library operations are available, for example for parallel file storage and access operations. Combining parallel disk access operations (in Figure 6 reading extents from disks) and processing operations as desired enables building highly pipelined applications incorporating the required parallel file system components [Messerli97].

4 Parallelizing the image slice extraction and visualization application using the CAP Computer-Aided Parallelization tool

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 PCs.

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 decomposing the application into basic independent sub-tasks that may be executed in a pipelined-parallel manner.

From section 2, 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 resample it on the display grid (possibly combined with zoom),
- merge the extracted and resampled 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. 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). 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 extacted and resampled *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 server PC where this extent resides[2]. Once an extent is read by the asynchronous *ReadExtent* operation, it is fed to the *ExtractAndResampleSlicePart* operation performed by a *ComputerServer[]* thread residing in the same computer as the *ExtentServer* thread. The resulting extracted and resampled slice part is sent back to the *Client* thread

1. Flow control and load balancing are highly coupled. The programmer may specify that a new token is to be generated by the split function and forwarded to a given operation once this operation has consumed one of the tokens of its token queue.

2. 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 computing the index of the processing node whose disk contains the desired volumic file extent.

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.

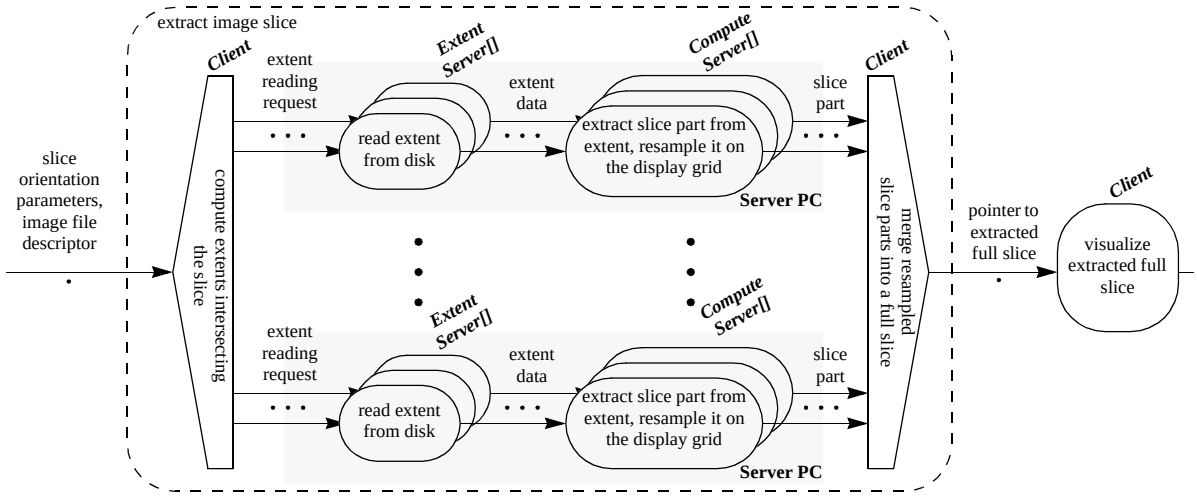


FIGURE 6. Graphical representation of the pipelined-parallel slice extraction and visualization operation

