Computer-Aided Synthesis of Parallel Image Processing Applications

Benoit Gennart, Roger D. Hersch[1]
Ecole Polytechnique Fédérale, Lausanne, Switzerland

ABSTRACT

We present a tutorial description of the CAP Computer-Aided Parallelization tool. CAP has been designed with the goal of letting the parallel application programmer having the complete control about how his application is parallelized, and at the same time freeing him from the burden of managing explicitly a large number of threads and associated synchronization and communication primitives. The CAP tool, a precompiler generating C++ source code, enables application programmers to specify at a high level of abstraction the set of threads present in the application, the processing operations offered by these threads, and the parallel constructs specifying the flow of data and parameters between operations. A configuration map specifies the mapping between CAP threads and operating system processes, possibly located on different computers. The generated program may run on various parallel configurations without recompilation. We discuss the issues of flow control and load balancing and show the solutions offered by CAP. We also show how CAP can be used to generate relatively complex parallel programs incorporating neighbourhood dependent operations. Finally, we briefly describe a real 3D image processing application: the Visible Human Slice Server (http://visiblehuman.epfl.ch), its implementation according to the previously defined concepts and its performances.

Keywords. Parallel programming, image processing.

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 yields synchronous parallel programs, where communications and I/O operations do generally not overlap with computing operations. Creating parallel programs with completely asynchronous communications and I/O accesses is possible[6, 10], 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).

To be competitive, parallel processing needs to exploit the potentialities of the underlying parallel hardware and software (native operating system). Parallel applications may hide communication and disk access times by pipelining them with processing operations. When decomposing an image into tiles to be processed by a set of *n* processors, communication can be largely hidden if the original image is segmented into a number of tiles ($k \cdot n$) which is a multiple of the number of available processors (k integer, k >>1). Then, assuming that the computation is compute-bound, the total processing time is composed by the time to fill the pipeline, i.e. the time to send the first *n* tiles to the *n* processors, the time to compute in parallel $k \cdot n$ tiles and the time to send the last tile back to the master processor. If the pipeline set-up time is small in respect to the pure parallel computation time, a good speed-up may be attained. Similar considerations apply when hiding disk access times. For example, an application which requires 1 second disk access time and 1 second processing time can be executed as a pipeline comprising disk access and processing operations and take only slightly more than 1 second.

Ensuring on each contributing processing unit that data transfers to the network or to the disk are done at the same time as data processing requires generally several threads within the same address space : threads responsible for communications, threads responsible for I/O operations and a thread responsible for computation operations. These threads may synchronize when

1. {gennart,hersch}@di.epfl.ch, http://visiblehuman.epfl.ch

exchanging messages and through appropriate synchronization semaphores. Conceiving explicitly multi-threaded parallel applications is therefore difficult and error-prone. The alternative of programming in each processing unit an event loop relying on asynchronous message passing and file access primitives is also relatively difficult to achieve.

Further complexity is introduced if one would like to ensure load balancing, i.e. making sure that each slave processing unit is busy during the time that the computation goes on. Finally, image processing operations are often neighbourhood-dependent, i.e. to process a pixel or a voxel, the values of neighbouring pixels or voxels need to be known. Processing units need therefore to be able to exchange image tile borders in a synchronized manner.

In this paper, we would like to give a tutorial-like presentation of CAP, the *Computer-Aided Parallelization* tool we propose for simplifying the creation of efficient pipelined parallel image processing programs on distributed memory multiprocessor systems. CAP has been successfully applied for creating real applications, such as the Visible Human Slice Server (*http://visiblehuman.epfl.ch*) running on a multi-PC multi-disk platform.

2. COMPUTER-AIDED PARALLELIZATION: THE CONCEPT

The CAP Computer-Aided Parallelization tool has been designed with the goal of letting the parallel application programmer having the complete control about how his application is parallelized, and at the same time freeing him from the burden of managing explicitly a large number of threads and associated synchronization and communication primitives.

The CAP 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 parallel constructs specifying 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 suboperation to one of the program threads. The CAP preprocessor automatically compiles the high-level description into a C++ program source that implements the required schedule, i.e. the synchronizations and communications to satisfy the data dependencies underlying the parallel constructs. CAP also handles for a large part memory management and communication protocols, freeing the programmer from low level issues.

CAP operations are defined by a single input, a single output, and the computation that generates the output from the input. Input and output of operations are called tokens and are defined as C++ classes with serialization routines that enable the tokens to be packed, transferred across the network, and unpacked. Communication occurs only when the output token of an operation is transferred to the input of another operation.

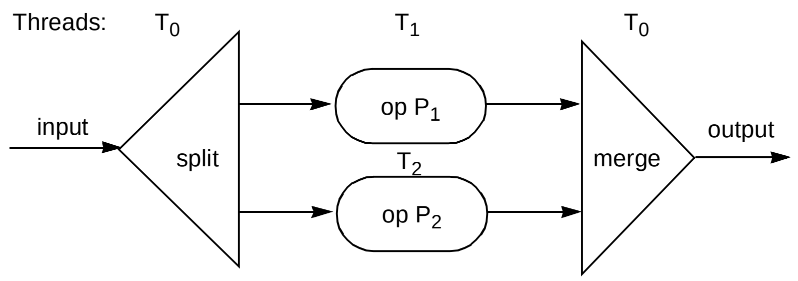


Fig. 1 Parallel split - merge construct.

An operation specified in CAP as a schedule of suboperations is called a parallel operation. A parallel operation specifies the assignment of suboperations to threads, and the data dependencies between suboperations. When two consecutive operations are assigned to different threads, the tokens are redirected from one thread to the other. As a result, parallel operations also specify communications and synchronizations between sequential 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 1). The mapping of the threads to the computing units is specified by a configuration file. Figure 2 shows a possible mapping.

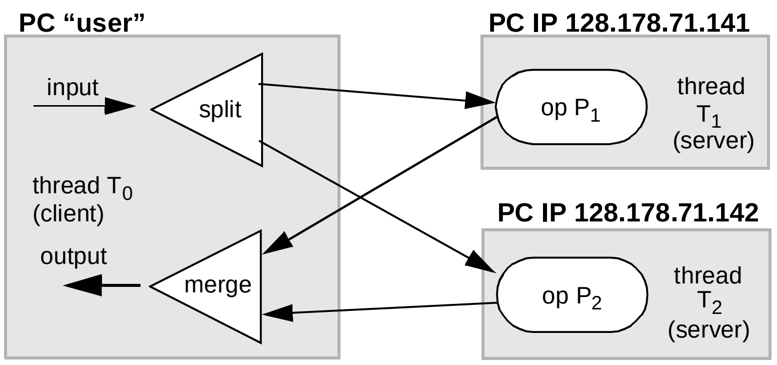


Fig. 2 Mapping of the parallel construct onto a simple multi-PC configuration.

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[3]. The macro data flow model which underlies the CAP approach has also been successfully used by the creators of the MENTAT parallel programming language [5].

Thanks to the automatic compilation of the parallel application, the application programmer does not need to explicitly program the protocols to exchange data between parallel 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. Supported platforms are WindowsNT and Unix.

