View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

PROGRAM PARALLELIZATION WITH CAP

1. Introduction

A parallel program comprises generally a *set of threads* containing parallel processing operations performed by the parallel program. Most parallel programs are based on the *client-server* (also called master-slave) paradigm. Parallel processing is initiated by a client thread (also called *master*) which communicates with the server threads (also called s*laves*) located in the server processing nodes. In the case the data to be processed can be segmented in chunks, the client *splits* the input data into sub-parts which are sent for processing to the server nodes. Each server computes its sub-result and sends it back to the client. The client *merges* all sub-results into the final result.

Generally, parallel programs execute *in Single Program Multiple Data* (SPMD) mode: the same program is distributed on both the client and the server threads and at start time, every program instance checks if it has to behave as a client or as a server, and executes accordingly either the client or the server code.

To facilitate development and debugging, a parallel program is first developed as a set of threads (also called lightweight processes) residing within the same process. Such a program can be debugged with standard debugging tools. In a second step, the threads are mapped to different processes running on the same computer and the program is further tested and debugged. Finally, the parallel application is completed by having the threads mapped onto different processes running in different computers. No further debugging should be necessary.

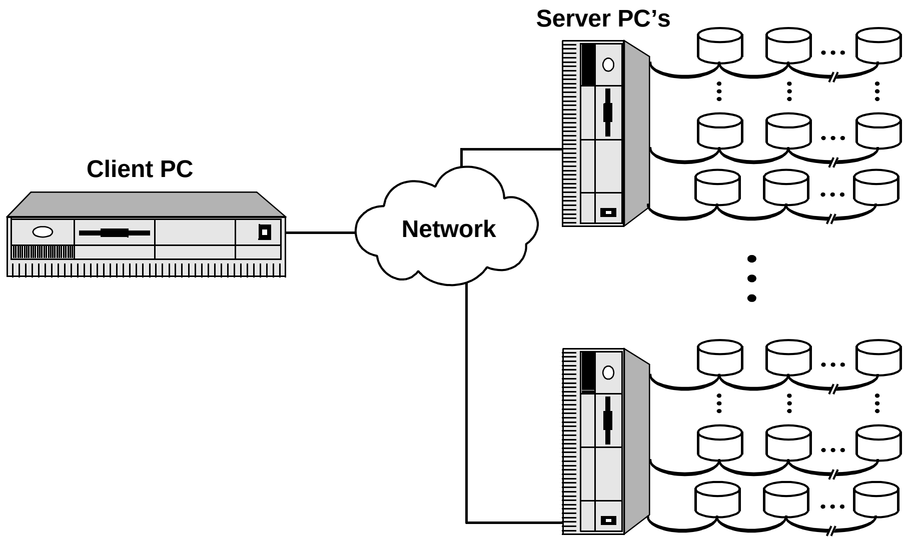

Fig. 1 Parallel program comprising client and server threads.

Most operating systems supporting distributed environments, such as Unix or WindowsNT offer primitives enabling threads running on different processing nodes to communicate with one another (named pipes, socket based TCP-IP communication). However, developing parallel programs making direct use of operating system threads and pipes is rather tedious. Threads need to be explicitly created and managed and for each pair of communicating processes, a named pipe needs be explicitely created and opened.

1.1 Message passing based parallel programming (MPI)

In order to facilite to development of parallel programs, the parallel programming community developed the MPI message passing based programming environment. This environment offers simple

MPI_Send(buffer, count, datatype, destProcessId, Tag, communicator)

and

MPI_Receive(buffer, count, datatype, sourceProcessId, Tag, communicator, status)

communication primitives for transfering data located in a buffer from the current thread to the destination thread. Programming at the message passing level is however difficult for the following reasons :

- two threads may comprise each different procedures communicating with each other : the only way to distinguish between messages is to use the Tag field (integer).
- At the end of a computation, the client may have to receive results simultaneously from server threads. This is done by a polling function testing continuously if a message arrives from one of the server threads. To avoid successive polling of all message sources, MPI provides the *MPI_gather()* communication primitive to gather messages from all present threads.

Furthermore, having only one thread running in each computer is highly inefficient: when this thread waits for a message from another thread or it executes a file read or write operation, the processor remains idle. Trying to make the parallel program more efficient by developing multi-thread client and server programs is even more difficult: one has to manage explicitely synchronizations and information exchanges between threads running either on the same or on different computers.

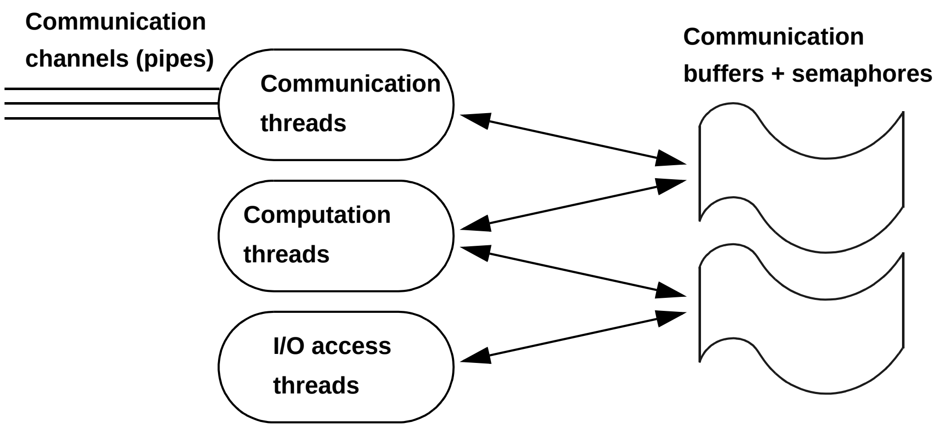
Communication channels (pipes)
Communication threads
Communication buffers + semaphores
Computation threads
I/O access threads

Fig. 2 Multithread multiprocessor programs.

An alternative is to view a single running server thread as an event loop: on a given event (for example, receiving a certain message), the thread accomplishes a certain action. To avoid any kind of thread suspension, sending messages and accessing the disk must be performed asynchronously. A call back procedure may be associated to the termination of an input/output step, for example reading a file. Within the call-back procedure, the next step to be accomplished can be launched.

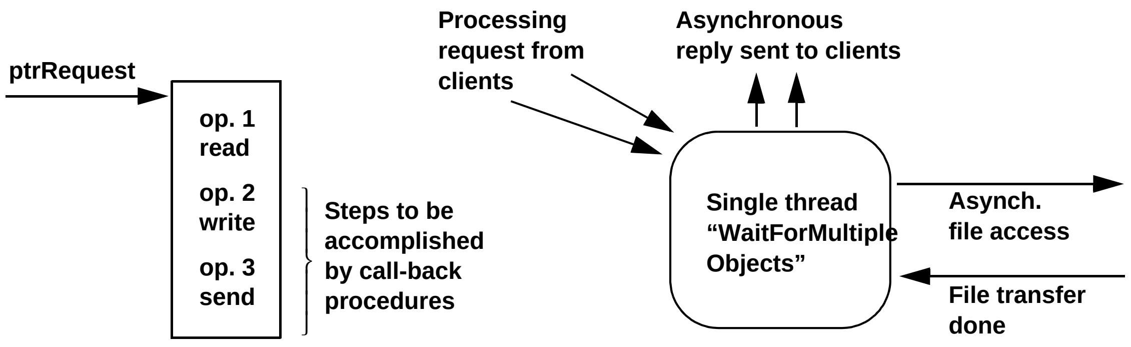
ptrRequest
op. 1 read
op. 2 write
op. 3 send
Steps to be accomplished by call-back procedures
Processing request from clients
Asynchronous reply sent to clients
Single thread "WaitForMultiple Objects"
Asynch. file access
File transfer done

Fig. 3 Programming an event loop: a request corresponds to a list of steps to be accomplished possibly by call-back procedures.

1.2 Hiding data communication and disks access time

Parallel programming can only be competitive, if the potentialities of the underlying parallel hardware and software (native operating system) are fully exploited. To offer good performances, parallel applications need to hide communication time, i.e. the time to transfer the data between the client and the server threads, the time to transfer data among the server threads and the time to transfer the results back to the client thread. In addition, disk access times (disk read and write operations are done by DMA hardware and do not need processing power) need also be hidden.

Transfer and disk access times can be hidden if data transfers or disk accesses are done in parallel with computations. This can be done by pipelining data transfers or disk accesses with processing operations. 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 a total of 1.1 second.

With a message passing interface, ensuring that data transfers are done at the same time as data processing requires generally two communicating threads within the same address space : one thread is responsible for communication and the other thread responsible for the computation. Both threads may synchronize each other with an input and output message queue and appropriate synchronization semaphores (Fig. 2). Conceiving explicitely multi-threaded parallel applications is difficult and error-prone. The alternative of programming in each address space an event loop relying on asychronous message passing and file access primitives is also relatively complex to achieve (Fig. 3).

2. Introduction to the CAP-based parallelization

To develop efficient parallel programs, we need a means of specifying a *set of threads* running possibly in the same or in different address spaces (possibly on different computers), associating to each of the threads one or several *operations*, and specifying at a high level of abstraction the flow of parameters and data between the operations located in different threads.

The *Computer-Aided Parallelization framework* (CAP) aims at providing exactly this functionality. It offers the parallel application developer the capability of describing his program at a high-level of abstraction. This high-level description is then automatically compiled into a C++ program source incorporating all low level communication and synchronization features required to ensure (1) that parameters and data are correctly transferred between operations located in different threads and (2) that correct pipelining is achieved, i.e. that data is transferred or read from disks while previous data is being processed.