•

The CAP program of the pipelined-parallel slice extraction and visualization operation corresponding to the graph of Figure 6 is shown in Figure 7.

```
1  external leaf operation Ps2ExtentServerT::ReadExtent
2    in ExtentReadingRequestT* InputP
3    out ExtentT* OutputP;
4
5  leaf operation Ps2ComputeServerT::ExtractAndResampleSlicePart
6    in ExtentT* InputP
7    out SlicePartT* OutputP
8  {
9    OutputP = new SlicePartT;
10   ...C++ sequential code
11 }
12
13 bool SplitSliceRequest(SliceExtractionRequestT* FromP,
14                         ExtentReadingRequestT* PreviousP, ExtentReadingRequestT* &ThisP) {
15   ThisP = new ExtentReadingRequestT;
16   ...C++ sequential code
17   return (IsNotLastExtentReadingRequest);
18 }
19
20 void MergeSlicePart(SliceT* IntoP, SlicePartT* ThisP) {
21   ...C++ sequential code
22 }
23
24 operation Ps2ServerT::ExtractSlice
25   in SliceExtractionRequestT* InputP
26   out SliceT* OutputP
27 {
28   parallel while (SplitSliceRequest, MergeSlicePart, Client, SliceT Output)
29   (
30     ExtentServer[thisTokenP->ExtentServerIndex].ReadExtent
31     ->
32     ComputeServer[thisTokenP->ComputeServerIndex].ExtractAndResampleSlicePart
33   );
34 }
35
36 leaf operation ClientT::VisualizeSlice
37   in SliceT* InputP
38   out capTokenT* OutputP
39 {
40   OutputP = new capTokenT;
41   ...C++ sequential code
42 }
43
44 operation Ps2ServerT::ExtractAndVisualizeSlice
45   in SliceExtractionRequestT* InputP
46   out capTokenT* OutputP
47 {
48   Ps2Server.ExtractSlice
49   ->
50   Client.VisualizeSlice;
51 }
```

thread where *MergeSlicePart* is executed

split function

merge function

output token of the parallel while construct

FIGURE 7. CAP specification of the pipelined-parallel slice extraction and visualization operation

Pipelining is achieved at three levels:

- slice extraction and resampling is performed by the compute server thread on one extent while the extent server thread reads the next extents,
- an extracted and resampled slice part is merged by the client thread into the full image slice while the next slice parts are being extracted, 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 resampling operations are done in parallel by several processors; the number of processors can be increased to improve the slice part extraction and resampling performance.

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 *ExtractAndResampleSlicePart* (line 5) extract the slice parts from the received extents, resample them in the display grid, and return them to the *Client* thread who originally started the operation. The *Client* thread merges the resampled slice parts into a single *SliceT* full slice using the *MergeSlicePart* routine (line 20).

The sequential *VisualizeSlice* operation (line 36) is a standard 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 (Figure 8).

```
1  configuration {
2  processes:
3    A ( "User" ) ;
4    B ( "128.178.75.65", "\\FileServer\SharedFiles\SlicePartExtraction.exe" );
5    C ( "128.178.75.66", "\\FileServer\SharedFiles\SlicePartExtraction.exe" );
6    D ( "128.178.75.67", "\\FileServer\SharedFiles\SlicePartExtraction.exe" );
7  threads:
8    "Client" (A) ;
9    "ComputeServer[0]" (B);
10   "ComputeServer[1]" (B);
11   "ComputeServer[2]" (C);
12   "ComputeServer[3]" (C);
13   "ComputeServer[4]" (D);
14   "ComputeServer[5]" (D);
15   "ExtentServer[0]" (B);
16   "ExtentServer[1]" (C);
17   "ExtentServer[2]" (D);
18 };
```

FIGURE 8. Configuration map specifying the layout of ExtentServer and ComputeServer threads onto different PC's

Changing a configuration map enables the same program to run without recompilation on different hardware configurations. CAP programs are debugged by first running the parallel program as a multi-threaded single process program on a single computer. On Windows NT, they are debugged using the standard Visual C++ debugger. Once a program executes correctly within a single adress space, changing the configuration file enables mapping the threads to different processes (different address spaces) on the same computer and continuing debugging using several instances of the Visual C++ debugger. In the final phase, the final configuration is established and the threads may run in processes located on different computers.

The program shown in Figure 7 features a major drawback. Numerous extent reading requests have to be sent from the client PC to the server PCs creating an important load on both the client and server network interfaces and processors. To give a figure, with 5 server PCs, each with one disk and an enabled disk cache, 4.8 512x512 image slices per second are visualized. Since each slice perpendicular to the volume's main diagonal intersects in the average 437 volumic extents, extracting a slice requires in 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 server PCs consists in sending a full slice access request to all the server PCs. The server PCs segment themselves the slice access request into local extent reading requests. Thanks to CAP, a minor modification of the program shown in Figure 7 enables to generate the new optimized application (Figures 9 and 10).

