Computer-Aided Parallelization of Continuous Media Applications: the 4D Beating Heart Slice Server

J. Tárraga, V. Messerli, O. Figueiredo, B. Gennart, R.D. Hersch

Ecole Polytechnique Fédérale de Lausanne, EPFL
CH-1015 Lausanne, Switzerland
{jtarraga, messerli, figueiredo, gennart, hersch}@di.epfl.ch

Abstract

Parallel servers for I/O and compute intensive continuous media applications are difficult to develop. A server application comprises many threads located in different address spaces as well as files striped over multiple disks located on different computers. The present contribution describes the construction of a continuous media server, the 4D beating heart slice server, based on a computer-aided parallelization tool (CAP) and on a library of parallel file system components enabling the combination of pipelined parallel disk access and processing operations. Thanks to CAP, the presented archictecture is concisely described as a set of threads, operations located within the threads and flow of data and parameters (tokens) between operations. Continuous media applications are supported by allowing tokens to be suspended during a period of time specified by a user-defined function. Our target application, the 4D beating heart server supports the extraction of freely oriented slices from a 4D beating heart volume (one 3D volume per time sample). This server application requires both a high I/O throughput for accessing from disks the set of 4D sub-volumes (extents) intersecting the desired slices and a large amount of processing power to extract these slices and to resample them into the display grid. With a server configuration of 3 PCs and 24 disks, up to 7.3 slices can be delivered per second, i.e. 43 MB/s are continuously read from disks and 4.1 MB/s of slice parts are extracted, transfered to the client, merged, buffered and displayed. This performance is close to the maximal performance deliverable by the underlying hardware. The observed single stream server delay jitter varies between 0.6s (52% of maximal display rate) and 1.4s (92% of the maximal display rate). For the same resource utilization, the jitter is proportional to the number of streams that are accessed synchronously. The presented 4D beating heart application suggests that powerful continuous media server applications can be built on top of a set of simple PCs connected to SCSI disks.

Keywords

Parallel continuous media server, parallel I/O streaming, 4D tomographic images, resource reservation, disk scheduling, computer-aided parallelization.

***Proc. ACM Multimedia 1999*, (Orlando, Florida), 431-441**

1. Introduction

Since the early nineties, there has been considerable interest in developing concepts and computer architectures for continuous media applications, especially for video on demand services. Recent research has focussed either on the design of single computer video servers [9], [13] or on the design of large parallel video servers able to serve thousands of client viewers [1], [2], [8]. These servers generally incorporate dedicated admission control, file striping and file access mechanisms ensuring that the hard real-time video stream delivery constraints (guaranteed throughput, bounded jitter) can be met.

Continuous media services may not only consist of the transfer of data between the server's disks and clients but also of processing operations. The server might for example need to transcode the media stream from one compression standard to another, or to compute from an original video stream a derived stream with smaller size frames and a lower frame rate.

In the present paper, we focus our attention on the design and implementation of a parallel continuous media server application requiring intensive I/O access and processing operations: the *4D beating heart slice server*. A 4D tomographic beating heart incorporates as many 3D tomographic volumic images as time slices. In our experimental server, we store a beating heart made of 320 volumic images, corresponding to time slices of 1/16 s over a time interval of 20 seconds.

This server receives from clients slice stream access requests. Each slice stream request specifies the orientation of the slices within the 3D time-varying tomographic image and the access rate. After admission control (to allocate the required resources), the server continuously extracts slices according to the specified orientation and rate and transmits them to the client for visualization purposes.

To serve several continuous slice viewing requests simultaneously, the server needs both significant processing power and high disk throughput. To benefit from low-cost commodity components, the server architecture we consider consists of a cluster of PCs interconnected by a Fast Ethernet switch. Each PC is connected to several SCSI-2 disks (up to 12 disks).

Creating a parallel server application requires the explicit creation and management of processes, threads and

communication channels. In addition, accessing simultaneously and in parallel many disks 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 managing meta-information. To minimize communications, processing operations should be executed on processors which are close to the disks where the data resides.

In order to build parallel continuous media servers, we use a computer aided parallelization tool (CAP) and parallel file system components [10] that have been created for facilitating the development of parallel server applications running on distributed memory multi-processors, e.g. PCs connected by Fast Ethernet. We generate parallel server applications from a high-level description of threads, operations (sequential C++ functions) and of high-level parallel constructs specifying the flow of parameters and data 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.

Parallel volumic data access methods are detailed in section 2. The Beating Heart slice server application is described in section 3. Section 4 describes the computer-aided parallelization preprocessor (CAP) which is used to synthesize the parallel Beating Heart Slice Server application. Admission control, resource allocation, data streaming and synchronization issues are described in section 5. Performances and scalability issues are discussed in section 6.

2. The Parallel 3D Tomographic Slice Server

Before explaining how the 4D beating heart is stored and accessed, let us briefly present the principles underlying the storage and access of slices within a 3D tomographic volume declustered over several disks and PCs.

For enabling parallel storage and access, the volumic data set is segmented into 3D volumic extents of small size for example 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 neighbors reside on different disks hooked on different server PCs. We achieve such a distribution by storing successive extents on successive disks located in different PCs and by introducing an offset between two successive rows and between two successive planes of volumic extents [11]. This ensures that for nearly all extracted slices, disk and server PC accesses are close to uniformly distributed.

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 (Fig. 1).

Based on these principles, a parallel PC-based tomographic slice server was created using the CAP parallelization tool (see section 4) and parallel file system components [10]. This server works in a pipelined-parallel manner and combines high-performance computing and I/O intensive operations. It offers to any client the capability of interactively specifying the exact position and orientation of a desired slice and of requesting and obtaining that slice from the 3D tomographic volume. Performances scale close to linearly from one PC with 3 disks to 5 PCs with 60 disk [11]. A scaled down version of the server (1 Bi-Pentium II with 16 disks) offers its slicing services on the Web for accessing the *Visible Human* data set (http://visiblehuman.epfl.ch).