3. INTRODUCTION TO THE CAP-BASED PARALLELIZATION

In a CAP program, the application developer specifies a set of threads (keyword ***process***), processing operations available within each thread (keyword ***operations***) and global variables (keyword ***variables***) in each thread which are maintained during the life of the thread. The basic CAP parallel construct comprises a *split* function, an *operation* possibly located in server threads and a *merge* function.

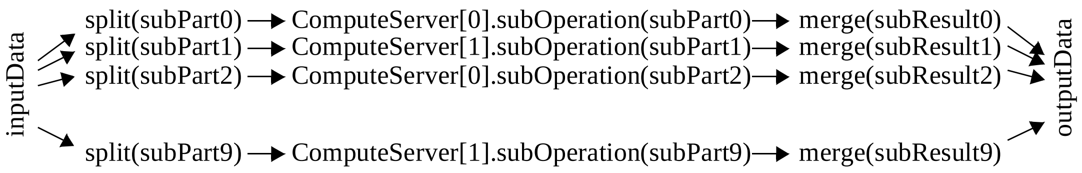

inputData
split(subPart0)
ComputeServer[0].subOperation(subPart0)
merge(subResult0)
split(subPart1)
ComputeServer[1].subOperation(subPart1)
merge(subResult1)
split(subPart2)
ComputeServer[0].subOperation(subPart2)
merge(subResult2)
split(subPart9)
ComputeServer[1].subOperation(subPart9)
merge(subResult9)
outputData

Fig. 3 Parallel execution of 10 suboperations on two compute servers

The *split* function is called *p* times to split the input data into *p* subparts which are distributed to the different compute server thread operations (***ComputeServer[i].subOperation***). Each operation running in a different thread ***ComputeServer[i]*** receives as input the subpart sent by the split function, processes this subpart and returns its subresult 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. Split and merge functions are executed as many times as specified in the split function (***parallel while*** construct) or as specified in the parallel construct iterator (***indexed parallel*** construct).

Fig. 3 shows the subdivision of an operation computing ***outputData*** from ***inputData*** into 10 suboperations computing ***subResult0*** to ***subResult9*** from ***subPart0*** to ***subPart9***. The suboperations are allocated evenly among two compute servers. ***subPart0*** results from the first call to the split function. The subresults are merged into the output data as soon as they are completed, i.e. ***subResult0*** is not necessarily merged first. All the operations have the potential to be performed in parallel, but subparts processed by the same compute server are processed sequentially.

```
int splitInput (splitInputTokenType* inputToken,
                splitInputTokenType* previousToken,
                splitOutputTokenType* & outputToken
               )
{ // sequential C++ body
  // programmer needs to create the outputToken
  // function returns 1 if split function is to
  // be called again, otherwise 0
}
```

CAP defines a standard way of passing data as input to the split function, to take the output of the split function and forward it as input to an operation, to take the output of an operation and to forward it as input to the merge function. Data passed between split, operation and merge functions is embedded into a ***token*** structure. Token types are defined at the beginning of the program.

```
10 leaf operation ComputeServerT::subOperation
11    in splitOutputTokenType* inputP
12    out mergeInputTokenType* outputP
13 {  // sequential C++ body }
14    // attention: outputP token needs
15    // to be created by programmer
16
17 void mergeOutput(mergeOutputTokenType* outputResult
18                  mergeInputTokenType* mergeInput)
19 {  // sequential C++ body }
20    // outputResult generated by CAP
```

Prog. 1. split, merge and leaf operation

Prog. 1 specifies the interface of typical split functions, leaf operations and merge functions. In the ***splitInput*** function (lines 1 to 9), the ***inputToken*** contains the input data to be divided into subparts ; the ***previousToken*** contains the subpart resulting from the previous call to the ***splitInput*** function, and the ***outputToken*** parameter is the current subpart being computed by the split function. The ***leaf operation*** contains the C++ code of the ***ComputeServerT::subOperation*** (lines 10 to 15), which computes a subresult (***outputP***) from a subpart (***inputP***). In the ***mergeOutput*** function (lines 17 to 20), the ***mergeInput*** parameter contains the subresult to be merged into the output data (***outputResult*** parameter). Leaf operations, split functions and merge functions are sequential procedures written in the C++ language. The programmer needs to create the output tokens of the split function and the output tokens of the operations. CAP directs automatically an output token to the input token of the next operation. CAP creates the merge function output token of type ***mergeOutputTokenType*** defined by the user.

```
 1 process ParallelServerT {
 2 subprocesses :
 3    MainProcessT Main;
 4    ComputeServerT ComputeServer[4];
 5 operations :
 6    ParallelComputation
 7       in splitInputTokenType* inputP
 8       out mergeOutputTokenType* outputP ;
 9 } ;
10
11 // instantiation of the high-level thread
12 ParallelServerT ParallelServer;
```

Prog. 2. thread hierarchy

CAP requires the programmer to specify explicitly the thread (or set of threads) which perform an operation. CAP is compositional, i.e. it enables to declare abstract high-level threads which include lower level "real" threads. Low-level threads are mapped to operating system threads. For example, in Prog. 2, the high-level thread ***ParallelServerT*** consists of one ***Main*** thread running in the same address as the main program, and 4 ***ComputeServer*** threads running on separate computers. Real threads perform sequential operations. High-level threads (or set of threads) perform parallel operations. A client program launches the parallel execution of a program by calling a high level operation (e.g. ***ParallelComputation***) which is part of a high-level thread (e.g. ***ParallelServerT***).

```
1 operation ParallelServerT::ParallelComputation
2    in splitInputTokenType* inputP
3    out mergeOutputTokenType* outputP
4 {
5    parallel while (splitInput, mergeOutput,
6                    Main, mergeOutputTokenType Result)
7      (ComputeServer[thisTokenP->index].subOperation);
8 }
```

Prog. 3. parallel operation

The high-level operation ***ParallelComputation*** contains a ***parallel while*** CAP construct enabling the client to split the input data into parts to be sent to operations running in threads located in the same or in different address spaces, possibly on different computers (PC's). The ***parallel while*** construct directs the token originating from the split function according to a user defined field (***index***) located in the token generated by the split function. The index of the destination thread contained in the field ***thisTokenP->index*** can be dynamically varied during the computation.