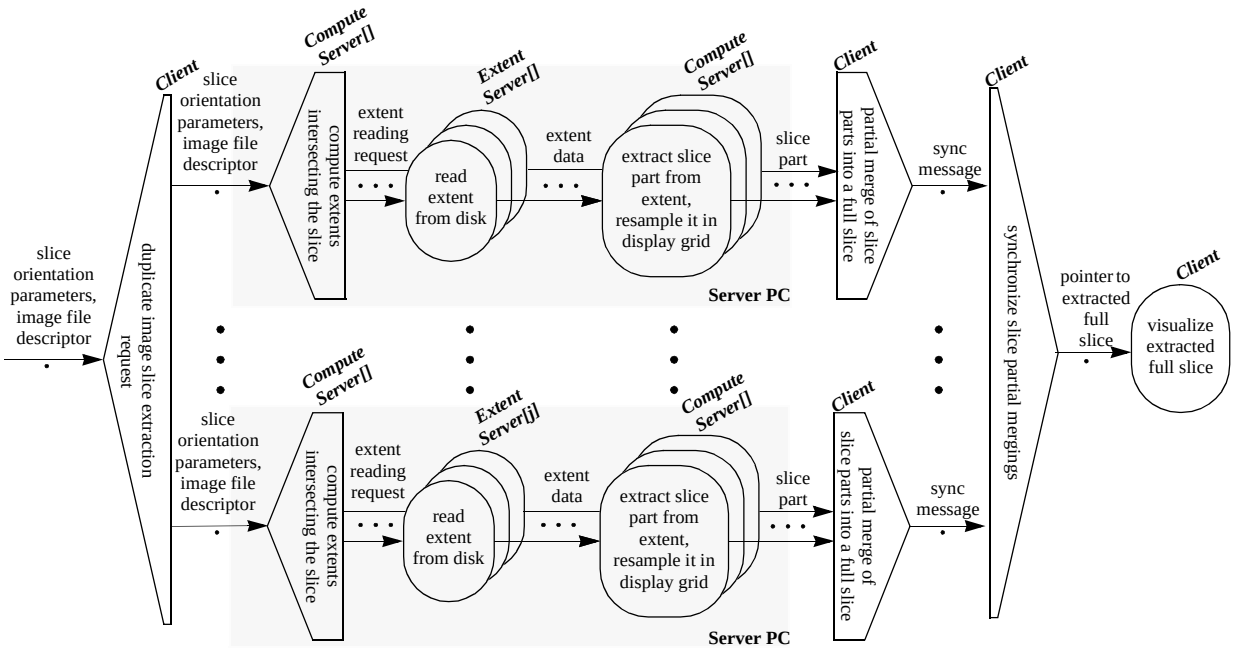


FIGURE 9. Graphical representation of the improved operation requiring much less communication between the client PC and the server PCs

```
1  void DuplicateSliceExtractionRequest(SliceExtractionRequestT* FromP,
2                                       SliceExtractionRequestT* &ThisP,
3                                       int ServerPC) {
4    ThisP = new SliceExtractionRequesT(FromP);
5  }
6
7  void SynchronizeServerPCs(SliceT* IntoP, SliceT* ThisP) { ...C++ sequential code }
8
9  operation Ps2ServerT::ExtractSlice
10   in SliceExtractionRequestT* InputP
11   out SliceT* OutputP
12 {
13   indexed (int ServerPC = 0; ServerPC < NSERVERPCS; ServerPC++)
14   parallel (DuplicateSliceExtractionRequest, SynchronizeServerPCs, Client, SliceT Output)
15   (
16     ComputeServer[ThisTokenP->ComputeServerIndex].{ }
17     ->
18     parallel
19     while (SplitSliceRequest, MergeSlicePart, Client, SliceT Output)
20     (
21       ExtentServer[ThisTokenP->ExtentServerIndex].ReadExtent
22       ->
23       ComputeServer[ThisTokenP->ComputeServerIndex].ExtractAndResampleSlicePart
24     )
25   );
26 }
```

FIGURE 10. Improved CAP specification requiring much less communication between the client PC and the server PCs

Comparing the first version in Figure 7 and the improved specification in Figure 10, only 5 lines (line 13 through line 17) have been added to the pipelined-parallel *ExtractSlice* operation. For each server PC (line 13), the *SliceExtractionRequestT* request is duplicated by the *DuplicateSliceExtractionRequest* routine (line 1) and sent to a *ComputerServer* thread located on that particular server PC (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 7). Line 16 forces the *SliceExtractionRequestT* request to be sent from the client PC to a server PC where the *SplitSliceRequest* routine will be executed.