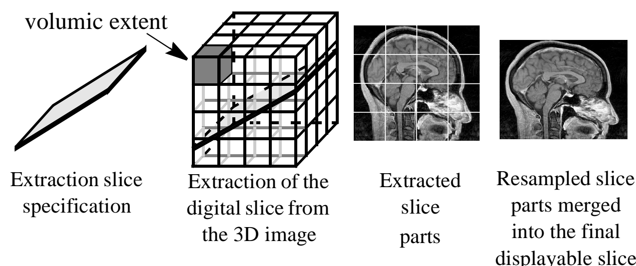

Fig. 1. Extraction of slice parts from volumic file extents

3. The 4D Beating Heart Slice Stream Server Application

Let us introduce the 4D beating heart slice stream server. The beating heart dataset consists of a sequence of 8-bits 3D volumic images, each one of size 512 x 512 x 512 (i.e. 128 MBytes). With 320 time instants, the 4D beating heart sequence reaches a size of 320 x 128 MBytes = 40 GBytes.

As shown in Fig. 2, each 3D volume of the beating heart is segmented into sub-volumes of size 16x16x16 voxels. A sequence of 16 successive sub-volumes is packed into one extent, i.e. into a sequence of bytes making up a stripe unit.

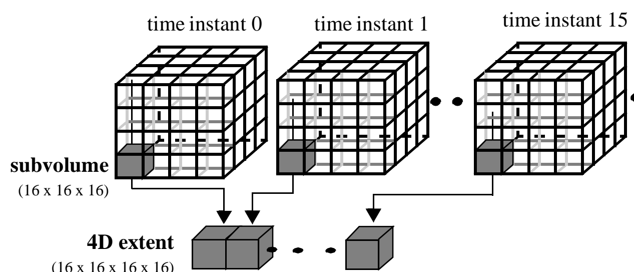

Fig. 2. 4D extent created from a sequence of 3D subvolumes

Creating extents which incorporate sub-volumes belonging to several consecutive time instants reduces the number of disk accesses, since the visualization of consecutive slices in

time requires sub-volumes associated to consecutive time instants. For a same extent size, the incorporation of several time instants reduces the extent's spatial volume. The smaller the spatial subvolumes, the less information is to be read from disks in order to extract a full slice. .

The client interface of the beating heart server (Fig. 3) enables users to specify the position and orientation of a 2D slice within the 3D volume. Then, the beating heart server executes extent accesses and slice extractions in order to create and send the desired slice stream (sequence of slices over time) at a user-specified rate to the client. A slice stream requires the extraction of the "same" slice from consecutive 3D image volumes. A slice is extracted from a 3D image by extracting sub-slices from subvolumes and merging them into a full slice as described in section 2 (Fig. 1)

The server application consists of a proxy residing on the client's site and of server processes running on the server's parallel processors. Once a slice stream access request is accepted (section 5.3), the proxy sends the stream parameters (slice orientation, slice position, display rate, stream duration) to the servers whose disks contain extents of the required stream. Each server generates at regular time intervals the requests for the slices making up the slice stream. From the generated slice request parameters, each server determines the extents which reside on its disks and accesses them. They extract the slice parts from the extents and send them to the client proxy, which assembles them into displayable slices. After having constructed a set of 16 slices, the proxy can start to display the slices at the specified rate while the next slice set is being prepared (Fig. 4).

Users may also specify (Fig. 3) different slice streams (i.e. streams with different slice positions and/or different slice orientations) and visualize them synchronously.

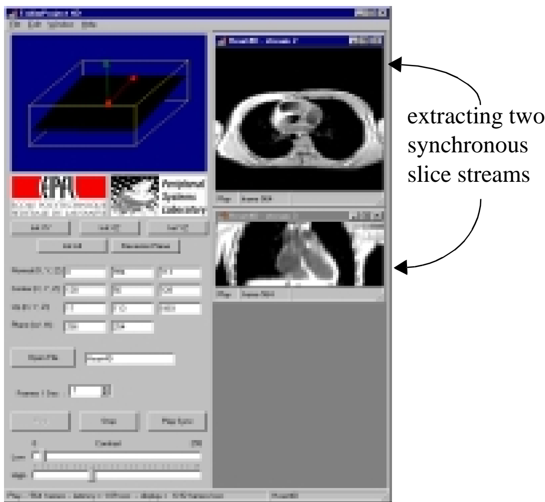

Fig. 3. Client's graphic user interface

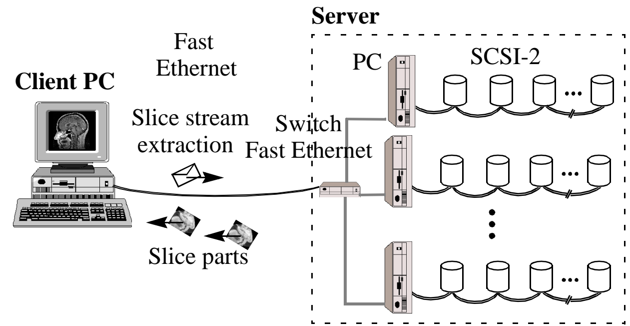

Fig. 4. Sending a slice stream extraction request and receiving the corresponding slice parts

4. 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 hierarchically specify the schedules of parallel operations and the flow of parameters and data (*macro dataflow*) between *operations*. Operations consist of sequential code performed by a single execution thread and characterized by input and output values. The input and output values of an operation are called *tokens*. In the context of this paper, tokens consist of image data (3D or 2D) and of additional application dependent parameters. 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 merge function collecting the results. The merge 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 (Fig. 5)

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 Fig. 5 has the syntax shown in Fig. 1. Besides *split* and *merge* functions, it incorporates two successive pipelined *operations* (*myOperation1* followed *by myOperation2*).