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 one or several server threads and a *merge* function:

```
split() ..
ComputerServer[i].myOperation()
merge()
```

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].myOperation()*) . 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 explicitely in which thread the merge function is executed (often in the same thread as the split function). It receives a number of subresults equal to the number of subparts sent by the split function. Split and merge functions are executed as many times as specified in the split function (*parallel while* construct) or as specified in the parallel construct iterator (*indexed parallel* construct).

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 in a *token* structure. Token types are defined at the beginning of the program. Here is the definition of the structure and sequence of parameters passed to the split function, operation and merge functions.

```
int splitInput (splitInputTokenType* inputToken,           // input token to the split function
                splitInputTokenType* previousToken,        // previous token, zero if first pass
                splitOutputTokenType* & outputToken,       // output of split function created by user
               )
{ ..// sequential C++ body
    // programmer needs to create the outputToken
    // function returns 1 if split function is to be called again, otherwise 0
}
leaf operation ComputeServerT::myOperation()
     in splitOutputTokenType* inputP
     out mergeInputTokenType* outputP
{ ..// sequential C++ body }                 // attention: outputP token needs
                                             // to be created by programmer

void mergeOutput(mergeOutputTokenType* outputResult,
                 mergeInputTokenType* mergeInput)
{ ..// sequential C++ body }                 // outputResult generated by CAP
```

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.

Leaf operations, split functions and merge functions are sequential procedures written in the C++ language. 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, the CAP construct residing on the client which launches the parallel execution is the high level operation *ParallelComputation* which is part of the high-level thread *ParallelServerT*.

```
process ParallelServerT {
subprocesses :
  MainProcessT Main;                 // thread that runs in the same address space as the main program
  ComputeServerT ComputeServer[NUMBER_OF_COMPUTESERVERS]; // compute server threads
Operations :
  ParallelComputation in splitInputTokenType* inputP
                      out mergeOutputTokenType* outputP ;
} ;
ParallelServerT ParallelServer;      // instantiation of the high-level thread
```

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.

```
  operation ParallelServerT::ParallelComputation                thread
   in splitInputTokenType* inputP                  merge        executing       result and
   out mergeOutputTokenType* outputP   split function  function     merge function  its type
  {
    parallel while (splitInput, mergeOutput, Main, mergeOutputTokenType Result)
      (ComputeServer[thisTokenP->index].myOperation)
    ;
  }
```

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 :

```
void splitInput (splitInputTokenType* inputToken,
                 splitOutputTokenType* & outputToken,
                 int index)                  // current index of splitInput call
{ ..// sequential C++ body }

void mergeOutput(mergeOutputTokenType* outputResult,
                 mergeInputTokenType* mergeInput,
                 int index)                  // current index of mergeOutput call
{ ..// sequential C++ body }                 // outputResult generated by CAP
```

The corresponding indexed parallel construct has the following structure

```
operation ParallelServerT::ParallelComputation
  in splitInputTokenType* inputP
  out mergeOutputTokenType* outputP
{
  indexed (int i=0 ; i<NUMBER_OF_PARALLEL_ITERATIONS; i++)         // this is the explicit index
  parallel (splitInput, mergeOutput, Main, mergeOutputTokenType Result)
    (ComputeServer[i%NUMBER_OF_COMPUTESERVERS].myOperation) ;
}
```

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:

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

The *Main* thread executes the merge functions. According to the configuration file (see below), it runs in the same address space as the main program. The result token is *Result* of type *mergeOutputTokenType*.

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

```
int main(int argc, char** argv)
{
  splitInputTokenType* inputP= new (splitInputTokenType);          // create input token
  mergeOutputTokenType* mainOutput;                                // define output token pointer
  call ParallelServer.ParallelComputation in inputP out mainOutput;// calling the high-level operation

                      // process or display the results located in token mainOutput
  delete mainOutput;  // Delete the parallel operation's output token
  return 0;
}
```

Under WindowsNT, the program is developed in the Visual C++ environment as a single multihread 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 [Messerli98, chapter 4].

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.

```
       //mandelbrot.cnf
       configuration {
       processes :
           A ( "user" ) ;
           B ( "128.178.71.141", "\\Lamilspsrv1\Users\Periph\etudlsp40\CapProject\Src\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) ;
       } ;
```

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. In this example, two server threads execute on the client PC (master, named “user”) and two server threads on the server PC (slave, named with IP number `128.178.71.141`).

3. A first CAP program

Let us consider a simple program converting in parallel all lower-case letters to capitals. First, the client creates a string with lower-case letters "a" to "z". The string is segmented into individual characters by the split function *splitInput* and sent to server threads which run the leaf operation *myOperation*. This sequential procedure (leaf operation) converts lower-case letters to capital letters. These capital letters are merged into the final resulting string by the merge function *mergeOutput*. The main program *main()* running on the client thread displays the resulting string.

Please read and understand program *lowerToUpperCaseIP.pc* .

```
// lowerToUpperCaseIP.pc
// Simple didactic CAP Program with "indexed parallel" construct
// RDH 4.8.98

const int NB_OF_CHARS=26;                          // constant in the program
parameter int NumberOfProcesses = 4;               // number of threads

token aTozStringT {                                // token type for input and output string
  char aTozString[NB_OF_CHARS+1];                  // string embedded into token
  aTozStringT();                                   // constructor declaration
};

aTozStringT::aTozStringT()
{                                                  //constructor: build string abcd..z
  for (int i=0;i<NB_OF_CHARS;i++) {
    aTozString[i]= (char) (i+ ’a’);
  }
  aTozString[NB_OF_CHARS]= 0;                      // terminate with zero
}

token SingleCharT {
  int index;                                       // current character index
  char ThisChar;
  SingleCharT() {ThisChar=’$’;index=0;}            // inline constructor
};


void splitInput (aTozStringT* inputToken,
                 SingleCharT*& outputToken,
                 int splitIndex)                   // gives the ordinal of the current invocation
{
  outputToken = new SingleCharT;                   // creation of output token
  outputToken->index = splitIndex;
  outputToken->ThisChar= splitIndex + ’a’;         // current ASCII code
}
```

```
void mergeOutput(aTozStringT* outputResult,
                 SingleCharT* mergeInput,
                 int mergeIndex)                    // gives the ordinal of the current invocation

  // merge function is called as many times as split function was called
{ // sequential C++ body, outputResult token is created by CAP
  outputResult->aTozString[(mergeInput->index)] = mergeInput->ThisChar;
}

// every thread class (keyword process) needs to be declared

process MainProcessT {
operations:
};

process ComputeServerT {
operations:
  myOperation in SingleCharT* inputP out SingleCharT* outputP ;
};

leaf operation ComputeServerT::myOperation          // operation performed on server threads
  in SingleCharT* inputP
  out SingleCharT* outputP
{ // sequential C++ body, attention: outputP token needs to be created by programmer
  outputP = new SingleCharT;
  outputP->index = inputP->index;                   // copy index
  outputP->ThisChar = inputP->ThisChar - 0x20;      // capital letter: substract H'20
}

process ParallelServerT {                           // declaration of high-level thread structure
subprocesses :
MainProcessT Main;                                  // client thread
ComputeServerT ComputeServer[NumberOfProcesses];    // compute server threads
operations :
  ParallelComputation in aTozStringT* inputP
                      out aTozStringT* outputP ;
} ;

// The operation ParallelComputation contains a CAP parallel construct,
// for example the "indexed parallel" construct enabling to run the program
// on a given number of "real" ComputeServer threads,
// located in different address spaces, possibly on different computers (PC's).

// instantiate the high level process "ParallelServer"
// offering the high-level operation "ParallelComputation"

ParallelServerT ParallelServer;

operation ParallelServerT::ParallelComputation      // defines high-level parallel operation
in aTozStringT* inputP                              // this input is passed to splitInput
out aTozStringT* outputP                            // this output is obtained from mergeOutput
{ indexed
  (int i = 0 ; i < NB_OF_CHARS ; i++)
  parallel (splitInput, mergeOutput, Main, aTozStringT Result)
    (ComputeServer[i%NumberOfProcesses].myOperation)
  ;
  // the token from the split function is directed towards one of the 4 ComputeServer threads
}

// The high-level operation ParallelComputation may then be called
// from the main C++ program running in the client thread.

int main(int argc, char** argv)                     // main residing in client program
{
  aTozStringT* inputP= new aTozStringT;
  printf("input = %s \n", inputP->aTozString);
  aTozStringT* mainOutput;
  call ParallelServer.ParallelComputation in inputP out mainOutput;
  printf("output = %s \n", mainOutput->aTozString);
  delete mainOutput;                                // It is the programmers responsability to delete
                                                    // the parallel operation's output token
  return 0;
}
```

☞ ***Experiment 1***

Call an MS-DOS console. Place yourself in the directory XXXX. Execute macro *CapSetup*. Precompile your program by typing

cap.deb lowerToUpperCaseIP

If there are no errors, you may compile your program

comp.deb lowerToUpperCaseIP

If there are no errors, execute your program

lowerToUpperCaseIP

Without configuration file, all threads run in the same address space in your computer.

☞ ***Questions 1***

1.1. What is the content of the parallel program's input token *inputP*?

1.2. How many characters are generated by a single invocation of the split function *splitInput* ?

1.3. What is the procedure executed by the server threads and what does it do?

1.4. How does the merging function *mergeOutput* generate the output string?

1.5. How many times is the *splitInput* function invoked? Why?

1.6. How many times is the *mergeOutput* function invoked? Why?

1.7. The program incorporates 4 server threads. These threads are indexed by indices 0 to 3 (*ComputeServer[0]* to *ComputeServer[3]*). By which thread is character "f" converted to upper-case character "F" ?