By being able to direct at execution time the *ReadExtent* and *ExtractAndResampleSlicePart* operations to the storage server PC where the extents reside, 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 Equation 1. We do not use collective I/O [Reed95] since there are no disk accesses to small pieces of data. On each disk access, at least 1 extent (51 KBytes) is read.

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 expressing 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.

5 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 3). Each server PC 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 server PC's, one client 333MHz Bi-PentiumII PC located on the network runs the 3D tomographic image visualization task (Figure 2) 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 server PC's, 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 9).

A TCP/IP socket-based communication library called MPS implements the asynchronous *SendMessage* and *ReceiveMessage* primitives enabling CAP generated messages, i.e. tokens, to be sent from the application program memory space of one PC to the application program memory space of a second PC, with at most one intermediate memory to memory copy at the receiving site. In order to reduce the overhead of sending and receiving many small TCP/IP packets, the *SendMessage* primitive is able to coalesce several tokens into the same TCP/IP packet. This greatly improves the network throughput and reduces the processor utilization at both sides of the connection.

The present application incorporates several potential bottlenecks: insufficient parallel disk I/O bandwidth, insufficient parallel server processing power for slice part extraction and resampling, insufficient network bandwidth for transferring the slice parts from server PC's 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 the diagonals traversing the Visible Human's rectilinear volume.

5.1 Zoom factor 1, disk 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 server PC, 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. .

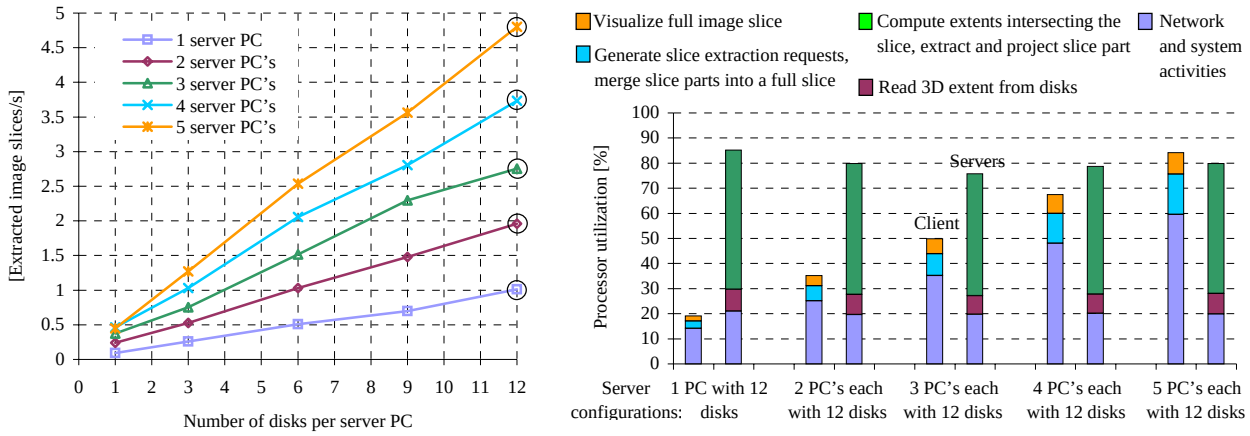


FIGURE 11. Performances at zoom factor 1, without disk caching

Figure 11 shows the performances obtained, in number of image slices per second, as a function of the number of contributing server PC's and a function of number of disks per contributing server PC. In order to approximate the worst case behavior, i.e. the general case where successive requests are directed towards completely different extents, no disk caching is allowed.

When disabling the disk caches, with up to 12 disks per server PC, disk I/O bandwidth is always the bottleneck. Therefore increasing either the number of disks per server PC or the number of server PCs (assuming each PC incorporates an equal number of disks) increases the number of disks and offers a higher extracted image slice throughput (Figure 11). With the 5 server PC 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 indicates that the disks are the bottleneck since, when measured separately, they exhibit an effective I/O throughput of 1.88 MBytes/s.