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 explicitly 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. For futher information on the CAP preprocessor language syntax, please consult the CAP

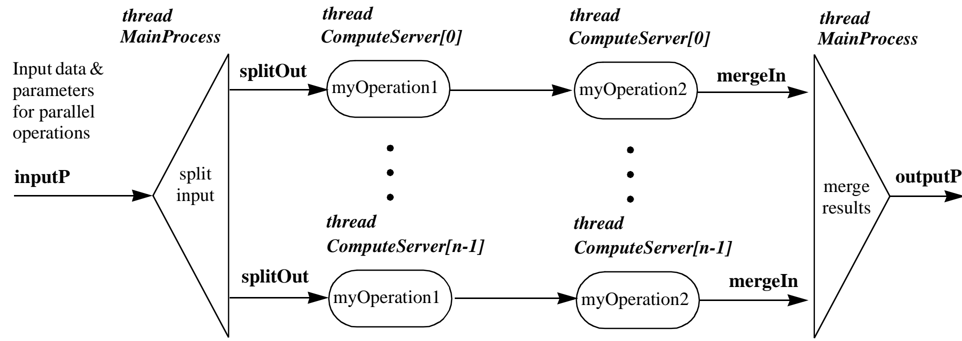

Fig. 5. Graphical CAP specification: parallel operations are displayed as parallel horizontal branches, pipelined operations are operations located in the same horizontal branch

tutorial [3]. 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 CAP threads onto the set of available processes and PCs. The macro data flow model which underlies the CAP approach has also been used successfully by the creators of the MENTAT parallel programming language [6], [7].

The CAP approach works at a higher abstraction level than the commonly used parallel programming systems based on message passing (for example MPI [14] and MPI-2 [12]). 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.

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.

CAP distinguishes itself from DCOM or CORBA by the fact that data and parameters circulate in an asynchronous manner between operations. CAP programs therefore make a much better utilization of the underlying hardware, specially for data streaming applications (continuous media servers).

5. The Parallel Continuous Media Server

This section describes the parallel continuous media server. Section 5.1 describes the parallel access to files striped across the available disks, possibly located on different PCs. Section 5.2 describes the threads making up the server and section 5.3 the admission control mechanisms. Section 5.4 describes the scheduling of disk requests. The global schedule for the pipelined parallel slice streaming sever is described in section 5.5.

5.1 Parallel Access to Striped Files

The parallel and continuous media server uses a library of parallel file system components [10] to stripe continuous media files over several local files located each one in a different disk. Each local file comprises a set of extents and a table whose entries specify the size and the storage location of extents, i.e. their byte address within the corresponding local file. When creating a continuous media parallel file, one must provide the number of local files, a set of path names, one for each local file, and a structure specifying the media format. When opening a continuous media parallel file, the system returns the parallel file descriptor, the number of subfiles, a table of disk server thread indices (one per local file) and a table of local file descriptors. These parameters allow the execution of extent oriented I/O operations on individual local files, such as *ReadExtent* or *WriteExtent* operations. The parallel continuous media server executes pipelined parallel data access and processing operations, where processing operations (e.g. slice part extractions from extents) are

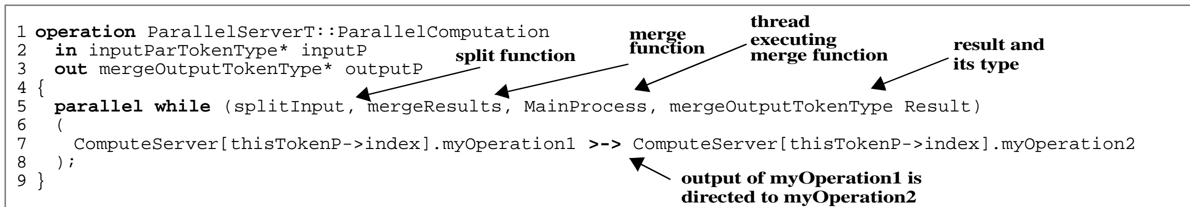

Fig. 1. Syntax of a parallel while construct

executed on the same nodes as the corresponding data access operations (i.e. extent disk accesses).

5.2 Parallel Continuous Media Server Threads

The parallel continuous media server (Fig. 7) consists of one *Client* thread and one *Display* thread running on the client PC, one *ResourceAllocator* thread running on one of the server PCs, and in each server PC of one *StreamTimer* thread, one *DiskScheduler* thread and one *ComputeServer* thread. The *ResourceAllocator* thread applies the admission control algorithm to evaluate the availability of server resources and to determine whether a new request stream can be accepted. In order to serve each accepted stream, the *ResourceAllocator* thread reserves the necessary resources. The *StreamTimer* threads are in charge of generating at regular time intervals the successive media access requests. Media requests (e.g. slice extraction requests in a slice stream) are timed, i.e. they must satisfy a certain deadline. The *ComputeServer* threads compute which extents contribute to the currently requested slice and perform computations on the extents extracted from the disks (e.g. extraction of slice parts). The *DiskScheduler* threads schedule the timed extent access requests before performing asynchronous extent read/write operations.

5.3 Admission Control Algorithm

Admission control is performed by the *ResourceAllocator* thread to ensure that a new stream request does not cause the violation of the real-time requirements of streams already being serviced. For each accepted stream, the admission control algorithm reserves the resources to serve them. The parallel server comprises the following resources (1) parallel disk I/O bandwidth, (2) parallel server processing power (for slice part extraction and resampling), (3) network bandwidth (for transferring the slice parts from server PCs to the client PC), and (4) 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). The implemented admission control algorithm reserves the disk bandwidth, the parallel server processor utilization and the network bandwidth. These resources are reserved up to a given bound (e.g. 2 MBytes/sec per disk). The determination of a pessimistic bound ensures that deadlines are met, but implies a poor utilization of the resource. An optimist bound implies a high resource utilization but may overload the resource. In the parallel server, the reservation bound is determined from worst-case experimental results since in the 3D beating heart application, no information loss is accepted.

The *ResourceAllocator* thread maintains the reserved server load (disk bandwidth, processor utilization, network bandwidth), i.e. the reserved capacity for each resource in the parallel server, and a reservation table whose entries specify the reserved resources for each of the streams being served. The *ResourceAllocatorT::ReserveResources* operation evaluates the server resources in order to accept or refuse the new stream. If the stream is accepted, the reserved server load is updated and an entry in the reservation table is created for the new stream. The *ResourceAllocatorT::FreeResources* operation is called by the *Client* thread when the stream retrieval is terminated. This operation deletes the reservation table entry specified by the stream descriptor of its input token, and updates the reserved server load.