4. Developing and debugging CAP programs

To develop, compile and debug CAP programs within the Msdev Visual C++ environment, apply the following steps

1. Make sure that all your source files are located in a subdirectory called *CapProj/Src*. Let us assume you have your source file *lowerToUpperCaseIP.pc* in the subdirectory called *CapProj/Src*. Normally, at CAP installation time, the variable *MSDEV_HOME_DIR* has been set to *C:\Application\Msdev* or to *C:\Program Files\DevStudio*, the variable *LSP_HOME_DIR* set to *z:\username\Cap0.1* or to *c:\CapUser* and the variable *UCAP* has been initialized to *z:\username\Cap0.1* or to *C:\CapUser.* You may verify that these variables exist by clicking in the *Control Panel* on *System* and then selecting the *Environment* window. If the variables are not set, run the script *CapSetup.bat* located in *C:\CapUser,* either by clicking or by running it from a console.
2. Run the macro "*MakeCapProject lowerToUpperCaseIP lowerToUpperCaseIP*"from a MS-DOS console in the *CapProject* directory a (the directory including the *Src* subdirectory). This macro generates a project *lowerToUpperCaseIP.dsp* . Click on that project and MS-Dev will create a workspace with the same name (*lowerToUpperCaseIP.dsw).* You may then look at your workspace (*View workspace*), open the CAP project and find your source file *lowerToUpperCaseIP.pc.*
3. You may precompile your program with *Build*, then *Compile* command. You may generate the executable file with the *Build* then *Build lowerToUpperCaseIP.exe* command. By clicking on the error messages, you directly access the corresponding erroneous source lines.
4. If your program needs an argument in the command line, you may select the project, *Settings*, and in the *Debug* field, enter in the *Program argument* field the argument (for example a filename or a number, etc).

5. To run the executable from Msdev, select *Build* and *Start Debug*. To insert breakpoints, place the cursor at the corresponding program line and type *F9*. You may remove the breakpoint by typing again at the same place *F9*. To execute and stop at breakpoints, select *Build*, respectively *Debug* and *Go.*

If it is not possible to place a breakpoint (error message from Msdev), just use *printf* to print at runtime the value of a variable.

☞ *Experiment 2*

2.1. Create a project *lowerToUpperCaseIP* incorporating file *lowerToUpperCaseIP.pc* (see above).

2.2. Precompile, compile and run the program *lowerToUpperCaseIP* from Msdev and verify that it executes correctly

2.3. Use the debugger to verify that the *splitInput* split function generates a correct *outputToken.* Verify in the debugging window, that *outputToken->ThisChar* takes successive values 'a', 'b', 'c',...

2.4. Use the debugger or a *printf* function call to verify that the merge function *mergeOutput* correctly translates lower-case characters to upper-case characters.

☞ *Programming exercise 1: local variables in server threads*

Each thread may incorporate variables, visible by its operations. The following program lines specify an integer variable in the ComputeServer threads:

```
process ComputeServerT {
variables:
  int passNumber;
operations:
  myOperation in SingleCharT* inputP out SingleCharT* outputP ;
};
```

Modify program *lowerToUpperCaseIP.pc* so as to convert in each server thread only in the first two passes characters from lower-case to upper-case. Name the resulting program *lowerToUpperCaseIPvar.pc* .

5. Token redirection by conditional expressions

One may want, at run time, to send or not send tokens to server threads depending on their values. With CAP, a conditional expression enables selecting which tokens are to be sent to given server threads. All tokens within a parallel expression are issued by a split function and are merged by a merge function. Therefore, the tokens which are not sent to server threads are directly merged into the resulting parallel operation output token.

5.1 The if expression

The following program part gives an example of the conditional execution of server thread operations. The keyword *thisTokenP* is always a pointer to the current token coming out from the split operation.

```
const int nbOfProcessedChars=8 ;

operation ParallelServerT::ParallelComputation
in aTozStringT* inputP
out aTozStringT* outputP
{
  indexed
  (int i = 0 ; i < NB_OF_CHARS ; i++)
  parallel (splitInput, mergeOutput, Main, aTozStringT Result)
    (if (thisTokenP->index <nbOfProcessedChars)
      (ComputeServer[i%NumberOfProcesses].myOperation)
    )
  ;
  // the token from the split function is directed towards one of the 4
  // ComputeServer threads
}
```

☞ *Experiment 3: conditional processing by server threads*

Insert the program piece shown above in the *lowerToUpperCaseIP.pc* program, compile it and execute it. Verify that conditional token routing works.

5.2 The ifelse expression

This expression enables redirecting a token to either one or the other compute thread operation. The following lines illustrate an example of the *ifelse* expression.

```
indexed  (int i = 0 ; i < NB_OF_CHARS ; i++)
parallel (splitInput, mergeOutput, Main, aTozStringT Result)
    (ifelse (thisTokenP->index %2 ==0)
      (ComputeServer[i%NumberOfProcesses].evenOperation)
      (ComputeServer[i%NumberOfProcesses+1].oddOperation)
    )
;
```

6. Pipelining multiple server leaf operations

It is often necessary to perform several leaf operations (sequential operations running in server threads) in a pipeline parallel manner. To continue with the present simple ASCII character conversion example, let us assume that we would like, in a second leaf operation, to shift the character's ASCII values by a given amount, for example by one.

The new operation *shiftChar* may be declared as a second operation offered by the *ComputeServerT* thread. Within the parallel program structure, the pipelining of leaf operations is shown by

```
ComputeServer[index1].myOperation >-> ComputeServer[index2].shiftChar
```

If *index1* is equal to *index2*, then both operations execute in the same thread, otherwise they will be executed in different threads.

Only the following lines of program *lowerToUpperCaseIP.pc* need to be changed or added in order to add the additional *shiftChar* operation in the pipeline.

```
process ComputeServerT {
operations:
  myOperation in SingleCharT* inputP out SingleCharT* outputP ;
  shiftChar in SingleCharT* inputP out SingleCharT* outputP;
};

leaf operation ComputeServerT::shiftChar
  in SingleCharT* inputP
  out SingleCharT* outputP
{ // sequential C++ body, attention: outputP token needs to be created by programmer
  outputP = new SingleCharT;
  outputP->index = inputP->index;                   // copy index
  outputP->ThisChar = inputP->ThisChar+1;           // shift by one character
}


operation ParallelServerT::ParallelComputation
in aTozStringT* inputP
out aTozStringT* outputP
{
  indexed
  (int i = 0 ; i < NB_OF_CHARS ; i++)
  parallel (splitInput, mergeOutput, Main, aTozStringT Result)
    (
      ComputeServer[i%NumberOfProcesses].myOperation >->
      ComputeServer[i%NumberOfProcesses].shiftChar
    )
  ;
}
```

☞ *Experiment 4: conditional processing by server threads*

Insert the program piece shown above in the *lowerToUpperCaseIP.pc* program, compile it and execute it. Verify that pipelining of operations works.

☞ *Questions 2*

2.1. Are the successive operations myOperation and shiftChar executed in different or in the same address space? Explain why!

2.2. If we would like, instead of shifting the character by one, to get the position of the capital letter within the set of capital letters {A,B,C,..,Z}, what program parts need to be changed, respectively added?

7. The for and while pipelined loop expressions

If several instances of the same operation, possibly running in different compute threads need to pipelined, i.e. one token has to repeat the same operation several times, it is possible to use a *for* or *while* CAP expression. The number of repetitions can be made data-dependent. The following program variant increases for each input token its ASCII value using the *shiftChar* operation until all ASCII values are equal or larger 'e'.

```
indexed
  (int i = 0 ; i < NB_OF_CHARS ; i++)
parallel (splitInput, mergeOutput, Main, aTozStringT Result)
    ( while (thisTokenP->ThisChar< 'e')            // first places in char array will be 'e'
      (ComputeServer[i%NumberOfProcesses].shiftChar)
    )
;
```

An example showing how to shift each character by 4 positions, using the *for* construct

```
indexed
  (int i = 0 ; i < NB_OF_CHARS ; i++)
parallel (splitInput, mergeOutput, Main, aTozStringT Result)
    ( for (int j=0;j<4;j++)                        // first places in char array will be 'e'
      (ComputeServer[i%NumberOfProcesses].shiftChar
    )
;
```

☞ *Experiment 5: repeated pipelined operations by server threads*

Insert the first or second program piece (while or for construct) shown above in the *lowerToUpperCaseIP.pc* program, compile it and execute it. Verify that repeated pipelined loops of operations work.

8. The parallel while constuct

The parallel while construct is used instead of the indexed parallel construct described in section 2 when an explicit index for terminating the parallel loop is not available. The termination of the parallel loop is data dependent, i.e. dependent on data available in the split function. Once the split function decides that the parallel while loop is to be terminated, no further tokens are sent to the parallel operations.

The split function must return 1 to be called again and return 0 during its last invocation. The syntax of the split function for the parallel while constuct is the following:

```
int splitInput (splitInputTokenType* inputToken,
                splitInputTokenType* previousToken,
                splitOutputTokenType* & outputToken,
               )
{ ..// sequential C++ body
    // attention: outputToken needs to be created

    // function returns 1 if split function is to be called again
    // function returns 0 if there is no further invocation
    // of the split function
}
```

The parallel while construct directs the token originating from the split function according to a user defined field (*index*) located in the token passed from the split function to the desired compute server thread. The syntax is the following, using the keywords *operation*, *in*, *out* and *parallel while* :

```
operation ParallelServerT::ParallelWhileComputation
  in splitInputTokenType* inputP
  out mergeOutputTokenType* outputP
{
  parallel while (splitInput, mergeOutput, Main, mergeOutputTokenType Result)
    (ComputeServer[thisTokenP->index].myOperation)
  ;
}
```

The syntax of the merge function remains identical to the merge function used in the indexed parallel construct.