Each time you add a group of 3 disks to a single server PC 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 server PC can handle is 15 disks per server PC 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 server PCs each with 15 disks and the extracted image slice throughput is (75/3) x 0.24[slices/s] = 6 image slices/s. From 5 server PCs 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.

5.2 Zoom factor 1, disk 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 server PCs disk caches, the bottleneck shifts quickly from the disks to the limited processing power available on the server PC's (Figure 12). This shows the efficiency of the disk caches when browsing through the Visible Human.

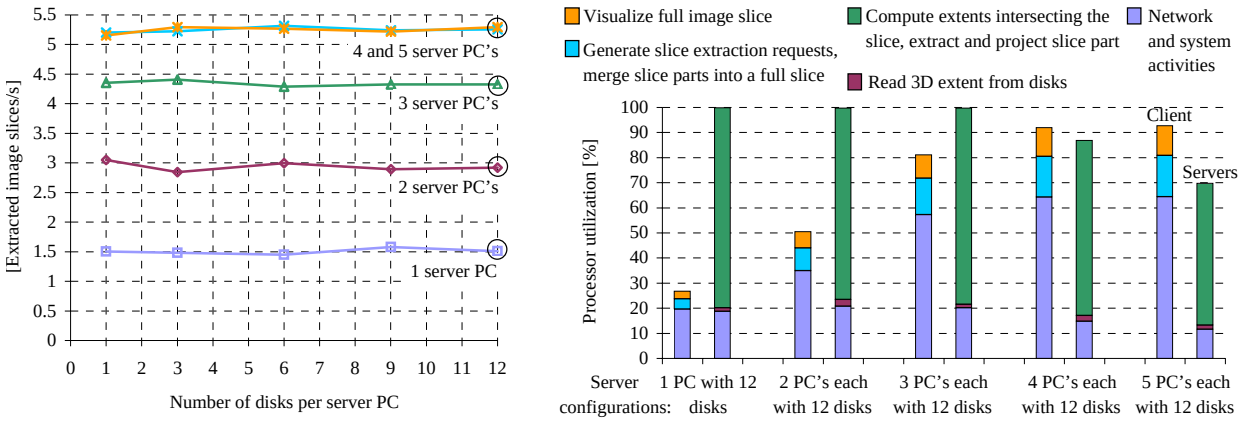


FIGURE 12. Performances at zoom factor 1, with a disk cache of 25 MBytes per server PC

Figure 12 shows that with up to 3 server PCs the slice part extraction and resampling operation running on the server PCs is the bottleneck. Since the disk cache comprises 25 MBytes per server PC and the amount of extents that must be read for extracting a single 512x512 image slice is 22 MBytes, the maximum disk cache efficiency, i.e. the case when each extent is read only once from a disk, is already reached with a single server PC. At this maximum cache hit rate, 97.9% of the accessed extents are read from the disk 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 server PC with a single disk and an enabled disk cache is able to extract and resample 1.5 image slices/s. Without the disk 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 caches.

The next bottleneck is not the client processor but resides in the limited bandwidth of its Fast Ethernet network interface (Figure 12, 4 and 5 server PCs), 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.

5.3 Zoom factor 2, disk 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 13 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.