If the number of parallel branches is independent of the token generated by the split function, an ***indexed parallel*** construct can be used, which requires slightly modified split and merge functions (Prog. 4, lines 1 to 9). The corresponding ***indexed parallel*** construct has the structure coded in Prog. 4, lines 15 to 17).

```
 1 void splitInput (splitInputTokenType* inputToken,
 2                  splitOutputTokenType* & outputToken,
 3                  int index)                       // current index of splitInput call
 4 { ... // sequential C++ body }
 5
 6 void mergeOutput(mergeOutputTokenType* outputResult,
 7                  mergeInputTokenType* mergeInput,
 8                  int index)                       // current index of mergeOutput call
 9 { ... // sequential C++ body }                    // outputResult generated by CAP
10
11 operation ParallelServerT::ParallelComputation
12    in splitInputTokenType* inputP
13    out mergeOutputTokenType* outputP
14 {                                                 // this is the explicit index
15    indexed (int i = 0 ; i < NUMBER_OF_PARALLEL_ITERATIONS ; i++ )
16    parallel (splitInput, mergeOutput, Main, mergeOutputTokenType Result)
17      ( ComputeServer[i%NUMBER_OF_COMPUTESERVERS].subOperation ) ;
18 }
```

Prog. 4. Indexed parallel construct

In the case that the operations to be executed in parallel differ one from another (e.g. in the case of functional parallelism), a third parallel construct enables specifying custom split, custom operations and custom merge functions for each of the parallel branches. The syntax of the ***parallel*** operation is the following, for 4 parallel branches (Prog. 5). The ***Main*** thread executes the merge functions. According to the configuration file (next page), it runs in the same address space as the main program. The result token is ***Result*** of type ***mergeOutputTokenType***.

```
parallel (Main, mergeOutputTokenType Result) (
   (SplitInput0, ComputerServer[0].operation0, MergeOutput0)
   (SplitInput1, ComputerServer[1].operation1, MergeOutput1)
   (SplitInput2, ComputerServer[2].operation2, MergeOutput2)
   (SplitInput3, ComputerServer[3].operation3, MergeOutput3)
) ;
```

Prog. 5. Parallel construct

```
int main(int argc, char** argv)
{  // create input token
   splitInputTokenType* inputP = new (splitInputTokenType);
   mergeOutputTokenType* mainOutput;
   call ParallelServer.ParallelComputation
      in inputP out mainOutput;
   // display results contain in the output token mainOutput
   delete mainOutput;
   return 0;
}
```

Prog. 6. Main program

The ***ParallelComputation*** high-level operation may be called from the main C++ program running in the client (or master) thread. After making use of the results (printing them, storing them in a file or processing them further), it is the programmer's responsibility to delete the high-level parallel operation's output token.

Under WindowsNT, the program is developed in the Visual C++ environment as a single multithreaded program running on a single PC. The standard Visual C++ debugger is used to debug the Cap program. Once the program is running correctly as a single NT process, a configuration map can be created to run the program as several NT processes on the same PC. If the program behaves correctly, the configuration file can be adapted to run the program on multiple PC's. The communication between the processes relies on a TCP/IP socket-based message passing system.

```
// mandelbrot.cnf
configuration {
processes :
   A ( "user" ) ;
   B ( "128.178.71.141", "mandelbrotPar.exe" ) ;
threads :
   "Main" (A) ; // thread "Main" is located in the
                // same address space as the main program
   "Server[0]" (A) ;
   "Server[1]" (A) ;
   "Server[2]" (B) ;
   "Server[3]" (B) ;
} ;
```

Prog. 7. Configuration file

A *configuration file* specifies the mapping between CAP threads and underlying NT processes. A list of NT processes (A, B, C, etc..) is defined in the section ***processes*** and the Cap threads defined in the section ***threads*** are mapped to the declared NT processes (Prog. 7). In the configuration file example shown above, A and B are NT processes. Process A is associated to the PC where the program is started and process B is a server process running on the PC designated by its IP number. The executable file is given by its full path specifier. To provide a correct initial load distribution, two server threads execute on the master PC and two slave threads on the slave PC.

4. A DIDACTIC EXAMPLE : COMPUTING THE MANDELBROT SET

The Mandelbrot set is a set of complex numbers $\{c \in \mathbb{C}\}$, where after an infinite number of applications (in the program, **MAX_ITERATIONS**) of complex function $f_c(0) = z^2+c$, the resulting absolute value $|f_c^{\,n}(0)|$ is smaller than infinity (in the program, smaller than **MAX_MAGNITUDE**). The Mandelbrot set is included within a region of radius 2 from the center of origin.

The complex map showing the Mandelbrot set can be easily computed: we define the width of each pixel to be a given fraction, for example 1/100 and draw an image ranging from approximately (-2,-2) to (+2,+2).

The Mandelbrot program uses a simple ***parallel while*** loop for asking in round-robin manner the compute server threads to compute the image scanlines (see operation ***ParallelServerT::GlobalOperation***). The split function distributes scanline indices (token ***TileDescriptionT***) to the server thread operations (***ComputeMandelbrot***). Each server thread generates the scanlines it is asked to synthesize and sends each one as a token of type ***TileT*** to the merge function. The merge function merges the scanlines into the final image (token ***ImageT***). The full program, comprising the definition of constants, tokens, CAP threads, user functions, leaf operations and main program is shown.

4.1. Program head with user defined constants and functions

The ***MandelbrotFunction*** computes the color of each pixel in the Mandelbrot set, as a function of its x, y coordinates (Prog. 8).

```
/* mandelbrotPar.pc : simple parallel program for computing the Mandelbrot set */
#include "complex.h"             // complex number class
const int NUMBER_OF_COMPUTE_SERVERS = 4;
const int IMAGE_SIZE_X = 512; const int IMAGE_SIZE_Y = 512;

```

```
// Mandelbrot fonction constants
const double X_START = -2.0; const double Y_START = -2.0; const double GAP = 0.01;
const int    MAX_ITERATIONS = 200; const double MAX_MAGNITUDE = 200.0;


// Mandelbrot function renders magnitude after a number of iterations
// if magnitude very small (zero after rounding operation),
// (x,y) value belongs to the Mandelbrot set

unsigned char MandelbrotFunction(int x, int y){
   complex c(X_START + x*GAP, Y_START + y*GAP); complex z(0.0,0.0);
   int k = 0; double m;           // initial values
   while (z.Magnitude() < MAX_MAGNITUDE && k < MAX_ITERATIONS) {
      z = z * z + c;
      k++;
   }
   m = z.Magnitude();
   if (m>2550) m = 2550;          // limit the magnitude to the maximal diplayable value: 255 *10
   return (unsigned char)(m/10);  // return truncated magnitude as pixel intensity between 0 and 255
}
```

Prog. 8. Mandelbrot constants and function definition

4.2. Token definitions

The necessary set of tokens comprises one token to start the parallel computation, one token with the scanline index, one token incorporating one full scanline and the output token comprising the full image.

```
token StartT {};               // empty token is input to the split function:
                               // it specifies the start of the computation
token TileDescriptionT {       // this token is sent by the split function to specify the scanline index
   int lineIndex;              // scanline index
   TileDescriptionT (int Index) { lineIndex = Index; }
};
token TileT {                  // resulting image tile comprising one scanline
   int lineIndex;              // scanline index of computed line
   unsigned char buffer[IMAGE_SIZE_X];
   TileT (int index){
     lineIndex = index ;       // inline constructor
   }
};
token ImageT {                 // token comprising the full image,
                               // used as the output of the merge function
   unsigned char buffer[IMAGE_SIZE_X*IMAGE_SIZE_Y];
};
```