As an example, the previous *lowerToUpperCaseIP* program may be modified as follows. The split function needs to be adapted and there is no index parameter in the merge function.

```
int splitInput (aTozStringT* inputToken,
                SingleCharT* previousToken,
                SingleCharT*& outputToken)
{
  outputToken = new SingleCharT;                    // creation of output token
  if (previousToken==0) { outputToken->ThisChar='a';}// first time in split function
  else {
       outputToken->index = previousToken->index + 1;
       outputToken->ThisChar=  previousToken->ThisChar + 1;
  }
  if (outputToken->index < NB_OF_CHARS-1) {return 1;}// means: call again split function
  else {return 0;}                                  // means: this was the last split function
                                                    // call
}

void mergeOutput (aTozStringT* outputResult,
                  SingleCharT* mergeInput)
{ // sequential C++ body, outputResult token is created by CAP
     outputResult->aTozString[(mergeInput->index)] = mergeInput->ThisChar;
}
```

The parallel operation construct is modified as follows:

```
operation ParallelServerT::ParallelComputation
in aTozStringT* inputP
out aTozStringT* outputP
{
  parallel while (splitInput, mergeOutput, Main, aTozStringT Result)
    (ComputeServer[thisTokenP->index%NumberOfProcesses].myOperation)
  ;
  // the token from the split function is directed towards the one of the 4
  // ComputeServer threads
}
```

☞ ***Experiment 6: using the parallel while construct***

Insert the program pieces shown above in the *lowerToUpperCaseIP.pc* program, compile it and execute it. Verify that the program with the parallel while construct executes correctly.

9. The Sieve of Eratosthenes: an example of "parallel while" program with many pipelined leaf operations

Let us now look at a real problem. Real problems have their own processing structure. We will see how to adapt the given problem structure to the available parallel CAP constructs.

The Sieve of Eratosthenes enables generating prim numbers in a pipeline of processing operations. A prim number generator generates successive numbers according to a given algorithm, for example 2,3,4,5,6.. These numbers pass through the processor pipeline. The first number arriving at a processor is stored as its prime number. The following numbers that pass through this processor are checked: if they are multiples of the stored prim number, they are discarded; if not, they are passed to the next processor in the pipeline. The set of prim numbers stored in the processors is collected and represents the resulting set of prim numbers.

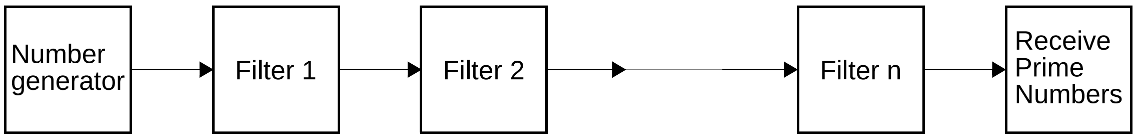

Fig. 4 Number generator and processor pipeline.

There may be various ways of generating a sequence of numbers incorporating all prim numbers up to a given value. For example, one may generate the numbers 2,3,5,7,9... by going, after number 3 (initialization) through all odd numbers, or one may, after initialization go through the numbers according to : 2,3,6-1,6+1,12-1,12+1,18-1,18+1, ..

From theory (there are slightly more than *x/lnx* prim numbers from 2 to *n*) we derive the size of an array able to store prim numbers up to a value called *maxPrimValue.*

nbMaxOfPrimes = 10 + maxPrimValue/4; // upper bound for storing prim numbers up to maxPrimValue

Let us now describe how to implement with CAP the Sieve of Eratosthenes.

The number generator can be part of the split function. It receives as input the *threshold*, i.e. the value up to which numbers need to be generated. The parallel construct will be of the type *parallel while* since the number generator does not necessarily generate the numbers one by one in increasing order.

The resulting prim number will be inserted by the merge function into an array. The size of the array is derived statically from the threshold, i.e. from the value of the highest prim value. We will construct a pipeline having the maximal required number of stages, i.e. as many pipelined leaf operations as there are places in the result array. Since, according to CAP's philosophy, all splitted values need to be merged, we have to create for each generated number two boolean variables specifying if it is a prim number and if it is a valid number. Valid numbers are numbers which may potentially be prim numbers. Non-valid numbers are numbers which have been detected to be non-prime. Therefore, the tokens generated by the split function and travelling through the set of pipelined *filterPrime* operations have the following structure

```
token SubInputOutputT {int nombre; bool prim; bool valid;};
```

The result array where all prim numbers have been merged is imbedded in a token (class) which includes an inline constructor. Tokens may include fixed size structures or arrays. For dynamic array structures, please refer to the CAP language reference manual (sect. 3. token serialization, sect 4.2: the predefined capArrayT class).

```
token ResultPrim {                          // result is an array of numbers
  int Index;
  int prims[nbMaxOfPrimes];
  ResultPrim ()                             // constructor
    {Index=0; for (int i=0;i<nbMaxOfPrimes;i++) prims[i]=0;
    }
};
```

Each number token generated by the split function travels through the successive *filterPrime* operations. The first valid token received by a filterPrime operation is its prim number (*ComputeServerT* process variable *myPrime*) . The next valid tokens are checked and marked as non-valid if they are divisible by *myPrime*. The *PipelinedPrime* parallel operation does not allow non-valid tokens to be forwarded to further *filterPrime* operations. The *for* pipelined loop ensures that tokens may travel up to the maximal number of required filterPrime operations.

```
operation ParallelServerT::PipelinedPrime
  in InputT* InputP
  out ResultPrim *result
{
  parallel while (SplitInput,MergeOutput, Main, ResultPrim result())
    (for (int i=0;i<(nbMaxOfPrimes-1);i++)
      (
       if (thisTokenP->valid) (sequential[i].filterPrime)
      )
    );
}
```

The full *primNumbers.pc* program is given below.

```
// Program primNumbers.cp :  parallel while program, with many pipelined leaf operations
// Generates numbers in a pipelined parallel manner
//   until a certain threshold is reached: each slave operation checks if the number
//   is divisible by the prim number it has previously stored
//   the sequence of generated numbers depends on the algorithm chosen:
//   (a) 2,3,4,5,6,... (b) 2, 3, -> 5, 7, 9, 11, 13, ..
//   (c) 2,3, -> 6-1, 6+1, 12-1,12+1, 18-1,18+1,..
// RDH 11.8.98,
#define true 1
#define false 0
#define bool int

const int maxPrimValue= 40;
const int nbMaxOfPrimes = 10 + maxPrimValue/4;      // upper bound for nb of prim numbers

token InputT {                                      // indicates threshold to reach with prim numbers
  int threshold;
  int currentNb;                                    // intermediate variable for split function
  InputT (int myThreshold)
    {threshold=myThreshold;currentNb=1;}            // threshold: maximal prim number value
 };

token ResultPrim {                                  // result is an array of numbers
  int Index;
  int prims[nbMaxOfPrimes];
  ResultPrim ()                                     // constructor
    {Index=0; for (int i=0;i<nbMaxOfPrimes;i++) prims[i]=0;}
};

token SubInputOutputT {int nombre; bool prim; bool valid;};

int SplitInput (InputT* inputP, SubInputOutputT* prevP, SubInputOutputT*& nombreOut)
{
  nombreOut = new SubInputOutputT;
  inputP->currentNb++;                              // input used in same way as a static variable
  nombreOut->nombre=inputP->currentNb;              // generates successive numbers: version (a)
  nombreOut->prim=false;
  nombreOut->valid=true;
  if (inputP->currentNb < inputP->threshold) return 1; else return 0;
}

void MergeOutput (ResultPrim* outputP, SubInputOutputT* subOutputP)
{
  printf("mergeIn=%d  ", subOutputP->nombre);
  if (subOutputP->prim==true)
    outputP->prims[outputP->Index++]= subOutputP->nombre;
}


process MainProcessT {
operations :
} ;

process ComputeServerT {
variables:                                  // variables residing in each computer server thread
  bool first;                               // first number received, not previously identified as prime
  int myPrime;                              // prim number associated to present thread
  ComputeServerT(){first=true; myPrime=-1;}  // constructor
operations :
  filterPrime in SubInputOutputT* subInput out SubInputOutputT* subOutput ;
} ;

process ParallelServerT {
subprocesses :
  MainProcessT Main ;
  ComputeServerT sequential[nbMaxOfPrimes-1] ;
operations :
    PipelinedPrime in InputT* InputP out ResultPrim* result ;
} ;


leaf operation ComputeServerT::filterPrime
  in SubInputOutputT* subInput
  out SubInputOutputT* subOutput
{
  subOutput = new SubInputOutputT;
  // the number received is prime: send it further (required to merge same number
  //                                  of tokens as tokens generated by the split fct)
  if (subInput->prim) *subOutput= *subInput;

  // this number was not previously identified as a prim number, if it is the first
  // number received not yet identified as prime, it becomes the prim number of the
  // present thread operations
  else if (first && subInput->valid ) {
    first=false;                                    // this->first=false
    myPrime= subInput->nombre;                      // this->myPrime
    subOutput->nombre=myPrime;
    subOutput->prim=true;                           // now, number is identified as prim
    subOutput->valid=true;
  }
  // not the first non-prim valid number received: if it is not a multiple of myPrime,
  // pass it further as a valid candidate, otherwise output it as an unvalid candidate
  else if (subInput->valid && ((subInput->nombre%myPrime)!= 0)) {
    subOutput->nombre=subInput->nombre;
    subOutput->prim=false;
    subOutput->valid=true;
  }
  else  {                                           // number is either not valid or it is not prime,
                                                    // mark it as false and pass a copy further down
    subOutput->nombre=subInput->nombre;
    subOutput->prim=false;
    subOutput->valid=false;
  }
}

operation ParallelServerT::PipelinedPrime
  in InputT* InputP
  out ResultPrim *result
{
  parallel while (SplitInput,MergeOutput, Main, ResultPrim result())
```

```
    (for (int i=0;i<(nbMaxOfPrimes-1);i++)
      (
       if (thisTokenP->valid) (sequential[i].filterPrime)
       )
    );
}

ParallelServerT ParallelServer; //instanciation of high-level parallel thread

int main (int argc, char** argv)
{
    int i;
    InputT* inputP = new InputT (maxPrimValue) ;
    ResultPrim* resultP ;
    call ParallelServer.PipelinedPrime in inputP out resultP ;
    for (i=0;i<nbMaxOfPrimes;i++)
      fprintf (stdout, "\n indexe = %i, nombre premier = %i\n", i, resultP->prims[i]) ;
 return 0 ;
}
```

☞ *Experiment 7*

Precompile, compile and run program *primNumbers.cp.*

Verify that it works.