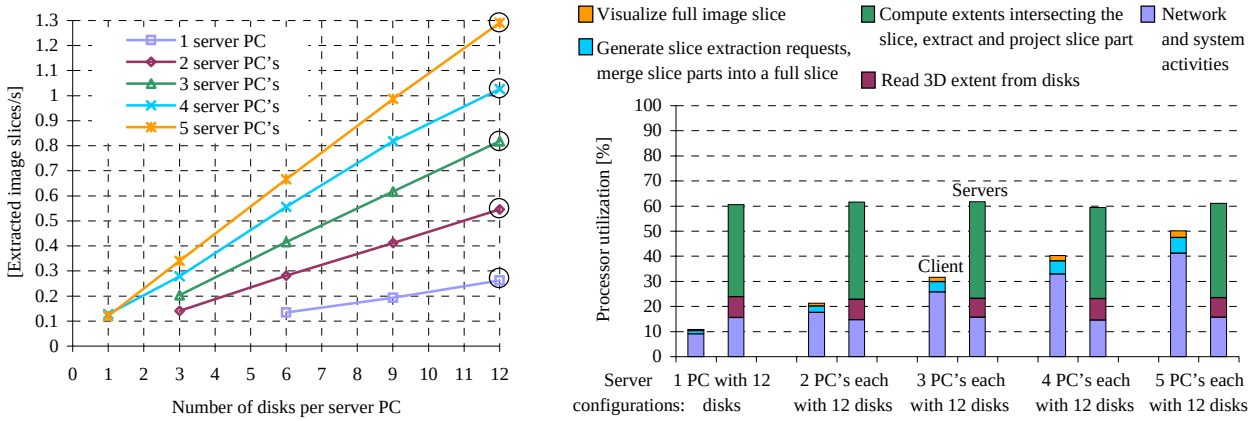


FIGURE 13. Performances at zoom factor 2, without disk caching

As expected, the aggregate disk I/O throughputs are the same as with zoom factor 1 (Figure 11). For example, with the 5 server PC 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 server PC 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 11), 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 resampled 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.

Extrapolation from the measured performance figures shows that the maximum number of disks that a single server can handle is 18 disks. In that case, 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 server PCs 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 server PCs, 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.

5.4 Zoom factor 2, disk cache enabled

As with zoom factor 1 (Figure 12), when enabling the disk caches the bottleneck shifts quickly from the disks to the limited processing power available on the server PCs (Figure 14).The disk caches become effective only from 2 server PCs. With a single server PC 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 disk caches. A single server PC with an enabled disk cache sustains a maximum throughput of 3[image slices/s] / 5[server PCs] = 0.6 image slices/s. With a disabled disk cache, it sustains only 0.39 image slices/s.

The efficiency of the disk 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 disk cache is available from 4 server PCs (each with 25 MBytes disk cache). 97.9% of accessed extents are read from the disk caches (as expected, same hit rate as zoom factor 1). With 5 server PCs a global I/O throughput of up to 3[slices/s] x 80[MBytes/slice] = 240 MBytes/s is obtained.

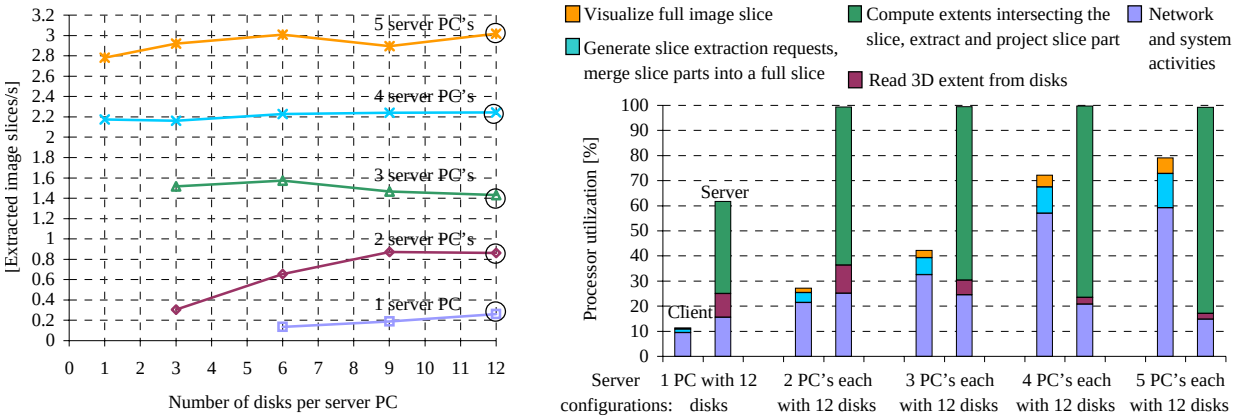


FIGURE 14. Performances at zoom factor 2, with a disk cache of 25 MBytes per server PC

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 disk caches are enabled (Figure 14, 79[%] / 3[slices/s] = 26% for a rate of 1 slice/s) than disabled (Figure 13, 50% / 1.3[slices/s] = 38% for a rate of 1 slice/s). This explains why with the disk 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 *SendMessage* 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. The efficiency of this technique depends on the output token rate which is higher when disk caches are enabled. Without this clever technique, the client processor would have rapidely become the bottleneck.