Prog. 9. Mandelbrot tokens

4.3. CAP threads

A high-level thread containing all other threads as well as the “Main” thread running in the same address space as the C++ main program are required. High-level threads incorporate high-level operations and normal threads incorporate leaf operations and possibly thread variables. Only normal threads are mapped to operating system threads.

```
process ParallelServerT {            // higher-level abstract thread with subthreads
subprocesses :
   MainProcessT Main ;
   ComputeServerT Server[NUMBER_OF_COMPUTE_SERVERS] ;
operations :                         // higher-level operation, callable from main program
   GlobalOperation in StartT* InputP out ImageT* OutputP ;
} ;

process ComputeServerT {             // declaration of compute server thread
operations :
   ComputeMandelbrot in TileDescriptionT* InputP out TileT* OutputP;
};

process MainProcessT {               // main process is the thread running the main program
operations :
} ;

ParallelServerT ParallelServer ;     // instantiation of high-level parallel thread
```

Prog. 10. Mandelbrot thread hierarchy

4.4. CAP operations, split and merge functions

A high-level CAP operation generally incorporates a parallel construct. This parallel construct either incorporates other CAP high-level operations or leaf operations and specifies the split and merge functions.

```
int SplitFunction (StartT* inputP,                 // split function with fixed sequence
   TileDescriptionT* previousP,                    // of input parameters
```

```
    TileDescriptionT*& nextP)
{
    int nextIndex = 0;                           // scanline index
    if (previousP!=0) nextIndex = previousP->lineIndex + 1;
    nextP = new TileDescriptionT(nextIndex);     // allocates output token with correct index value
    if (nextIndex == IMAGE_SIZE_Y-1) return 0;   // if last scanline, returns 0
    else return 1;                               // else continue calling the split function
}

leaf operation ComputeServerT::ComputeMandelbrot // code of leaf operations
    in TileDescriptionT* InputP                  // input token is coming from split fct
    out TileT* OutputP                           // output token travels to merge function
{
    OutputP = new TileT(InputP->lineIndex);      // allocates output token:
                                                 // tile comprising a single scanline
    for (int i=0; i< IMAGE_SIZE_X; i++)          // computes magn values of each pixel of scanline
      OutputP->buffer[i] = MandelbrotFunction(i,InputP->lineIndex);
}

void MergeFunction (ImageT* intoP, TileT* inputP)  // merges scanlines into output image buffer
{                                                  // copies one full scanline from
                                                   // &inputP->buffer to &intoP->buffer
    CopyMemory(&intoP->buffer[inputP->lineIndex*IMAGE_SIZE_X],&inputP->buffer, IMAGE_SIZE_X);
}

operation ParallelServerT::GlobalOperation       // definition of parallel operations
    in StartT* InputP
    out ImageT* OutputP
{
    parallel while ( SplitFunction, MergeFunction, Main, ImageT Result())
       ( Server[thisTokenP->lineIndex%NUMBER_OF_COMPUTE_SERVERS].ComputeMandelbrot) ;
}
```

Prog. 11. Mandelbrot parallel computation

4.5. Main program

The main program incorporates an instantiation of the token which is the input to the parallel ***GlobalOperation***. CAP generates the output token of the required type ***ImageT***.

```
int main ()                                 // main program
{
    long StartupTime = GetTickCount();
    long EndComputingTime;
    StartT* InputP = new StartT() ;         // input token to global operation
    ImageT* OutputP;                        // output token coming from global operations
                                            // calling the parallel operation
    call ParallelServer.GlobalOperation in InputP out OutputP ;

    EndComputingTime = GetTickCount();

    printf("Computing time [ms] is %l \n", EndComputingTime-StartupTime );
    return 0 ;
}
```

Prog. 12. Mandelbrot main program

5. FLOW CONTROL AND LOAD BALANCING ISSUES

In the current CAP implementation, the split function generates the tokens at a much higher rate than they can be consumed, i.e. processed by operations within the parallel construct and merged by the merging operation. Tokens may therefore accumulate in front of operations and merging functions. This may require considerable amounts of memory and induce disk swapping operations (transfer of virtual memory to and from disk).

```
flow_control (maxNbTokens)
indexed
   (int index = 0 ; index < indexMax ; index++)
parallel (splitfct, mergefct, Main, OutT outP)
   (ComputeServer[...].operation ) ;
```

Prog. 13. flow-control

Another problem is load balancing. In real applications, the load may be different in different compute servers. There is therefore a need to direct tokens generated by the split function towards a compute server which has terminated an operation on a previous token.

```
indexed
    ( int indexFC = 0 ; indexFC < maxNbTokens ; indexFC++ )
parallel (copyInputToken, copyOutputToken, Main, OutT outP) (
    for ( int nbCirculations = 0 ;
            nbCirculations<indexMax/maxNbTokens ;
            nbCirculations++ )
        (    Main.splitfct
        >-> ComputerServer[...].operation
        >-> Main.mergefct
        )
);
```

Prog. 14. CAP flow-control implementation

```
(
    flow_control (20)
    indexed
        ( int index = 0 ; index < indexMax ; index++ )
    parallel (splitfct, mergefct, main, OutT outP)
        ( ComputerServer[cap_fcindex0%NbOfComputeServers].
                operation );
)
```

Prog. 15. Load balancing

For the purpose of flow-control and load balancing, the CAP preprocessor translates an ***indexed parallel*** loop, respectively a ***parallel while*** loop, into a combination of ***indexed parallel***, respectively ***parallel while*** and a ***for*** CAP construct. The constructs shown in Prog. 13 is translated into the construct described in Prog. 14, where a token is recirculated in a ***for*** loop, and at each new entry into the ***for*** loop, the split function is called. The cycling around the ***for*** loop ensures that at one time, only ***maxNbTokens*** are in circulation. The application developer specifies by the instruction ***flow_control(maxNbTokens)*** the number of tokens in circulation, for example 20.

The *load-balancing* mechanism uses the same construct: tokens recirculate according to a ***for*** loop along the branch of an operation, which has just terminated a previous loop. For example, if the flow-control variable specifies 20 circulating tokens, then each token represents an independent execution branch, i.e. each token may be forwarded to an operation located possibly in a different address space. The execution branch index is available through the CAP variable ***cap_fcindex0***. This means that when a branch has terminated a single execution, it is again available to receive a token and to execute an operation. To ensure sufficient pipelining, several tokens should circulate in each compute server thread, for example 20 tokens circulating in 4 distinct compute server threads. The modulo operation ***cap_fcindex0%NbOfComputeServers*** specifies the current compute server thread index. An example of a construct enabling flow-control and load-balancing is the following shown in Prog. 16. To ensure both flow control and load balancing, the ***parallel while*** construct of the previously described ***MandelbrotPar*** program needs to be modified as follows:

```
operation ParallelServerT::GlobalOperation
    in ImageT* InputP
    out ImageT* OutputP

{   flow_control(20)
    parallel while (SplitFunction, MergeFunction, Main, ImageT Result())
    ( Server[cap_fcindex0%NUMBER_OF_COMPUTE_SERVERS].ComputeMandelbrot ) ;
}
```

Prog. 16. Flow-control, load balanced Mandelbrot parallel operation

To show that load balancing really works, a simple example consists in running the Mandelbrot program as a set of 4 slave CAP threads on two PC's, both with and without flow control, once distributed as 2 CAP threads per PC (equilibrated version) and once as 1 CAP thread on one PC and the other 3 CAP threads on the other PC (non equilibrated). Table 1 shows the results. The

Configuration	without flow control	with flow control
single PC	12.26 s	12.37 s
2 PC, equilibrated	6.63 s	6.62 s
2 PC, non equilibrated	9.55 s	6.67 s

TABLE 1. Mandelbrot program execution times on one and two Pentium II PC's

single processor execution time is around 12s. In the non-equilibrated configuration without flow-control, the second processor executes 3/4 of the jobs and should take around 9s (neglecting the communication overheads). The measured execution

time in that configuration is 9.55s. When flow control is enabled, the thread executing alone on the first PC consumes as many jobs as the three threads executing on the second PC, and the total execution time is down to around 6s. Flow control ensures that the PC with only one of the four CAP threads remains busy, i.e. more Mandelbrot image line generation requests are directed to that CAP thread than to each of the other 3 CAP threads located in the second PC.