☞ *Questions 3*

3.1. In operation filterPrime, we check if a number is valid. Is this check a necessary operation. Can you imagine simplifying this procedure. If yes, simplify it and run it again!

3.2. The prim numbers remain valid and travel through all stages of the pipeline. Can you think of a solution, where a prim number is directly merged into the resulting array? Program that solution and verify that it runs correctly.

10. Running the parallel program on multiple PC's

Before running a CAP program on multiple PC's, it is adviced to first run the CAP program as a multithread monoprocess program on a single PC. In this case, no configuration file is needed. Then the program should be executed and tested as a multithread program on multiple processes located on a single PC. Finally, the program may run on multiple processes located in multiple PC's.

To run the program on multiple NT processes, a configuration file is needed. As example consider the configuration file shown below. It specifies the present NT *processes* (for example A the master process and B the slave process), the IP number of the computer running the additional slave NT processes (for example 128.178.71.141), and the mapping of CAP threads (Sequential[0], .. Sequential[3]) to NT processes. The example below specifies two NT processes, a process located on the user's computer and a second process located on its own or a different PC (128.178.71.141). The string specifies the location and name of the CAP executable code on the PC running the second process. The mapping between CAP threads and NT processes is specified in the *threads* section. You can modify the relative load of the two processes by changing the mapping.

To create a new configuration file, you have to modify the example configuration file (for example *mandelbrot.cnf)* to give the IP numbers of the contributing slave PC's, the access path of the application and the mapping between threads and NT processes. To get the IP number of a PC, run in a console the command *ipconfig*.

```
//mandelbrot.cnf
configuration {
processes :
    A ( "user" ) ;
    B ( "128.178.71.141", "\\Lamilspsrv1\Users\Periph\etudlsp40\CapProject\Src\mandelbrotPar.exe" ) ;
threads :
    "Main" (A) ;
    "Sequential[0]" (A) ;
    "Sequential[1]" (A) ;
    "Sequential[2]" (B) ;
    "Sequential[3]" (B) ;
} ;
```

Before lauching the executable on several NT processes, the *Mpsd* message passing system must be launched with a parameter file containing the ports to be used.

To launch the MPSD on a single computer, the command, if executed locally is

```
Mpsd Local.conf
```

where the Local.conf file contains the following information:

```
MPSThreadPortNumber = 5567
MPSDaemonPortNumber = 5568
```

To launch the application on several PC's, in addition to launching the master's local MPSD (`Mpsd Local.conf`), the local command for each slave PC is:

```
Mpsd Remote.conf
```

where the Remote.conf file contains additional information specifying the IP number of the node starting the program (master) and its port number. Don't forget to modify an existing *Remote.conf* to include the IP number of the master computer of your application.

```
MPSThreadPortNumber = 5567
MPSDaemonPortNumber = 5568
MPSDaemon = 128.178.71.142 5568
```

Once that the message passing system is launched, the executable source file can be executed in the specified configuration by typing in a second MS-DOS Command window of the master PC

```
mandelbrotPar -cnf \\Lamilspsrv1\Users\Periph\etudlsp40\CapProject\Src\mandelbrot.cnf
```

with the full path of the configuration file so as to make it available on each contributing PC.

By changing the configuration file and without recompilation, you may try another mapping of threads to PC processes and obtain different performances. To obtain accurate program *execution times,* be careful to precompile and compile in release mode (in console):

```
cap.rel     mandelbrotPar
comp.rel    mandelbrotPar
```

In the case your progam is launched from Msdev, don't forget, by clicking on the project, in *Project settings* to specify in the *Debug* field the following additional argument

```
-cnf \\Lamilspsrv1\Users\Periph\etudlsp40\CapProject\Src\mandelbrot.cnf
```

Generally, once the message passing system is launched, many program executions can be made. If however, the Mpsd process needs to be killed, then, by typing *Ctl-Alt-Delete*, one may call the *Task Manager* and destroy the Mpsd process.

Special programs may be used (Exceed for example) to have a script lauching the message passing system on multiple PC’s from a single computer.

11. A real example: computation of the Mandelbrot set

The Mandelbrot set is a set of complex numbers $\{c \in C\}$, where after an infinite number of applications (in the program, *n* applications) of complex function $f_c(0) = z^2+c$, the resulting absolute value $|f_c^n(0)|$ is smaller than infinity. 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 1/200 and draw an image ranging from approximatively -2 to +2.

The *mandelbrotPar.pc* program uses a simple *parallel while* loop which distributes the image scanlines in round-robin manner to the compute server threads.

```
/* mandelbrotPar.pc
   simple parallel program without load balancing
   18/8/98 by SV
*/
include "iostream.h";                // for cout, no # as indication to CAP
#include "complex.h"                 // complex number class, see Appendix 2
#include "ImageFile.h"               // operations on image files: ImageFile class, see Appendix 3

parameter (char * outputFilename = "mandelbrot.ppm", ShortHand -> "-ofn");


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                    = 0.005;
const int    MAX_ITERATIONS         = 200;
const double MAX_MAGNITUDE          = 200.0;

static ImageFile imageFile(outputFilename,IMAGE_SIZE_X,IMAGE_SIZE_Y,1);


token TileDescriptionT {
  int lineIndex;                                    // line index
  TileDescriptionT (int Index) {                    // inline constructor
    lineIndex = Index;
  }
};

token TileT {
  int lineIndex;                                    // line index of computed line
  unsigned char buffer[IMAGE_SIZE_X];               // line buffer
  TileT (int index){
    lineIndex = index ;                             // inline constructor
  }
};

token StartT {};                                    // empty token



token ImageT {
  unsigned char buffer[IMAGE_SIZE_X*IMAGE_SIZE_Y];  //image buffer as output token
};
```

```
// Mandelbrot function renders magnitude after a number of iterations
// if magnitude very small, x,y value belongs to Mandelbrot set
unsigned char MandelbrotFunction(int x, int y){
  double m;
  complex c(X_START + x*GAP, Y_START + y*GAP);
  complex z(0.0,0.0);
  int k = 0;
  while (z.Magnitude() < MAX_MAGNITUDE && k < MAX_ITERATIONS) {
    z = z * z;
    z = z + c;
    k++;
  }
  m = z.Magnitude();
  if (m>2550) m = 2550;
  return (unsigned char)(m/10);
}

process ComputeServerT {
operations :
  ComputeMandelbrot in TileDescriptionT* InputP out TileT* OutputP;
};

int SplitFunction (StartT* inputP,
  TileDescriptionT* previousP,
  TileDescriptionT*& nextP)
{
  cout << "S"<< flush;
  int nextIndex = 0;
  if (previousP!=0)
  nextIndex = previousP->lineIndex + 1;              // scanline index
  nextP = new TileDescriptionT(nextIndex);           // allocates new scanline
  if (nextIndex == IMAGE_SIZE_Y-1) return 0;         // if last scanline, returns 0
  else return 1;                                     // else continue calling splitfct
}

leaf operation ComputeServerT::ComputeMandelbrot
  in TileDescriptionT* InputP
  out TileT* OutputP
{
  OutputP = new TileT(InputP->lineIndex);            // allocates output token
  cout << "." << flush;
  for (int i=0; i< IMAGE_SIZE_X; i++)                // magn values of each pixel of scanline
    OutputP->buffer[i] = MandelbrotFunction(i,InputP->lineIndex);
}

void MergeFunction (ImageT* intoP, TileT* inputP)
{
  cout << "M"<< flush;
  CopyMemory(&intoP->buffer[inputP->lineIndex*IMAGE_SIZE_X],
                            &inputP->buffer, IMAGE_SIZE_X);
}

process MainProcessT {
operations :
} ;

process ParallelServerT {
subprocesses :
  MainProcessT Main ;
  ComputeServerT Sequential[NUMBER_OF_COMPUTE_SERVERS] ;
operations :
  GlobalOperation in StartT* InputP out ImageT* OutputP ;
} ;


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

```
ParallelServerT ParallelServer ;

int main (int argc, char** argv)
{
  cout << "Mandelbrot set generation program \n";
  cout << "S - Split function\n";
  cout << ". - Operation\n";
  cout << "M - Merge function\n\n";
  long StartupTime = GetTickCount();
  long EndComputingTime, EndFileWritingTime;

  StartT* InputP;
  ImageT* OutputP;
  InputP  = new StartT() ;
  call ParallelServer.GlobalOperation in InputP out OutputP ;

  EndComputingTime = GetTickCount();
  imageFile.SaveImage((unsigned char *)&(OutputP->buffer[0]));
  cout << '\n'<< flush;
  delete OutputP;
  EndFileWritingTime = GetTickCount();

  cout << '\n'<< flush;
  cout << "Computing time [ms]: "<< EndComputingTime-StartupTime << '\n';
  cout << "File writing time [ms] : "<< EndFileWritingTime-EndComputingTime << '\n';
  cout << "Total time   [ms] : "<< EndFileWritingTime-StartupTime << '\n'<<flush;
  return 0 ;
}
```

☞ *Experiment 8*

8.1. Precompile, compile and run program *mandelbrotPar.cp*. The program generates a **.ppm* or **.pgm* file that can be observed by the application *viewppm.exe* (*viewppm mandelbrot.ppm).*

Verify that the program works on a single PC, note the execution times and examine the produced image file.

8.2. Create, according to the instructions in section 9, a configuration file in order to run the program on two different processes of the same computer. Lauch the MPSD message passing system and run the program without recompiling it. Note the execution times.

8.3. Modify the configuration file to run the program on two computers. Launch the MPSD message passing system on both computers and run the program. Note the execution times.

You have to modify the configuration file (for example *mandelbrot.cnf)* to give the IP numbers of contributing slave PC's, the access path of the application and the mapping between threads and NT processes. To get the IP number of a PC, run in a console the command *ipconfig*.