6 Conclusions

Image processing server applications requiring high-performance and I/O intensive operations are generally implemented by relatively expensive shared memory multiprocessing systems (for example the SGI Origin 2000). Considerably cheaper servers may be created by using cheap distributed memory commodity computers such as PC's and standard interconnects (FastEthernet). However, without adequate tools, programming parallel applications running on distributed memory computers is a tedious and time-consuming task.

We propose the CAP computer-aided parallellization tool to simplify the creation of pipelined parallel distributed memory server applications. Application programmers create separatly the serial program parts and express the parallel behavior of the program with CAP constructs. The predefined CAP parallel structures ensure that the resulting parallel program executes as an acyclic directed graph and is therefore deadlock free. Furthermore, due to its macro data flow nature, it generates highly pipelined applications where communication and input/output operations run in parallel with processing operations[3].

A CAP program can be easily modified by changing the sequence of operations or by building hierarchical CAP constructs. It facilitates the maintenance of parallel programs by enforcing a clean separation between the serial and the parallel program parts.

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 *SendMessage* and *ReceiveMessage* routines.

This contribution has demonstrated the use of CAP on a real application, the Visual Human Slice Server. The measured performances are close to the performances offered by the underlying hardware, operating system (Windows NT) and network protocols (TCP/IP).

The parallel server architecture has been interfaced to a Web server using the ISAPI protocol [Genusa97]. A Java 1.1 applet runs on Web clients and enables users to specify slice position and orientation and generate access requests.

3. Communications are only partially hidden, since the TCP-IP communication protocols require considerable processing power.

Replies of the server are compressed using the JPEG standard and send back to the Web clients for display. The Web interface is operational at *http://visiblehuman.epfl.ch.*

A number of other applications have been successfully developed using CAP [Gennart96], [Mazzariol97], [Gennart98]. We expect CAP to offer a high potential for developing powerful parallel digital imaging servers.

References

[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

[CAP98] Program Parallelization with CAP, LSP-EPFL, see http://diwww.epfl.ch/w3lsp/pub/publications/gigaview/captutorial.pdf

[Gennart96] Benoit A. Gennart, Joaquin Tarraga, Roger D. Hersch, "Computer-Assisted Generation of PVM/C++ Programs using CAP," *Proceedings of EuroPVM'96*, LNCS 1156, Springer Verlag, Munich, Germany, October 1996, 259-269

[Gennart98] 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*, San Jose, California, USA, January 1998, 446-458

[Genusa97] Stephen Genusa, *Special Edition Using ISAPI*, Que Corporation, 1997

[Grimshaw93a] Andrew S. Grimshaw, "Easy-to-use Object-Oriented Parallel Processing with Mentat," *IEEE Computer*, Vol. 26, No. 5, May 1993, 39-51

[Grimshaw93b] Andrew S. Grimshaw, "Dynamic, Object-Oriented Parallel Processing," *IEEE Parallel & Distributed Technology*, Vol. 1, No. 2, May 1993, 33-47

[Gropp94] W. Gropp, E. Lusk, A. Skjellum, Using MPI, The MIT Press, 1994

[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, Krakow, Poland, November 1997, 351-358

[Messerli97] Vincent Messerli, Benoit A. Gennart, Roger D. Hersch, "Performances of the PS^2 Parallel Storage and Processing System," *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, IEEE Computer Society Press, Seoul, Korea, December 1997, 514-522

[MPI94] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *The International Journal of Supercomputer Applications and High Performance Computing*, 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

[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

[Reed95] D. Reed, C. Catlett, A. Choudhary, D. Kotz, M. Snir, "Parallel I/O: Getting Ready for Prime Time," *IEEE Parallel & Distributed Technology*, Summer 1995, 64-71

[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, March/April 1996, 118-130

[Wilson96] Gregory V. Wilson, Paul Lu (Eds), *Parallel Programming Using C++*, The MIT Press, 1996