6. NEIGHBORHOOD DEPENDENT PARALLEL OPERATIONS

In the previous sections, the parallel program considered only applications, where the operations running in parallel in different threads were able to proceed independently on their subset of data. In many real application cases (for example image filtering or image segmentation[11]), each processing element (thread) must, at a certain point of its program execution receive information from neighbouring processing elements (threads). As an example, let us consider the parallelization of the game of Life.

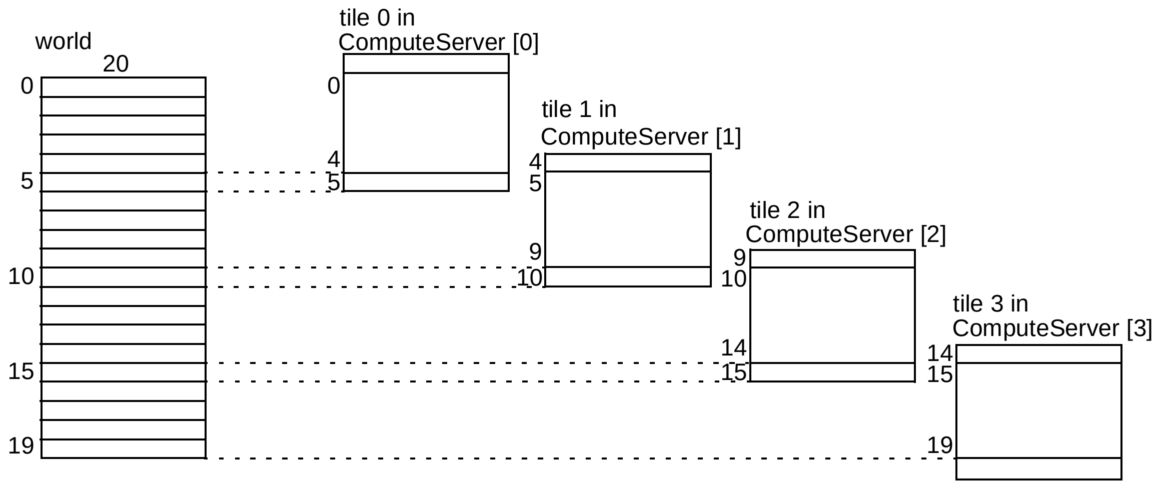

world
20
0
5
10
15
19
tile 0 in ComputeServer [0]
tile 1 in ComputeServer [1]
tile 2 in ComputeServer [2]
tile 3 in ComputeServer [3]
0
4
5
9
10
14
15
19

Fig. 4 The world, the tiles and the array located in server threads.

The world is made of a rectangular array of cells, which are either dead or alive. The state of a cell in the next cycle depends on its actual state as well as on the states of its 8 immediate neighbors according to the following rule: A living cell with 2 or 3 living cells remains alive, otherwise it dies. A dead cell with 3 neighbors becomes alive, otherwise it remains dead.

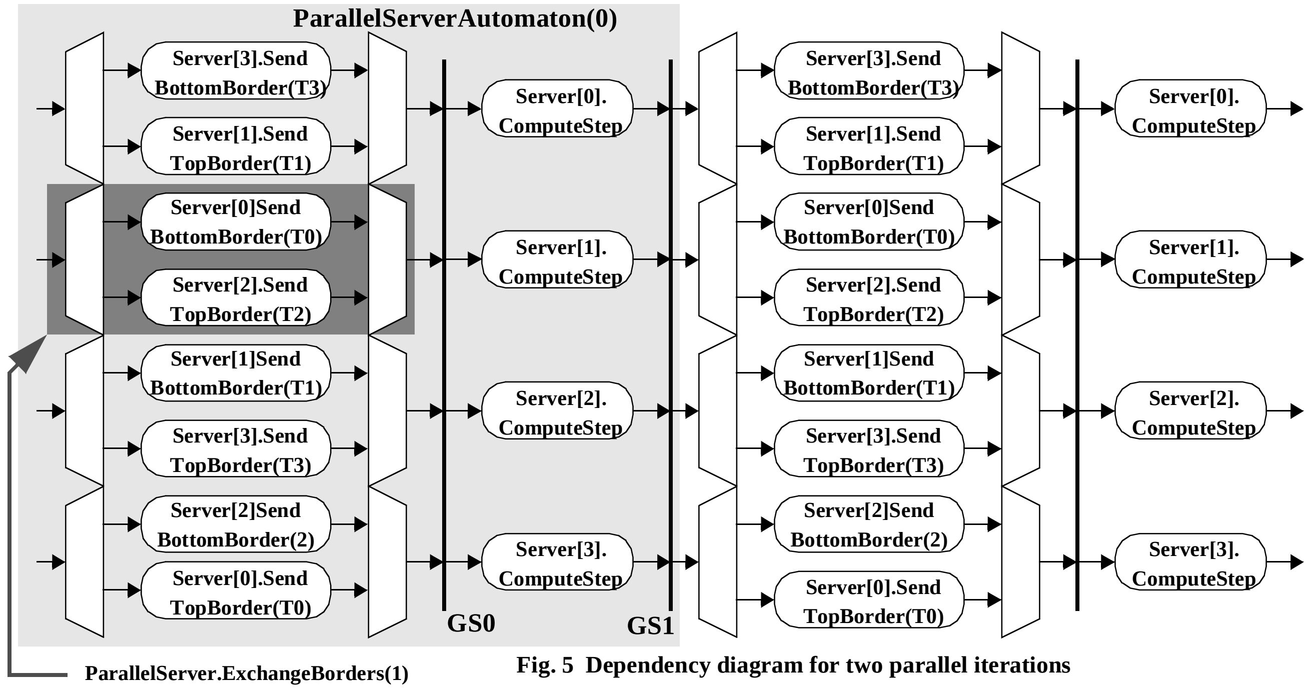

ParallelServerAutomaton(0)
Server[3].Send BottomBorder(T3)
Server[1].Send TopBorder(T1)
Server[0]Send BottomBorder(T0)
Server[2].Send TopBorder(T2)
Server[1]Send BottomBorder(T1)
Server[3].Send TopBorder(T3)
Server[2]Send BottomBorder(2)
Server[0].Send TopBorder(T0)
Server[0]. ComputeStep
Server[1]. ComputeStep
Server[2]. ComputeStep
Server[3]. ComputeStep
GS0
GS1
ParallelServer.ExchangeBorders(1)

Fig. 5 Dependency diagram for two parallel iterations

To parallelize the game of life, we segment the world into horizontal tiles whose vertical size is the total size divided by the number of compute server threads. In each server thread, we create an array of the size of the horizontal tile plus two horizontal lines: one for the top border which is a copy of the most bottom line belonging to the previous tile and one for the bottom border which is a copy of the most top line belonging to the following tile (Fig. 4).The parallel program comprises the following stages:

1. Read the world from a file and copy it into the arrays located in server threads. For this purpose, the world is read from a file, possibly passed as a parameter from the command line and tokens comprising each one part of the world are sent to each server thread (split-merge construct).