5.4 Disk Scheduling

In order to reduce seek times, to achieve a disk high throughput and to guaranty the real-time requirements of the media streams being serviced, the *DiskScheduler* thread schedules the extent requests as follows. An extent request consists of the stream descriptor, the server node index, the local file descriptor, the local extent index and the request deadline. A zero deadline value specifies an access to a non-continuous media extent. The *DiskScheduler* thread maintains two extent request lists per disk hooked on its server PC, a list containing extent requests for non-

```
1 process StreamServerT {// high-level abstract thread
2 subprocesses:            // real threads
3   ClientProcessT Client;
4   DisplayT Display;
5   ResourceAllocatorT ResourceAllocator;
6   ServerNodeT ServerNode[NSERVERS];
7 operations: // high level parallel operations
8  OpenSliceStream in StreamNameT Input
9                  out StreamDescriptorT Output;
10 ExtractSliceStream in SliceStreamExtractionReqT Input
11                    out ErrorT Output;
12 StopSliceStream in StreamDescriptorT Input
13                 out ErrorT Output;
14 ...
15};
16
17 process ServerNodeT {// high-level abstract thread
18 subprocesses:           // real threads
19   StreamTimerT StreamTimer;
20   DiskSchedulerT DiskScheduler;
21   ComputeServerT ComputeServer;
22 };
23
24 process ResourceAllocatorT {
25 operations:
26  ReserveResources   in ResourceRequirementsT Input
27                     out ErrorT Output;
28  FreeResources  in StreamDescriptorT Input
29                 out ErrorT Output;
30 };
31
32
33 process StreamTimerT {
34 };
35
36 process ComputeServerT {
37 operations:
38   ExtractAndResampleSlicePart in ExtentDataT Input
39                               out SlicePartT Output;
40   ...
41 };
42
43 process DiskSchedulerT {
44 operations:
45   ScheduleReadExtent in ExtentReadReqT Input
46                      out ExtentDataT Output;
47   ScheduleWriteExtent  in ExtentWriteReqT Input
48                        out ErrorT Output;
49   ...
50};
```

Fig. 7. CAP specification of the parallel and continuous media server threads

continuous media and a list containing extent requests for continuous media. The continuous media request list has always a higher priority than the non-continuous media request list, i.e. continuous media extent requests are always served first. Non-continuous media extent requests are served according to a first come first served policy. Continuous media extent requests are served according to the earliest deadline first strategy. If several extent requests have the same deadline, they are served as follows:

- for extent requests belonging to different streams (e.g. streams of different media, or different slice streams), extent requests containing more time slices are served first[1],
- for extent requests belonging to the same stream (e.g. extents contributing to the same set of slices), extent requests are served in the order of their extent indices. This enables to move the disk head in one direction, similar to the SCAN disk scheduling algorithm [15], since extents with consecutive indices are generally stored in consecutive blocks.

5.5 Parallel Stream Retrieval and Presentation

In order to create a global schedule for parallel continuous media stream access let us identify the basic sub-tasks that compose the parallel retrieval and presentation of a slice stream:

- copy the slice access parameters to the contributing server PCs,
- generate the successive slice requests that form the slice stream at regular time intervals,
- compute the extents intersecting a slice,
- read an extent from a single disk,
- extract a slice part from an extent and resample it onto the display grid,
- merge an extracted and resampled slice parts into a full slice,
- visualize a full slice on the client computer.

Fig. 8 shows the macro data flow specifying the schedule of these seven basic operations. The input is a *SliceStreamExtractionReqT* token comprising the slice parameters (slice orientation and position), the stream descriptor, the display rate and the stream duration. First, the *Client* thread sends the *SliceStreamExtractionReqT* token to all the *StreamTimer* threads located on the server PCs contributing to the stream[2]. Using the system timers, each *StreamTimer* thread generates at regular time intervals the timed slice extraction requests to generate the slice stream. Each slice extraction request is defined by the *SliceExtractionReqT* token containing the slice parameters, the stream descriptor and a deadline. The time interval between slice extraction requests is derived from the display rate. Each *SliceExtractionReqT* request is directed to a *ComputeServer* thread running in the same PC. Timed extent reading requests are generated with a deadline derived from the *SliceExtractionReqT* deadline. There is one reading request per local extent intersecting the slice. The reading requests are sent to the *DiskScheduler* thread who applies the disk scheduling algorithm before reading the extents from disk. Once an extent is read, it is processed by the *ComputeServer* thread to extract and resample the corresponding slice part. The resulting extracted and resampled slice part is sent back to the *Client* thread and merged into a full slice. Once constructed and buffered, full slices are ready to be consumed at the display rate by the *Display* thread.

This global schedule achieves both pipelining and parallelism. Pipelining is achieved at three levels:

- slice part extraction is performed by the *ComputeServer* thread on one extent while the *DiskScheduler* thread reads the next extents,
- an extracted slice part is merged by the *Client* thread into the full slice while the next slice parts are being extracted,
- a full slice is displayed by the *Display* thread while the next full slices are being prepared.

Parallelization occurs at two levels:

- several extents are read simultaneously from different disks; the number of disks can be increased to improve the I/O throughput,
- extraction of slice parts from extents is done in parallel by several processors; the number of processors can be increased to improve the slice part extraction and resampling performance.

The schedule of Fig. 8 is implemented by the parallel *StreamServerT::ExtracSliceStream* CAP operation specified in Fig. *9*.