```
configuration {
processes :
    A ( "user" ) ;
   B ( "128.178.71.141", "\\Lamilspsrv1\Users\Periph\etudlsp40\CapProject\Src\mandelbrotPar.exe" ) ;
threads :
    "Main" (A) ;
    "Sequential[0]" (A) ;
    "Sequential[1]" (A) ;
    "Sequential[2]" (B) ;
    "Sequential[3]" (B) ;
} ;
```

Once your configuration file is ready, prepare the *remote.conf* file for running the Mpsd on the remote PC. Work with a colleague and create in his directory the *remote.conf* file you need, by putting his IP number in that file:

```
MPSThreadPortNumber = 5567
MPSDaemonPortNumber = 5568
MPSDaemon = 128.178.71.142 5568
```

You should then first run the *Mpsd local.conf* on your PC and then on your colleague’s PC the *Mpsd remote.conf* command. In a second console of your PC, you may then run your parallel application, for example

```
mandelbrotPar -cnf \\Lamilspsrv1\Users\Periph\etudlsp40\Src\mandelbrot.cnf
```

with the full path of the configuration file so as to make it available on each contributing PC.

By changing the configuration file and without recompilation, you may try another mapping of threads to PC processes and obtain different performances. To obtain accurate program *execution times*, be careful to precompile and compile in release mode (in console):

```
cap.rel  mandelbrotPar
comp.rel mandelbrotPar
```

☞ ***Questions 4***

4.1. Does the parallel program ensure that when running on two compute nodes, the two PC's are always busy? Click on the performance monitor and verify how busy the two computers are during computation.

4.2. What speedup do you get when running the program on two PC's?

12. Flow-control and load-balancing issues

In the current CAP implementation, the split function generally 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 disks).

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.

For the purpose of flow-control and load balancing, the CAP preprocessor translates an *indexed parallel* loop or respectively a *parallel while* loop into a combination of *indexed parallel* or respectively *parallel while* and a *for* CAP construct:

```
indexed
  (int index=0; index<indexMax; index++)
parallel (splitfct,mergefct, Main, OutT outP)
  (ComputeServer.operation)         // ComputeServer is a compute server process
;                                   // Main is the starting main process
```

becomes a construct of the type

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

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.

To ensure flow-control, the user specifies by the instruction *flow_control(20)* the number of tokens in circulation, for example 20. This slight modification is in front of the standard *indexed parallel* or *parallel while* construct:

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

The load-balancing mechanism uses the same construct: tokens recirculate according to a *for* loop along the branch of 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 into 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 operations.

An example of a construct enabling flow-control and load-balancing is the following:

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

To ensure both flow control and load balancing, the *mandelbrotPar.pc* program needs to be modified so to incorporate these features into its main *parallel while* construct.

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

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

☞ *Experiment 9*

Modify the *mandelbrotPar.pc* program for flow-control and load-balancing. For verifying the effect of load balancing, create a configuration file *mandelbrot.cnf* with one server (slave) thread running on a separate PC and all other server thread running on the user PC, similar to

```
configuration {
processes :
 A ( "user" ) ;
 B ( "128.178.75.10",     "\\lsppc20\lsppc20d\users\hersch\cap\crsJul\mandelbrotPar.exe" ) ;
threads :
   "Main" (A) ;
   "Sequential[0]" (A) ;
   "Sequential[1]" (A) ;
   "Sequential[2]" (A) ;
   "Sequential[3]" (B) ;
} ;
```

Create also a second configuration file where 2 server threads run on the user PC and the 2 other server threads run on the second PC whose IP number is given explicitely.

Run the modified program *mandelbrotParFC.pc* successively on a single PC, on two NT processes of a single PC and on two PC's, one time with only one server thread on the second PC and the second time with two server threads on the second PC. Note the execution times.

☞ ***Questions 5***

Compute the speedup for all variants. Is the speedup similar for the two parallel variants?

Questions 6

Propose a strategy to verify that flow-control and load balancing effectively brings an improvement ! What measurements have to be done?

13. Combining pipelined parallel computations and file accesses

CAP's enables the pipelined parallel execution of operations. If one includes into these operations a file system call (reading a file part, writing a file part), it is possible to read and write files or file parts in parallel. If one includes a file access call in a split or merge, this file access may execute in pipeline with operations of the parallel split-merge construct.

Let us consider as example the mandelbrotPar.pc example presented in section 11. After generating in parallel for each pixel of the output image a magnitude value, the full pixmap containing all pixel values is written as a separate step to an image file. The file writing time can be completely hidden behind the image computation time, if the file writing operation is integrated into the parallel construct, i.e. if each scanline that arrives from a *ComputeMandelbrot* operation is directly merged into the final image file by writing it at its required position within the file.

A class ImageFile offers in addition to file opening and close functions a function

```
inline void ImageFile::SaveDataChunk(int position, int size, unsigned char * chunk){
  fseek(file,position,SEEK_CUR);
  fwrite(chunk,sizeof(unsigned char),size,file);
  fseek(file,-(position+size),SEEK_CUR);
}
```

This function can be used to save a scanline at its correct position within the image file. The merge function will therefore be changed to:

```
void MergeScanlineToFile (ImageT* intoP, TileT* inputP)
{
  cout << "M"<< flush;
  imageFile.SaveDataChunk(inputP->lineIndex*IMAGE_SIZE_X,IMAGE_SIZE_X,
                          &inputP->buffer[0]);
}
```

By having in the main program the instructions to create the image file and to generate its header, in the parallel GlobalOperation the merge function as described above and the instruction to close the file, file writing is completely pipelined with the processing operations.

```
imageFile.Open();                                            // create the image file
imageFile.SaveHeader();                                      // generate its header
call ParallelServer.GlobalOperation in InputP out OutputP ; // parallel operations
imageFile.Close();                                           // close the file
```

☞ ***Experiment 10***

Modify the *mandelbrotParFC.pc* in order to integrate file writing into the pipeline. Run the modified program *mandelbrotParFCWr.pc* successively on a single PC, on two NT processes of a single PC and on two PC's. Note the execution times.

☞ *Questions 7*

7.1. Evaluate the effective disk throughput obtained when writing the image buffer (512x512 bytes) to a file with the initial program *mandelbrotPar.pc* ? Taking into account that the initial seek operation takes 10ms and that a throughput of approximatively 4 MB/s can be achieved when writing a continuous file into a contiguous disk space, is the file writing time obtained realistic? What effects can lead to wrong conclusions?

7.2. Are the file writing times completely hidden in the *mandelbrotParFCWr.pc* program version? For this purpose, compare the execution times without any file writing and with file writing.

14. Suspending and reordering the flow of tokens

There are cases, where it is necessary to suspend momentarily the flow of tokens. This can be done by calling the predefined function

```
void capDoNotCallTokenSuccessor(myTokenT* OutputP);
```

This call inhibits sending at the end of the operation the Output token to the next operation. To resume the flow of tokens the operation

```
void capCallSuccessor(myTokenT* myTokenP)
```

is executed, where myTokenP is a pointer to the token that has been prevented from being sent out. Generally, the pointers to such inhibited tokens are stored in an array of token pointers.

Suspending the flow of tokens is useful to launch an asynchronous operation, for example a file access and to resume the flow of tokens by making the *capCallSuccessor* call for in the in the file access callback function. In continuous media applications, tokens may have to be sent periodicly. A *capCallSuccessor* call may for example be made as a result of a call back function made by a timer event.

To reorder the flow of tokens, it may be necesssary to suspend it momentarily. The flow of tokens may go out of order due to uneven processor load. Consider for example the *mandelbrotParFCWr.pc* program, which writes successive scanlines to the disk. If the amount of the data to be written to disk would be important, disk access times would become predominant. To reduce disk access times, one would like to eliminate seek function calls. This can be done by reordering the tokens in front of the merge function and by writing the scanlines continuously one after the other onto the disk.

For that purpose, the following modifications have to be made the *mandelbrotParFCWr.pc* program. The main process now includes a current line index indicated the next line to be sent to the merge function. It further includes a table (*tokenPArr*) for storing token pointers which have been prevented from being sent out by the *capDoNotCallTokenSuccessor* function. It also includes the new *ReorderTokens* operation.