2. Each parallel iteration, i.e. each computation of the new state of the tiles making up the world includes two steps (Fig. 5): (a) each thread must ask its neighbouring tiles to send its top and bottom neighbour lines and must merge these lines into its cell array (tile); (b) each thread must compute the new state of its part of the world based on the present state.
3. Once all iterations are terminated, a high-level parallel ***GetWorld*** operation collects the tiles computed by all threads and merges them in the resulting output token.

The graphical representation of the schedule of operations for two parallel iterations (step 2 above), in the case of a 4 tile world is shown in Fig. 5. In each step, the servers first exchange borders, synchronize to ensure that all borders have been received (GS0), compute their tile, and synchronize again (GS1), before staring the next iteration step.

The main program comprises three calls to parallel CAP constructs : one for initialization of the world, one for computing the iterations and one for gathering the results.

```
call ParallelServer.parallelInitPartWorld in fullworldP out resultP ;
call ParallelServer.Automaton (NB_ITERATIONS) in inputP out outputP;
call ParallelServer.GetWorld in inputP out lastResultP;
```

Prog. 17. Main program parallel operation call sequence

Let us assume that in each server thread, the tile representing a part of the world has been correctly initialized by the high-level ***parallelInitPartWorld*** parallel operation.

The ***Automaton*** high-level parallel operation comprises a sequence of two ***indexed parallel*** constructs: one for exchanging and merging tile borders and one for the computing the tiles' new state (game of life iteration, Prog. 18). The syntax of Prog. 18 matches the graphical representation of Fig. 5. The light gray box (single automaton step) in Fig. 5 is specified at lines 10 to 18. The first ***indexed parallel*** construct (line 10 to 13) specifies that all compute servers exchange borders simultaneously (a single ***ExchangeBorders*** operation is shown as a dark gray box in Fig. 5). The second ***indexed parallel*** construct specifies that all compute servers compute their respective tile in parallel (lines 15 to 18). The repetition of the automaton step is specified using the CAP for loop (line 9).

```
 1   // on Main, sequences the indexed parallel ExchangeBorder and the following
 2   // indexed parallel ComputeStep
 3
 4   operation ParallelServerT::Automaton(int nbIterations)
 5      in void* InputP
 6      out void* OutputP
 7   {
 8      // single iteration first
 9      for (int it = 0 ; it < nbIterations ; it++ ) (  // iterations ExchangeBorders->ComputeStep
10         indexed                                      // to synchronize exchange of borders
11            ( int i = 0 ; i < NUMBER_OF_COMPUTE_SERVERS; i++)
12         parallel ( void, void, Main, void output)
13            ( ExchangeBorders (i) )
14         >->
15         indexed
16            ( int j = 0; j < NUMBER_OF_COMPUTE_SERVERS ; j++ )
17         parallel ( void, void, Main, void output)
18            ( Server[j].ComputeStep )
19      ) ;
20   }
```

Prog. 18. CAP specification of the automaton iteration step

The ***ExchangeBorders*** operation is itself a high-level parallel operation comprising a ***parallel*** construct. Within each branch of the parallel construct, split operations are empty (***void***) and merge operations copy a top or a bottom border into the thread's tile. The merge of the parallel operation is executed in thread ***Server[partIx]*** and the operations themselves, i.e. respectively ***sendBottomBorder*** and ***sendTopBorder*** are executed in the thread responsible for the tiles above and respectively below the current tile (***Server[partIx+1], Server[partIx-1]***).

```
// this operation asks in parallel the neighbours to send their borders and merges them
// exchange of borders works in wraparound mode

operation ParallelServerT::ExchangeBorders (int partIx)
   in void* InputP
   out void* OutputP
{
   parallel (Server[partIx], void result) (  // indicates merge in Server thread
      ( void
      , ifelse (partIx>0)
         ( Server[partIx-1].sendBottomBorder)                      // partIx>0
         ( Server[NUMBER_OF_COMPUTE_SERVERS-1].sendBottomBorder) // partIx==0
      , mergeBorders(TopBorder)
      )
      ( void
      , ifelse (partIx<NUMBER_OF_COMPUTE_SERVERS-1)       // partIx<NUMBER_OF_COMPUTE_SERVERS-1
         ( Server[partIx+1].sendTopBorder)               // partIx==NUMBER_OF_COMPUTE_SERVERS-1
```

```
            ( Server[0].sendTopBorder)
        , mergeBorders(BottomBorder)
        )
    ) ;
}
```

Prog. 19. CAP specification of the ExchangeBorders operation

According to the presented solution (operation ***Automaton***), all borders are exchanged in parallel and all computations are executed in parallel. However, there is no overlap between the exchange of borders and the computations. In order to introduce such an overlap, we can separate the ***ComputeStep*** procedure into a ***ComputeCenter*** procedure, i.e. the computation of those cells whose 8-neighbours are located within the present tile and into a ***ComputerBorder*** procedure, whose cells are located at the border of the current tile. Then, the central part may be computed at the same time as borders are exchanged between neighbouring tiles. The parallel program needs only a very slight modification: the ***ExchangeBorders*** high-level operation becomes the ***SendBorderCompCenter*** operation incorporating the new ***ComputeCenter*** leaf operation.

```
// this operation asks in parallel the neighbours to send their borders and merges them

operation ParallelServerT::SendBordersCompCenter (int partIx)
    in void* InputP
    out void* OutputP
{
    parallel (Server[partIx], void result) (        // merge functions executed in thread Server[partIx]
        ( void                                      // no split function
        , Server[partIx].ComputeCenter              // computeCenter executed in current thread
        , void                                      // no merging operation
        )
        ( void
        , ifelse (partIx>0)
            ( Server[partIx-1].sendBottomBorder)                       // partIx > 0
            ( Server[NUMBER_OF_COMPUTE_SERVERS-1].sendBottomBorder)    // partIx==0, take last tile
        , mergeBorders (TopBorder)
        )
        ( void
        , ifelse (partIx<NUMBER_OF_COMPUTE_SERVERS-1)
            ( Server[partIx+1].sendTopBorder )              // partIx<NUMBER_OF_COMPUTE_SERVERS-1
            ( Server[0].sendTopBorder )                     // partIx==NUMBER_OF_COMPUTE_SERVERS-1
        , mergeBorders (BottomBorder)
        )
    );
}
```