The input of the parallel *ExtractSliceStream* operation (lines 46 to 80) performed by the *Client* thread is a *SliceStreamExtractionReqT* request. Using an *indexed-parallel* CAP construct (line 50) this request is duplicated by the *DuplicateSliceStreamExtractionRequest* routine (line 1) and sent from the client PC to the *StreamTimer* threads (line 58) located on each server PC contributing to the stream extraction (lines 51 to 53). By using the *parallel-while-suspend* CAP construct (line 60), the *StreamTimer* thread calls the *GenerateSliceExtractionRequests* routine (line 9) to generate each time it is called a *SliceExtractionReqT* request with a certain deadline. Thanks to the suspension of the token flow during the time interval

[1] We assume that extent sizes are similar for different streams (between 40KB and 80KB)

[2] When a parallel stream is opened, the client thread obtains information about how the parallel stream is striped into subfiles and on which processing node each subfile resides. This enables computing the index of the processing node whose disk contains the desired file extent.

computed by the *CalculatePeriod* function (line 18), consecutive *GenerateSliceExtractionRequests* functions are called at appropriate time intervals. The number of generated slice extraction requests depends on the stream duration. Each *SliceExtractionReqT* request is directed to the *ComputeServer* thread (line 65) residing on the same PC as the *StreamTimer* thread (line 58). The *ComputeServer* thread performs the *parallel-while* CAP construct (line 67). There, the *SliceExtractionReqT* request is processed by the *GenerateExtentReadRequests* routine (line 21) that incrementally computes the local extents intersecting the slice. Each time it is called, it generates an *ExentReadReqT* request with the same deadline than the *SliceExtractionReqT* request deadline. The *ComputeServer* threads redirect the *ExtentReadReqT* requests to their *DiskScheduler* thread companions (line 71) who read the required extents from the disks by performing the *ScheduleReadExtent* operation. The extents read from disks are redirected back to the *ComputeServer* threads (line 74). The *ComputeServer* threads running the *ExtractAndResampleSlicePart* operation (line 32) extract the slice parts from the received extents, and return them to the *Client* thread who originally started the operation. The *Client* thread merges the slice parts into a full slice using the *MergeSlicePart* routine (line 29). The full slices are redirected to the *Display* thread (line 77) who performs the *BufferAndVisualizeSlice* operation (line 41). In this operation, a set of full slices is constructed before starting a system timer to display the full slices at the user-specified rate.

To extract multiple synchronous slice streams, we may simply launch at the same time several asynchronous calls to the *StreamServerT::ExtractSliceStream* CAP operation and specify for all of them the same display rate.

The *Client* thread requests an entire stream and the server PCs send stream data to the client at a controlled rate. This data delivery model, called server-push [8] (as opposed to the client-pull model), may cause a synchronization problem due to the parallel transmissions from multiple independently running server PCs having each one a different clock. To improve the synchronization, the *Client* thread divides the stream request into requests for stream pieces (e.g. a 10 minutes stream is divided into 10 pieces of 1 minute substreams).

The *Client* thread makes timed requests for the stream pieces and resynchronizes the server PCs at each stream piece request. This pattern combines the client-pull and server-push delivery models.

A configuration file maps the parallel and continuous media server threads onto the available processors of the hardware architecture (Fig. 10). Changing the configuration file enables the same program to run without recompilation on different hardware configurations. In this example, process A runs the user application on the user machine. Processes B and C execute the server application on machines whose IP addresses are 128.178.75.65 and 128.178.75.66 respectively. In the CAP program, there are 9 threads. The *Client* and *Display* threads run in process A. Threads

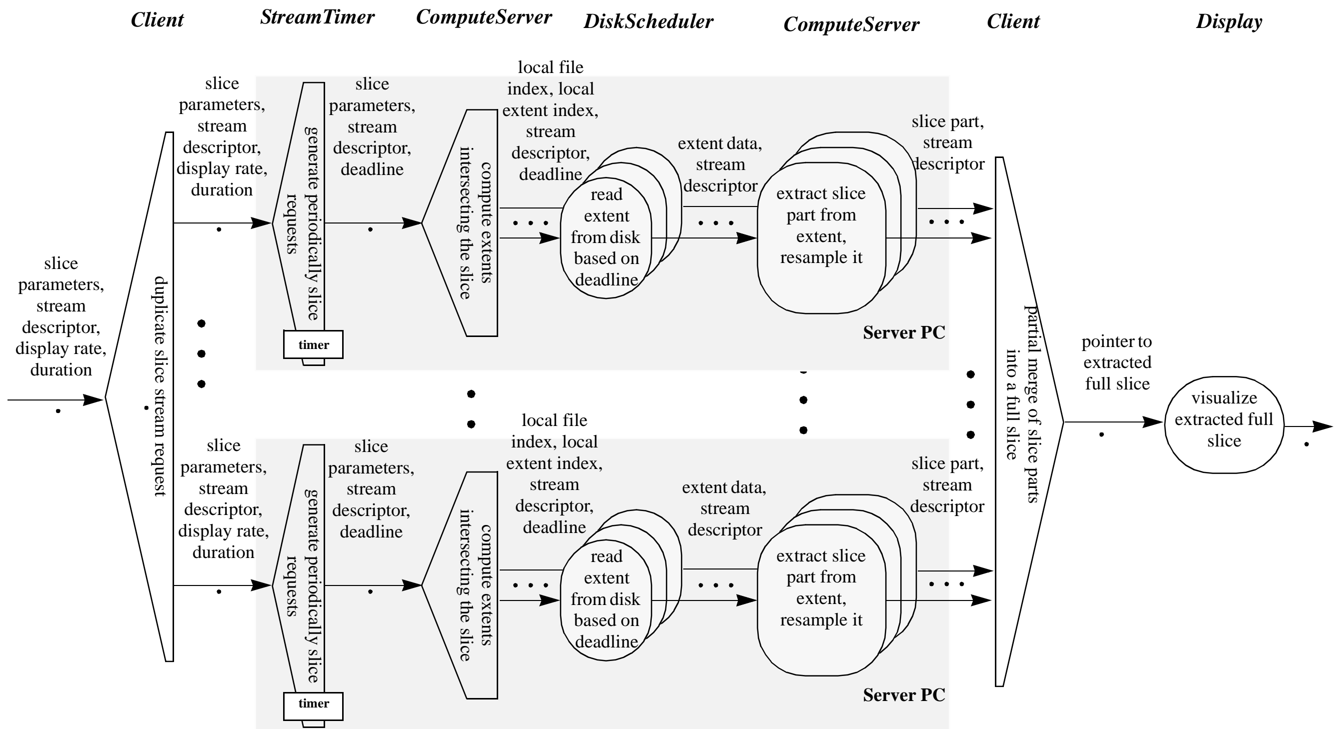

Fig. 8. Graphical representation of the pipelined-parallel slice stream extraction and visualization operation

ResourceAllocator, *ServerNode[0].StreamTimer, Server Node[0].ComputeServer* and *ServerNode[0].DiskScheduler* run in process B. Threads *ServerNode[1]. StreamTimer*, *ServerNode[1].ComputeServer* and *ServerNode[1].Disk Scheduler* run in process C.