```
process MainProcessT {
variables:
  int currLineIndex;
  TileT* tokenPArr[IMAGE_SIZE_Y];
  MainProcessT() {currLineIndex=0;
    for (int i=0;i<IMAGE_SIZE_Y;i++) tokenPArr[i]=0;
  } // constructor
operations :
  ReorderTokens in TileT* InputP out TileT* OutputP;
} ;
```

The operation *ParallelServerT::GlobalOperation* now also includes after the *ComputeMandelbrot* operation executed in the *Sequential* server processors the operation *ReorderTokens* executed in the *Main* thread.

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

{ flow_control(20)
  parallel while (SplitFunction, MergeScanlineToFile, Main, ImageT Result()) (
    Sequential[cap_fcindex0%NUMBER_OF_COMPUTE_SERVERS].ComputeMandelbrot >->
    Main.ReorderTokens
  ) ;
}
```

The *ReorderTokens* operation compares the arriving line with the current line index. If it is higher, its further circulation is inhibited and it is stored in the *tokenArr* array. If it is equal, it is also stored in the array, and together with possibly previously stored line tokens, they are extracted in order from the array and put into circulation.

```
leaf operation MainProcessT::ReorderTokens
  in TileT* InputP
  out TileT* OutputP
{
  if (InputP->lineIndex > currLineIndex) {                  // prepare successor in table
    OutputP = new TileT(InputP->lineIndex);                 // allocate output token
    tokenPArr[InputP->lineIndex]=OutputP;                   // queue in table
    for (int i=0; i< IMAGE_SIZE_X; i++)
      OutputP->buffer[i] = InputP->buffer[i];               // copy content
    capDoNotCallTokenSuccessor(OutputP);                    // (TileT*);
  }
  else {
        //InputP->lineIndex <= currLineIndex
    OutputP = new TileT(InputP->lineIndex);
    tokenPArr[InputP->lineIndex]=OutputP;                   // puts in array to be outputted
    cout << "," << flush;
    // copies the buffer of the input into output token
    for (int i=0; i< IMAGE_SIZE_X; i++)                     // copy buffer
      OutputP->buffer[i] = InputP->buffer[i];
    capDoNotCallTokenSuccessor(OutputP);                    // don't output this token at the end
      // of the procedure, output it in sequence from the tokenPArr array

    for (int j=0;(j<=currLineIndex) && (j<IMAGE_SIZE_Y);j++) {
      if (tokenPArr[j]!=0)  {
        capCallSuccessor(tokenPArr[j]);                     // output of token
        tokenPArr[j]=0;
        printf(" successor= %i", j);
        currLineIndex++;
      }
    }
  }
}
```

To make possible errors visible, the merge function calls the *ImageFile::SaveLineContiguous* function to store each scanline contiguously after the previous scanline in the image file.

```
void MergeScanlineToFile (ImageT* intoP, TileT* inputP)
{
  printf(" M= %i ", inputP->lineIndex);
  imageFile.SaveLineContiguous(IMAGE_SIZE_X, &inputP->buffer[0]);
}
```

☞ *Experiment 11*

Apply the above mentioned modifications to program *mandelbrotParFCWr.pc* and generate program *mandelbrotParFCWrCS.pc* . Precompile, compile and execute program *mandelbrotParFCWrCS.pc* . Verify the sequence of generated line tokens and the sequence of line tokens after the reordering function. Visualize the resulting mandelbrot.ppm file and check that the image is ok.

☞ *Questions 8*

8.1. In the case, a line token has the same line index as the current index (*currLineIndex*), why is it first necessary to store in it the *tokenPArr* array and then to send all tokens from this array having a line index smaller or equal to *currLineIndex* ?

8.2. Why is it necessary to run the *ReorderTokens* operation as part of the *MainProcessT* process structure? Could the *ReorderTokens* operation run in a Sequential server process?

15. Neighbourhood-dependent parallel operations (example: game of Life)

In the preceeding 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, each processing element (thread) must, at a certain point of its program execution receive information from neighbouring processing elements (threads). As an example, we consider parallelizing the game of Life.

This game was introduced in 1970 by the mathematician John Conway. The rules of the game are as follows:

1. The world is represented by an array of cells (TOT_VERT_SIZE x HOR_SIZE)
2. Every cell is always either dead or alive
3. 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, othewise it dies. A dead cell with 3 neighbors becomes alive, otherwise it remains dead.
4. All state transitions over the entire world are synchronized.
5. The world wraps around: all vertical neighbors are calculated modulo TOT_VERT_SIZE and all horizontal neighbors are calculated modulo HOR_SIZE.

Here follows a sequence of evolution at times t0 (initial state), t1, t2, t3:

```
t0      t1      t2      t3
..O..   ..OO.   .O.O.   .....
..OO.   .OOO.   .....   .....
..O..   .OO..   .O.O.   .....
..O..   .....   .....   .....
.....   .....   .....   .....
```

To parallelize the game of life, we segment the world into horizontal tiles of size TOT_VERT_SIZE/NUMBER_OF_COMPUTE_SERVERS x HOR_SIZE. 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.

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 a token comprising the world is sent to each server thread (split-merge construct).
2. Each parallel iteration, i.e. each computation of the new state of the worlds tile includes two steps: (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) and (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.

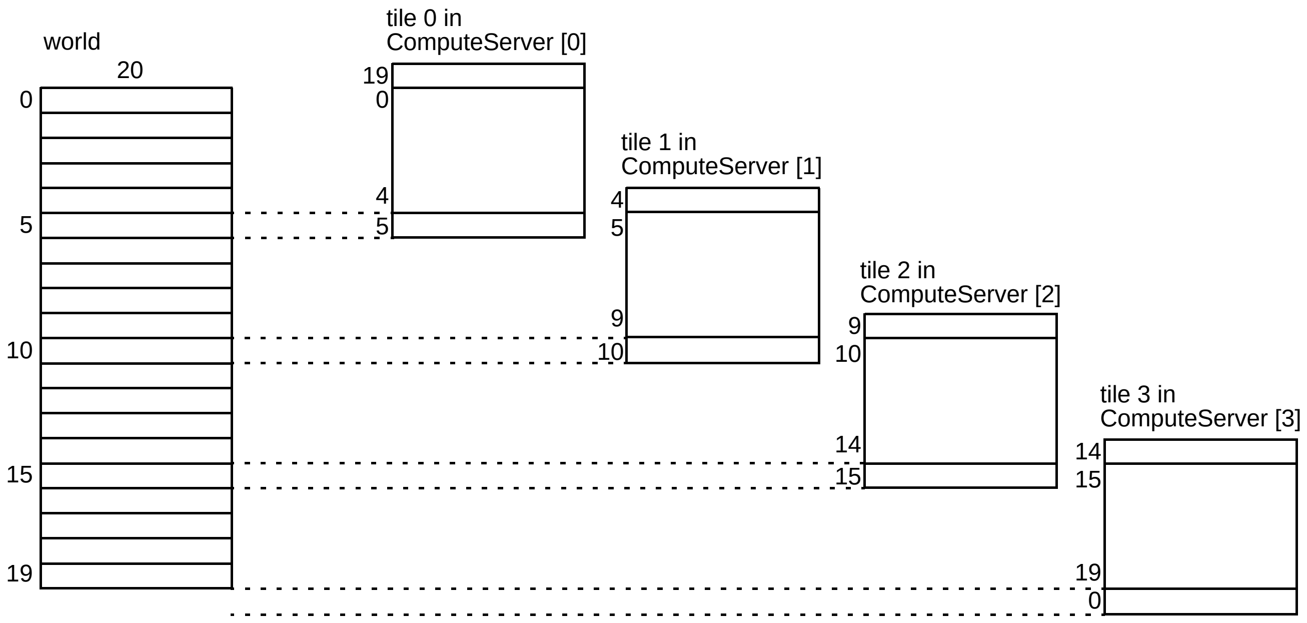

Fig. 5 The world, the tiles and the array located in server threads.

15.1 The parallel construct

In addition to the *indexed parallel* and to the *parallel while* construct, there is a *parallel* construct, again based on the *split* - *operation* - *merge* concept, where each parallel branch is separatly specified and may contain its own split function, its own operation and its own merge function (for more information, see Reference Manual, section 2.5.6). The syntax of the parallel operation is the following:

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

Main indicates the thread where the merging functions are executed and the result token is *Result* of type *OutputT*. The present parallel construct comprises four branches, each with its own split, operation and merge functions. One may however also use the same operations in different branches.

15.2 The parallel game of life

The main program comprises three calls to parallel CAP constucts. 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;
```

The *parallelInitPartWorld* high-level parallel operation is an indexed parallel construct, which circulates a token of type *WorldT*.

```
operation ParallelServerT::parallelInitPartWorld
  in WorldT* InputP
  out WorldT* OutputP
{ indexed
  (int i = 0 ; i < NUMBER_OF_COMPUTE_SERVERS ; i++)
  parallel ( SplitWorld, MergeWorld, Main,
             WorldT Result (thisTokenP->vertSize,thisTokenP->horSize))
             ( Sequential[i].initPartWorld(i)
  );
}
```

The *Automaton* high-level parallel 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 (life of game iteration).

```
operation ParallelServerT::Automaton(int nbIterations)
// on Main, sequences the indexed parallel ExchangeBorder and the following
// indexed parallel ComputeStep
  in void* InputP
  out void* OutputP
{
  // single iteration first
  for (int it=0;it<nbIterations;it++) (             // iterations ExchangeBorders->ComputeStep
    indexed                                         // to synchronize exchange of borders
      (int i=0; i<NUMBER_OF_COMPUTE_SERVERS; i++)
    parallel (void,void,Main,void output)
      ( ExchangeBorders (i) )
        >->
    indexed
      (int j=0; j<NUMBER_OF_COMPUTE_SERVERS; j++)
    parallel (void,void,Main,void output)
      ( Sequential[j].ComputeStep)
  ) ;
}
```

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 (*prevworld*). The merge of the parallel operation is executed in thread *Sequential[partIx]* and the operation itself is executed in another thread for example in *Sequential[partIx-1]*, *Sequential[partIx+1]*).