Prog. 20. CAP specification of the improved ExchangeBorders operation

The high-level operation ***Automaton***, which expresses the global parallel behavior of the application keeps the same structure, with ***ExchangeBorders (i)*** becoming ***ExchangeBordersCompCenter(i)*** and with ***ComputeStep*** becoming ***ComputeBorders***.

```
// on Main, sequences the indexed parallel ExchangeBorder and the following
// indexed parallel ComputeStep

operation ParallelServerT::Automaton(int nbIterations)
    in void* InputP
    out void* OutputP
{
    // single iteration first
    for (int it = 0 ; it < nbIterations ; it++) (
        indexed (int i=0; i<NUMBER_OF_COMPUTE_SERVERS; i++)
        parallel (void,void,Main,void output)
         ( ExchangeBordersCompCenter (i) )
        >->
        indexed
            (int j=0; j<NUMBER_OF_COMPUTE_SERVERS; j++)
        parallel (void,void,Main,void output)
            ( Server[j].ComputeBorders)
    ) ;
}
```

Prog. 21. CAP specification of the improved Automaton operation

This example demonstrates that the parallel behavior of the program is concentrated in a few high-level operations and that it can easily be modified to experiment with alternative parallelization schemes.

7. A REAL APPLICATION: THE VISIBLE HUMAN SLICE SERVER

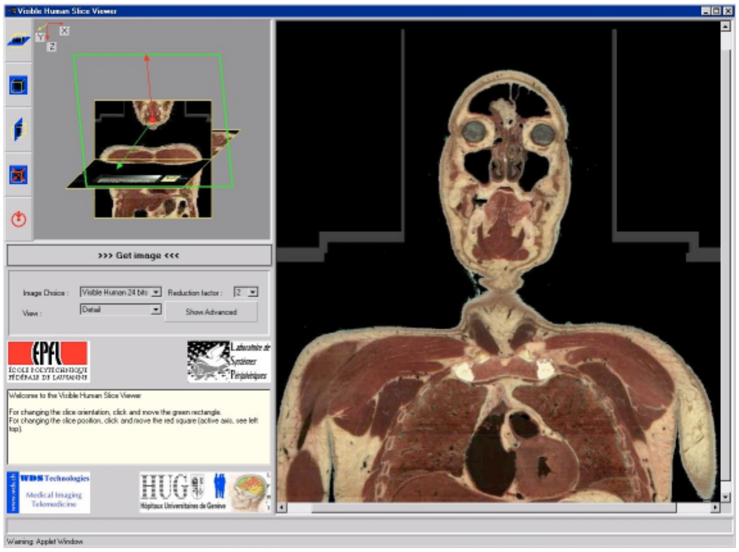


Image Choice : Visible Human 24 bits; Reduction factor : 2; View : Detail

Welcome to the Visible Human Slice Viewer

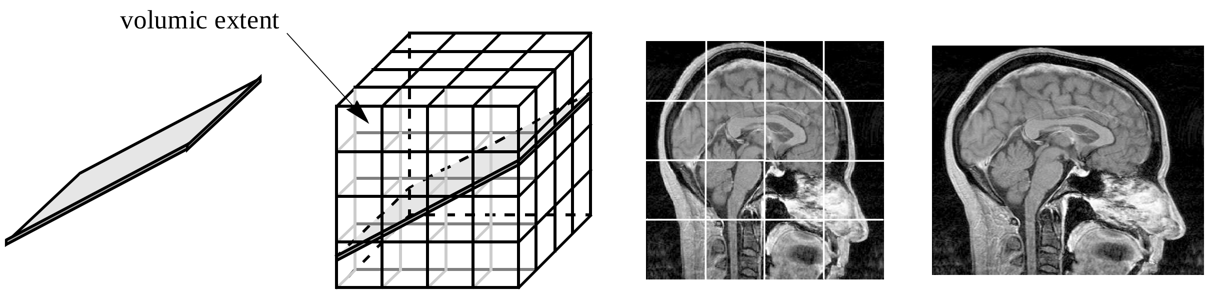
For changing the slice orientation, click and move the green rectangle.
For changing the slice position, click and move the red square (active axis, see left top).

Fig. 6 Selecting within a Java applet an image slice within a miniaturized 3D tomographic image

Thanks to CAP, a parallel Visible Human Slice Server has been developed, which offers the capability of interactively specifying the exact position and orientation of a desired slice (Figure 1) and of requesting and obtaining that slice from a 3D tomographic volume, made of either CT, MRI or cryosection images (digital color photographs of cross-sections).

Accessing and extracting slices from the 3D Visible Human image[1] requires high processing power and considerable storage space, e.g. 13GBytes for the human male dataset. We decided therefore to build the Visible Human Slice Server on top of 5 Bi-Pentium Pro 200MHz PC's. An additional Bi-Pentium II 333MHz PC acts as the client on which the slices are visualized. All the PC's are interconnected through a 100Mbits/s Fast Ethernet switch. A total of 60 disks is equally distributed among the server's 5 Bi-Pentium Pro PC's.

For enabling parallel access to its data, the Visible Human 3D volume is segmented into *volumic extents* of size 32x32x17 RGB voxels, i.e. 51KBytes, which are striped over the 60 disks residing on the 5 server PC's[8]. In order to extract an image slice from the 3D image, the extents intersecting the slice are read and the slice parts contained in these volumic extents are extracted and projected onto the display space (Figure 5).



Slice specification

Extraction of the digital slice from the 3D image

Extracted slice parts

Projected slice parts merged into the final displayable slice

Fig. 7 Extraction of slice parts from volumic file extents

The parallel slice server application consists of a client PC and of server processes running on the server's parallel PC's. The client PC interprets the slice location and orientation parameters defined by the user 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 part extraction requests. These servers execute the requests and transfer the resulting slice parts to the client PC which assembles them into the final displayable image slice. The parallel slice server application is described by the diagram of Fig. 6. This diagram, translated to the CAP language, corresponds to the *parallel while* operation described in Fig. 7.