Each *DiskScheduler* thread and its companion *ComputeServer* thread work in pipeline; multiple pairs of *DiskScheduler* and *ComputeServer* threads may work in parallel if the configuration map specifies that different *DiskScheduler*/*ComputeServer* threads are mapped onto different processes running on different computers (Fig. 10). In addition, by being able to direct at execution time the *ScheduleReadExtent* and *ExtractAndResampleSlicePart* operations to the storage server PC where the extents resides, operations are performed only on local data and superfluous data communications over the network are completely avoided. Load-balancing is ensured by appropriate distribution of extents onto the disks (section 3).

6. Performance and Scalability Analysis

The server architecture we consider comprises 4 200MHz Bi-PentiumPro PC's interconnected by a 100 Mbits/s Fast Ethernet crossbar switch (Fig. 4). 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 5400 rpm disks which have a measured mean physical data transfer throughput of 3.5 MBytes/s and a mean latency time, i.e. seek time + rotational latency time, of 12.2 ms [10]. Thus, when accessing 64 KBytes blocks, i.e. 16x16x16x16 8-bit extents, located at random disk locations, an effective throughput of 2.05 MBytes/s per disk is reached.

```
1 configuration {
2 processes:
3   A ("User") ;
4   B ("128.178.75.65", "\\Shared\StreamServer.exe");
5   C ("128.178.75.66", "\\Shared\StreamServer.exe");
6 threads:
7   "Client" (A) ;
8   "Display" (A) ;
9   "ResourceAllocator" (B);
10  "ServerNode[0].StreamTimer" (B);
11  "ServerNode[0].ComputeServer" (B);
12  "ServerNode[0].DiskScheduler" (B);
13  "ServerNode[1].StreamTimer" (C);
14  "ServerNode[1].ComputeServer" (C);
15  "ServerNode[1].DiskScheduler" (C);
16};
```

Fig. 10. Configuration file mapping the parallel and continuous media server threads onto different PCs

In addition to the server PCs, one client 200MHz Bi-PentiumPro PC located on the network runs the 3D beating heart visualization task which enables the user to specify interactively (Fig. 3) the desired slice stream access parameters (position, orientation and stream rate) and interacts with the server proxy to extract the desired slice stream. The server proxy running on the client sends the slice stream request to the server PCs, receives the slice parts and merges them into a set of displayable slices. The set of displayable slices is then passed to the 3D beating heart visualization task at the specified rate.

A TCP/IP socket-based communication library [17] called MPS implements the asynchronous *SendMessage* and

```
1 void DuplicateSliceStreamExtractionRequest
2  (SliceStreamExtractionReqT* FromP,
3   SliceStreamExtractionReqT* &ThisP,
4   int ServerNodeIndex) {
5  ThisP = new SliceStreamExtractionReqT
6                  (FromP, ServerNodeIndex);
7 }
8
9  bool GenerateSliceExtractionRequests
10   (SliceStreamExtractionReqT* FromP,
11    SliceExtractionReqT* PreviousP,
12    SliceExtractionReqT* &ThisP) {
13  ThisP = new SliceExtractionReqT;
14  //...C++ sequential code
15  return (IsNotLastSliceExtractionRequest);
16  }
17
18  int CalculatePeriod (SliceStreamExtractionReqT*InputP)
19  { //...C++ sequential code, return period in msec }
20
21  bool GenerateExtentReadRequests
22  (SliceExtractionReqT* FromP,
23   ExtentReadReqT* PreviousP, ExtentReadReqT* &ThisP ) {
24    ThisP = new ExtentReadReqT;
25    //...C++ sequential code
26    return ( IsNotLastExtentReadRequest );
27  }
28
29  void MergeSlicePart (SliceT* IntoP, SlicePartT* ThisP)
30  { //...C++ sequential code }
31
32  leaf operation
33  ComputeServerT::ExtractAndResampleSlicePart
34    in ExtentDataT* InputP
35    out SlicePartT* OutputP
36  {
37   OutputP = new SlicePartT;
38   //...C++ sequential code
39  }
40
41  leaf operation DisplayT::BufferAndVisualizeSlice
42    in SliceT* InputP
43    out ErrorT* OutputP
44    { //...C++ sequential code }
45
46  operation StreamServerT::ExtractSliceStream
47    in SliceStreamExtractionReqT* InputP
48    out ErrorT* OutputP
49  {
50    indexed (
51      int ServerNodeIndex = 0;
52      ServerNodeIndex < thisTokenP->ServerArray.Size ();
53      ServerNodeIndex++ )
54    parallel (DuplicateSliceStreamExtractionRequest,
55              void, Client, void Result2)
56    (
57      ServerNode[thisTokenP->ServerNodeIndex].
58      StreamTimer.{ }
59      >->
60      parallel while ( GenerateSliceExtractionRequests
61        suspend CalculatePeriod (thisTokenP),
62        void, Client, void Result3 )
63      (
64        ServerNode[thisTokenP->ServerNodeIndex].
65        ComputeServer.{ }
66        >->
67        parallel while ( GenerateExtentReadRequests,
68          MergeSlicePart, Client, void Output )
69        (
70          ServerNode[thisTokenP->ServerNodeIndex].
71          DiskScheduler.ScheduleReadExtent
72          >->
73          ServerNode[thisTokenP->ServerNodeIndex].
74          ComputeServer.ExtractAndResampleSlicePart
75        )
76        >->
77        Display.BufferAndVisualizeSlice
78      )
79    );
80  }
```

Fig. 9. CAP specification of the pipelined-parallel *StreamServerT::ExtractSliceStream* operation

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.

The present application comprises 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 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 slice stream extraction and visualization application performances, the experiment consists of requesting and displaying a 512x512 8-bit/pixel slice stream comprising 320 slices according to the server-push delivery model. In order to test the worst case behavior, the selected slice orientation is orthogonal to one of the diagonals traversing the Beating Heart's rectilinear volume.

In the experiment, 1 stream extraction request of 120 bytes is sent to each server PC. For each set of 16 consecutive slices, 1504 4D extents of size 64 KBytes (i.e. 16x16x16x16) are read in average from the disks (94 MBytes). The 16 consecutive slice parts (each one of size 390 Bytes approx.) contained in a 4D extent are extracted, packed and sent to the client, i.e 1504 messages of size 6.24 KBytes (i.e. 16 x 390 Bytes) are sent back to the client for each set of 16 512x512 image slices.

Fig. 11 (obtained from the Windows NT performance monitor) shows the load behavior over time of one server PC belonging to a 3 PC 24 disks server configuration. In Fig. 10a, one 512x512 8bit/pixel slice stream is extracted at 6 slices/sec and in Fig. 10b two synchronous 512x512 8bit/pixel slice streams are extracted at 3 slices/sec. Each cycle corresponds to the extraction of (a) one set of 16 slices and (b) two sets of 16 slices. It shows that while disk accesses are made, i.e. extents become available, the processor is 100% busy extracting slice parts.

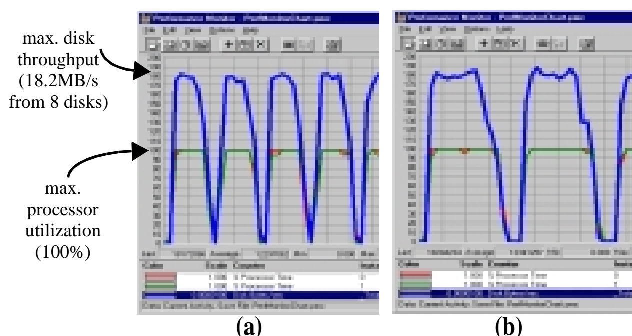

Fig. 10. Processor utilization and disk throughput of one server PC with 8 disks belonging to a 3 PC 24 disks server configuration, (a) when extracting one 512x512 8-bits slice stream at 6 slices/sec, and (b) when extracting two synchronous 512x512 8-bits slice streams at 3 slices/sec

Fig. 11 shows the performances obtained, in number of image slices per second, as a function of the number of contributing server PCs and a function of the number of disks per contributing server PC. With up to 7 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. From 8 disks per server PC, the bottleneck shifts from the disks to the limited processing power available on the server PCs. At 99% server processor utilization, 82% are dedicated for slice part extraction and resampling, 3% for extent reading and 14% for the network interface and system activities.

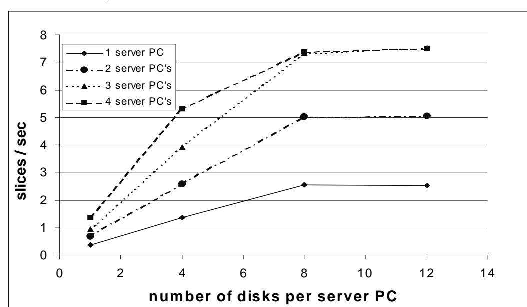

Fig. 11. Performances extracting a 512x512 8-bits slice stream

From 4 server PCs, each with 8 disks, the client PC is the bottleneck. 22% processor utilization is required for merging slice parts into a full slice and visualizing the full slices and 67% processor utilization is dedicated for the network interface and system activities. One of the two processors of the Bi-PentiumPro client PC is used at 95% for the network interface and system activities and is therefore the bottleneck.

For a single client, the optimal configuration consists of 3 server PCs and 24 disks. With such a configuration, up to 7.3 full slices/s can be generated. This corresponds to an aggregate disk throughput of roughly 43.24 MB/s (7.36/16 * 94 MB), i.e. 1.80 MB/s per disk. This aggregate disk throughput is slightly below the 2.05 MB/s measured mean throughput of individual disks due to the fact that with 8 disks per PC, the processor is the bottleneck.

To analyze the delay jitter of slice parts delivered by the server as a function of the server processor utilization, we consider a single slice stream request at a given nominal rate and two synchronous slice stream requests at half the nominal rate. Fig. 12 shows the delay distribution and its cumulative probability distributions (cpd) for server processor utilizations of 52% (corresponding to the extraction of one slice stream at the rate of 4 slices/sec) and 92% (7 slices/sec). For a 92% processor utilization, the jitter delay (1.4 sec) is slightly more than double the maximum

jitter delay (0.6 sec) at a 52% processor utilization.

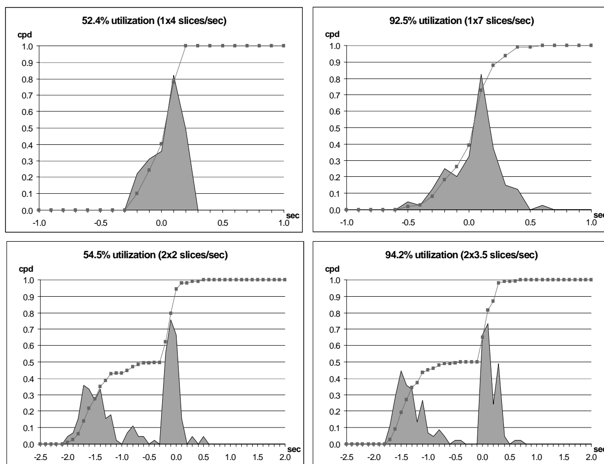

Fig. 12. Delay jitter for one single stream and for two synchronous streams

Fig. 12 shows the jitter delay when extracting two synchronous slice streams at 2 slices/sec (54.4% processor utilization) and 3.5 slices/sec (94.2% utilization). Due to the organization of 4D extents which incorporate data for 16 consecutive slices parts, accessing two synchronous streams requires reading from disks simultaneously the extents needed for two times 16 consecutive slices, i.e. 32 consecutive slices. At the same total display rate, in the case of two synchronous streams, double the number of disk accesses are made at each deadline. However, as Fig. 11b shows, the interval between deadlines is twice as large as in the equivalent single stream access application. This also explains why the delay jitter for two streams is approximately double the delay jitter for a single stream (Fig. 12). Slice parts belonging to 16 consecutive time instants are sent to clients. In double buffering mode, clients should therefore have the memory to store 32 slices per slice stream (8 MB per 512x512 8bit/pixel slice stream). Since the delay jitter is always less than the time to display 16 slices, delay jitter does not require additional buffer space.