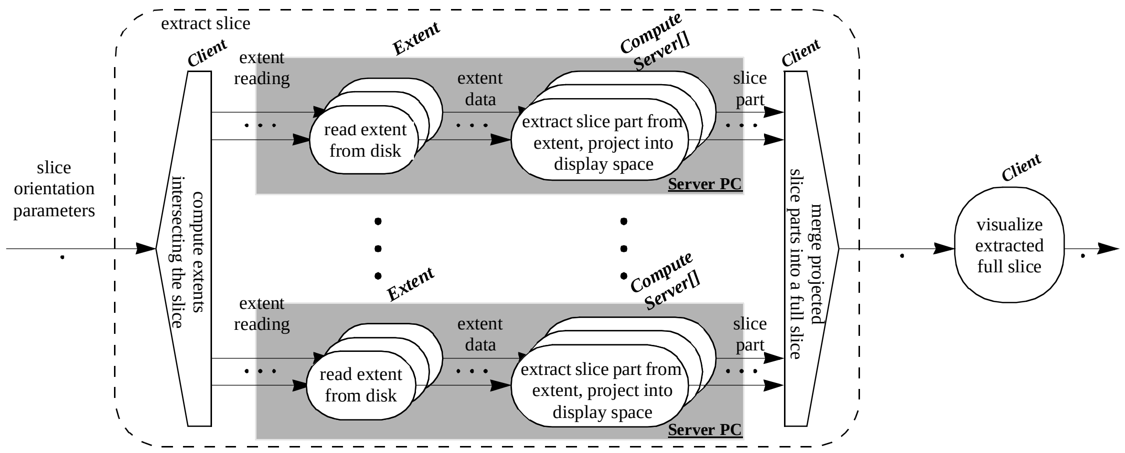

extract slice
Client
Extent
Compute Server[]
Client
extent reading
extent data
slice part
read extent from disk
extract slice part from extent, project into display space
Server PC
slice orientation parameters
compute extents intersecting the slice
merge projected slice parts into a full slice
Client
visualize extracted full slice

Fig. 8 Graphical representation of the pipelined parallel extent access and slice extraction operations

The parallel application consists of one large *split-merge* construct. The parallel branches comprise each an I/O operation to read one volumic extent from disk and a computing operation to extract and resample the part of the specified image slice intersecting the volumic extent previously read from disk. By reading extents asynchronously from disks, disks access operations and computing operations are completely pipelined, i.e. the computation time is hidden by the disk access time or vice-versa, depending if disk access time or computation time represents the bottleneck for a given hardware configuration i.e for a given number of PC's and a given number of disks per PC. Note that since the split function generates hundreds of extent access requests, each contributing server PC launches many simultaneous asynchronous extent access requests, providing thereby a global bandwidth proportional to the number of contributing disks.

split function   merge function   output token of the parallel while construct

```
operation Ps2ServerT::ExtractSlice
  in SliceExtractionRequestT* InputP
  out SliceT* OutputP
(
  parallel while (SplitSliceRequest, MergeSlicePart, Client, SliceT Output)
  (
    ExtentServer[thisTokenP->ExtentServerIndex].ReadExtent
    >->
    ComputeServer[thisTokenP->ComputeServerIndex].ExtractAndProjectSlicePart
  };
)
```

Prog. 22. CAP construct for the pipelined parallel extent access and slice extraction operations

The server's performance has been measured by striping the Visible Human (male dataset) onto 1 to 5 Bi-Pentium Pro server PC's and onto 1 to 12 disks per server PC (max. 60 disks). Figure 8 shows the number of extracted 512x512 colour slices per second for various configurations.

Each slice access request is decomposed into 437 volumic extent access requests (22 MBytes). For all the server configurations, disk I/O bandwidth is always the bottleneck (effective single disk throughput for 51KB blocks: 1.88 MBytes/s). With 4.8 image slices/s, the client PC is able to receive from the Fast Ethernet 7.8 MBytes/s of slice parts. These performances are close to the performances offered by the underlying hardware, operating system (Windows NT) and network protocols (TCP/IP).

A scaled-down version of the server comprising a single Bi-Pentium-II PC and 16 disks has been installed as a permanent Web Server and offers its interactive slicing services at *http://visiblehuman.epfl.ch*.

8. CONCLUSIONS

We presented in a didactic manner the *Computer-Aided Parallelization* tool we propose for simplifying the creation of efficient pipelined parallel image processing applications. Application programmers specify at a high level of abstraction the set of threads present in the application, the processing operations offered by these threads, and the high-level parallel constructs specifying the flow of data and parameters between operations. The generated program can run on various parallel configurations without recompilation. Only the configuration file mapping the CAP threads to NT processes and PC's needs to be modified. CAP also incorporates mechanisms for flow-control and load-balancing.

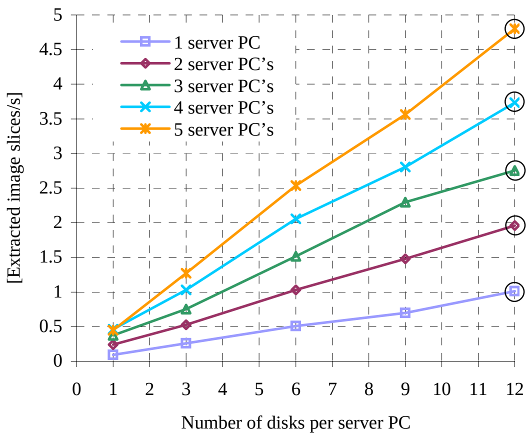


Fig. 9 Slice extraction performance under various configurations, without disk caching

Besides the Visible Human Slice Server, CAP has been applied successfully to a number of applications, both in the field of image processing[4] and in the field scientific computing[3,7]. We have shown[3] that the overhead specific to CAP is very low: each token incorporates in addition to its user-defined structure a 24 bytes header. The time to transfer this additional amount of information is generally negligible, when compared to the latency to launch the transfer of one block of data on commercially available networks (packet transfer latency on Fast Ethernet: 300 µs). Therefore, the overhead of CAP is generally negligibly small. Since CAP generates schedules described by directed acyclic graphs, CAP programs are deadlock free by construction.

We are currently preparing a distribution of CAP available to interested users for teaching, research, and demonstration purposes. This distribution will be downloadable from the Web.

Bibliography

1. M. 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.
2. I. T. Foster, *Designing and Building Parallel Programs*, Addison-Wesley Publishing Company, 1995.
3. B. A. Gennart, J. Tarraga, R. D. Hersch, "Computer-Assisted Generation of PVM/C++ Programs using CAP", In *Proc. of EuroPVM'96*, LNCS 1156, Springer Verlag, Munich, Germany, October 1996, pp. 259-269.
4. B. A. Gennart, M. Mazzariol, V. Messerli, R. 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, pp. 446-458.
5. A. S. Grimshaw, "Easy-to-use Object-Oriented Parallel Processing with Mentat", IEEE Computer, Vol. 26, No. 5, May 1993, 39-51.
6. W. Gropp, E. Lusk, A. Skjellum, *Using MPI*, The MIT Press, 1994.
7. M. Mazzariol, B. A. Gennart, V. Messerli, R. D. Hersch, "Performance of CAP-Specified Linear Algebra Algorithms", In *Proc. of EuroPVM-MPI'97*, LNCS 1332, Springer Verlag, Krakow, Poland, November 1997, pp. 351-358.
8. V. Messerli, O. Figueiredo, B. A. Gennart, and R. D. Hersch. Parallelizing I/O-intensive image access and processing applications. *IEEE Concurrency*, 7(2):28-37. April-June 1999.
9. V. Messerli, B. A. Gennart, R. D. Hersch, "Performances of the PS2 Parallel Storage and Processing System", In *Proc. of the 1997 International Conference on Parallel and Distributed Systems*, IEEE Computer Society Press, Seoul, Korea, December 1997, pp. 514-522.
10. Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," Technical Report, July 1997, http://www.mpi-forum.org.
11. I. Pitas (Ed.), *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, J. Wiley, 1993.
12. G. V. Wilson, P. Lu (Eds), *Parallel Programming Using C++*, The MIT Press, 1996.