7. Conclusions

Parallel distributed memory servers for I/O and compute intensive continuous media applications are difficult to develop. A server application comprises many threads located in different address spaces as well as files striped over multiple disks located on different computers. In order to ensure performances close to the potential offered by the underlying hardware and operating system, pipelined parallel programs need to be developed, which ensure that computations, disk accesses and network transfers occur simultaneously.

CAP greatly simplifies the creation of pipelined parallel continuous media applications. By construction, it generates completely pipelined parallel applications: in front of each I/O, each processing and each network transfer operation, a token queue ensures that as soon as the current operation returns, the next token is ready to be consumed. In addition, CAP enables application programmers to specify at a high level of abstraction the parallel application structure. The parallel application is concisely described as a set of threads, operations located within the threads and flow of data and parameters (tokens) between operations. Continuous media applications are supported by allowing tokens to be suspended during a period of time specified by a user-written function.

Our target application, the 4D beating heart server requires both a high I/O throughput for accessing from disks 4D extents intersecting the desired slices and a large amount of processing power to extract slices from 4D extents and resample them into the display grid. In order to obtain a slice set of 16 consecutive full slices, the server needs to read from the disks 1504 extents of size 64KB, i.e. 94 MB. From these extents, 1504 slice parts, each of 6.24 KB are extracted, resampled and merged at the client site (9.3 MB). With a server configuration of 3 Bi-Pentium Pro PCs and 24 disks (physical throughput: 3.5 MB/s, latency 12.2 ms), up to 7.3 slices can be delivered per second, i.e. 43 MB/s are continuously read from disks and 4.1 MB/s of slice parts are extracted, transferred to the client and merged. This performance is close to the maximal performance deliverable by the underlying hardware.

In the case of a single stream, and at a display rate of 52% of the maximal display rate, the delay jitter is 0.6 second. At 92% of the maximal display rate, the jitter grows to 1.4 s. For the same resource utilization, the jitter is proportional to the number of streams that are accessed synchronously. As long as the worst case delay jitter is smaller than the time to display a slice set, it does not require more memory than the memory for double buffering a full slice set (presently 16 slices).

The presented 4D beating heart application shows that thanks to CAP, computation- and I/O-intensive continuous media server applications can be built on top of a set of simple PCs connected to SCSI disks.

The creation of other continuous media applications may rely on the presented CAP parallel program skeleton. Data structures, individual operations and indices specifying thread locations would need to be appropriately modified.

The 4D beating heart application also suggests that tomographic equipment manufacturers may offer continuous 3D volume acquisition equipment by interfacing the acquisition device with a cluster of PCs, each PC being connected to several disks.

8. References

[1] Bolosky, W.J., et al. The Tiger Video Fileserver, in *Proc. 6th Int'l Workshop on Network and Operating System Support for Digital Audio and Video. IEEE Computer Society Press*, 1996, 97-104.

[2] Buddhikot, M.M. Design of a Large Scale Multimedia Storage Server. *Computer Networks and ISDN Systems*, Vol. 27, 1994, 503-517.

[3] Program Parallelization with CAP, LSP-EPFL, http://lspwww.epfl.ch/publications/gigaview/captutorial.pdf

[4] Gemmell, D.J., Vin, H.M., Kandlur, D.D., Rangan, P.V. Multimedia Storage Servers: A Tutorial. *IEEE Computer*, Vol. 28, No. 5, May 1995, 40-51.

[5] Gennart, B.A., Tárraga, J., Hersch, R.D. Computer-Assisted Generation of PVM/C++ Programs using CAP, in *Proc. of EuroPVM'96*, LNCS 1156, Springer Verlag, Munich, Germany, October 1996, 259-269.

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

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

[8] Lee, J.Y.B. Parallel Video Server: A Tutorial. *IEEE Multimedia*, Vol. 5, No. 2, April-June 1998, 20-28.

[9] Martin, C., Narayanan, P.S., Ozden, B., Rastogi, R., Silberschatz, A. The Fellini Multimedia Storage Server. *Multimedia Information Storage and Mangement*. Chung, S.M. (ed). Kluwer Academic Publishers, 1996, Chapter 5, 117-146.

[10] Messerli, V., Gennart, B.A., Hersch, R.D. Performances of the PS^2 Parallel Storage and Processing System, in *Proc. of the 1997 International Conference on Parallel and Distributed Systems,* Seoul, Korea, December 1997, *IEEE Press*, 514-522.

[11] Messerli, V., Figueiredo, O., Gennart, B., Hersch, R.D. Parallelizing I/O intensive Image Access and Processing Applications. *IEEE Concurrency*, accepted for publication, see also http://visiblehuman.epfl.ch.

[12] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, July 1997, http://www.mpi-forum.org

[13] Ozden, B., Rastogi, R., Silberschatz, A.. A Framework for the Storage and Retrieval of Continuous Media Data, in *Proc. of the IEEE International Conference on Multimedia Computing Systems*, Washington, 1995, IEEE Press, 2-13.

[14] Pacheco, P.S. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[15] Reddy, A.L.N., Wyllie, J.C. Disk Scheduling in a Multimedia I/O System, in *Proc. of the first ACM International Conference on Multimedia*, Anheim, 1993, 225-233.

[16] Reddy, A.L.N., Wyllie, J.C. I/O Issues in a Multimedia System, *IEEE Computer*, Vol. 27, No. 3, March 1994, 69-74.

[17] Messerli, V. *Tools for parallel I/O and compute intensive applications*. Ph.D. thesis 1915, Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, February 1999., http://lspwww.epfl.ch/publications/ps2/thesis-messerli