```
operation ParallelServerT::ExchangeBorders (int partIx)
  // this operation asks in parallel the neighbours to send their borders,
  //  merges them into the prevworld
  // the parallel construct is lauched in the Main thread;
  // where Sequential is called, it acts on the slave threads
  // exchange of borders works in wraparound mode
  in void* InputP
  out void* OutputP
{
  parallel (Sequential[partIx], void result) (  // indicates merge in Sequential
    (void
    , ifelse (partIx>0)
      (Sequential[partIx-1].sendBottomBorder)                    // partIx>0
      (Sequential[NUMBER_OF_COMPUTE_SERVERS-1].sendBottomBorder) // partIx==0
    , mergeBorders(TopBorder)
    )
    (void
    , ifelse (partIx<NUMBER_OF_COMPUTE_SERVERS-1) // partIx<NUMBER_OF_COMPUTE_SERVERS-1
      (Sequential[partIx+1].sendTopBorder) // partIx==NUMBER_OF_COMPUTE_SERVERS-1
      (Sequential[0].sendTopBorder)
    , mergeBorders(BottomBorder)
    )
  ) ;
}
```

The full *lifePar.pc* program is listed in Annex I. The presented version only provides support for a fixed sized world, whose size is determined at compile time. This is due to the fact that only fixed sized arrays and structures can be declared in tokens. However, the predefined *capArrayT* class supported by the CAP compiler provides support for dynamic arrays and for packing, respectively unpacking these dynamic arrays into tokens (see Reference Manual, setion 4.2).

☞ *Experiment 12*

Precompile, compile and execute the *lifePar.pc* program on one PC. Try to understand the execution of the program, by analyzing the printed comments and the printed values of the world or of part of it.

☞ *Questions 9*

9.1. (a) Is the initialization of the tiles (*prevworld*) in the parallel threads done in parallel or in a pipelined sequence? (b) Could it be programmed as a pipelined sequence? (c) What would need to be changed to have the initialization as a pipelined sequence?

9.2. (a) How are border lines transferred from one thread to the neighbouring threads? (b) Does a border line travel first to the main thread and then back to a neighbouring thread?

(c) Are border lines exchanged in parallel or as sequence of pipelined operations?

9.3. What makes sure that before a new state of the world is computed, neighbouring tiles are really available in the parallel threads operating on their local tile (*prevworld*)?

9.4. A first version of the program, called *lifeParWrong.pc* had a slightly different structure. Instead of two separate index parallel structures, one for exchanging tile borders and one applying one iteration, the two are unified as a sequence of operations as shown in operation *AutomatonStep*. Why does program lifeParWrong not work correctly?

```
operation ParallelServerT::AutomatonStep (int partIx)
// this operation asks in parallel the neighbours to send their borders,
// merges them into the prevworld
// the parallel construct is lauched in the Main thread;
// where Sequential is called, it acts on the slave threads
// exchange of borders works in wraparound mode

  in void* InputP
  out void* OutputP
{
  parallel (Sequential[partIx], void result) (          // indicates merge in Sequential
    (void                                               // empty split function
    , ifelse (partIx>0)
      (Sequential[partIx-1].sendBottomBorder)                          // partIx>0
      (Sequential[NUMBER_OF_COMPUTE_SERVERS-1].sendBottomBorder)       // partIx==0
    , mergeBorders(TopBorder)
    )
    (void
    , ifelse (partIx<NUMBER_OF_COMPUTE_SERVERS-1)       // partIx<NUMBER_OF_COMPUTE_SERVERS-1
      (Sequential[partIx+1].sendTopBorder)              // partIx==NUMBER_OF_COMPUTE_SERVERS-1
      (Sequential[0].sendTopBorder)
    , mergeBorders(BottomBorder)
    )
  ) >->
  Sequential[partIx].ComputeStep ;
}
```

```
operation ParallelServerT::Automaton(int nbIterations)
  in void* InputP
  out void* OutputP
{
  // single iteration first
  for (int it=0;it<nbIterations;it++) // iterations of  AutomatonStep: exchange->Compute
    indexed
      (int i=0; i<NUMBER_OF_COMPUTE_SERVERS; i++)
    parallel (void,void,Main,void output)
      ( AutomatonStep(i) )
  )
  ;
}
```

9.5. Does the *lifePar.pc* program support the overlap of computation (next step of the world) and the exchange of borders (communication)? If not, what would needed to be changed to ensure such an overlap ?

Appendix 1

```
// program lifePar.pc
// 1. read the input file (2D world), transfer it to the computer servers
//    and initialize the local 2D tiles in parallel
// 2. launch the computations
// 3. get the results from the parallel threads and display them
// 26.8.98, RDH

const int NUMBER_OF_COMPUTE_SERVERS =  4;
const int HOR_SIZE = 20;
const int TOT_VERT_SIZE = 20;
const int PART_VERT_SIZE = TOT_VERT_SIZE/NUMBER_OF_COMPUTE_SERVERS;
const int ALIVE = 1;
const int DEAD = 0;

int NB_ITERATIONS;

// data structures described by tokens & initialization of subtiles


token WorldT {                             // this is the full world, also used to store hor. parts
  int firstRow ;                                  // first valid row
  int horSize;
  int vertSize;                            // effective number of stored rows
  int world[TOT_VERT_SIZE][HOR_SIZE] ;
  WorldT (int vertsize,int horsize) ;
} ;


WorldT::WorldT (int vertsize, int horsize)   // constructor
{
  firstRow = 0 ;                           // first valid Row in array
  horSize = horsize ;
  vertSize = vertsize;
  for (int i=0;i<vertSize;i++)
    for (int j=0;j<horSize;j++)
      world[i][j]=0;
} ;

enum NeighbourT { BottomBorder, TopBorder, None };

void printWorld(WorldT* worldP)
{
  int i,j ;
  for (i = 0 ; i < worldP->vertSize ; i++)
  {
    if (((i%(TOT_VERT_SIZE/NUMBER_OF_COMPUTE_SERVERS))==0)&&(i!=0)) printf("\n");
    printf("\n");
    for (j = 0 ; j < worldP->horSize ; j++)
      printf(" %i", worldP->world[i][j]);
  }
  printf("\n");
}


void printArray(int array[][HOR_SIZE], int vertSize, int horSize)
{
  int i,j ;
  for (i = 0 ; i < vertSize ; i++)
  {
    printf("\n");
    for (j = 0 ; j < horSize ; j++)
    {
      printf(" %i", array[i][j]);
    }
  }
  printf("\n");
}

void SplitWorld
  ( WorldT* worldP
  , WorldT*& partworldP
  , int index
  )
{
  int result ;
  int partvertsize = TOT_VERT_SIZE/NUMBER_OF_COMPUTE_SERVERS ;
  int firstrow = index * partvertsize ;
  partworldP = new WorldT (partvertsize,worldP->horSize) ;
  partworldP->firstRow = firstrow ;

  for (int i=firstrow;i<(firstrow+partvertsize);i++)
    for (int j=0;j<partworldP->horSize;j++)
      partworldP->world[i][j]=worldP->world[i][j] ;
  printf("split: firstrow= %i \n", firstrow);
}

void MergeWorld (WorldT* resworldP , WorldT* partworldP, int index)
{ // every cell of partworld must be copied into corrrect hor tile of resworld,
  // for verification purposes
  for (int i=partworldP->firstRow;i<((partworldP->firstRow)+(partworldP->vertSize));i++)
  for (int j=0;j<partworldP->horSize;j++)
    resworldP->world[i][j] = partworldP->world[i][j];
}

process MainProcessT {
operations :
} ;

process ComputeServerT {
variables :
  int prevworld[PART_VERT_SIZE+2][HOR_SIZE];
  int currworld[PART_VERT_SIZE+2][HOR_SIZE];
  int HorSize;
  int VertSize;               // prevworld & currworld have  size VertSize+2 (with neighbours)
  int localFirstRow;
  void mergeBorders(void* resultP, WorldT* borderP, NeighbourT neighbour);
operations :
  initPartWorld(int threadIx) in WorldT* InputP out WorldT* OutputP;
  sendTopBorder in void* InputP out WorldT* OutputP;
  sendBottomBorder in void* InputP out WorldT* OutputP;
  ComputeStep in void* InputP out void* OutputP;
  GetSubWorld (int threadIx) in void* InputP out WorldT* OutputP;
} ;

process ParallelServerT {
subprocesses :
  MainProcessT Main ;
  ComputeServerT Sequential[NUMBER_OF_COMPUTE_SERVERS] ;
operations :
  parallelInitPartWorld in WorldT* InputP out WorldT* OutputP ;
  Automaton(int nbIterations) in void* InputP out void* OutputP;
  ExchangeBorders (int partIx) in void* InputP out void* OutputP;
  GetWorld in void* InputP out WorldT* lastResultP;
} ;


leaf operation ComputeServerT::initPartWorld(int threadIx)

// here, the initial segmented world is copied into the local buffer prevworld
// first and last raws are reserved for overlaps, but not initialized
// prevworld,currworld,VertSize,HorSize,localFirstRow are local process variables

  in WorldT* InputP
  out WorldT* OutputP
{
  localFirstRow = InputP->firstRow ;
  VertSize=InputP->vertSize; HorSize=InputP->horSize;

  OutputP = new WorldT(InputP->vertSize,InputP->horSize);
  OutputP->firstRow = localFirstRow ;        // same first row in output token

  for (int ii = 0 ; ii < InputP->vertSize; ii++)
    for (int jj=0; jj < InputP->horSize; jj++)
    {
      OutputP->world[localFirstRow+ii][jj] = InputP->world[localFirstRow+ii][jj];    // for test
      // row 0 is kept as overlap from neighbour
         prevworld[ii+1][jj] = InputP->world[localFirstRow+ii][jj];
    }
  printf ("process %s prevworld init \n, ", processP->cap_NameS) ;
}

leaf operation ComputeServerT::sendTopBorder
  in void* InputP
  out WorldT* OutputP
{
  OutputP = new WorldT(1,HorSize);
  OutputP->firstRow=0;                          //first row of array used to transfer border
  for (int i=0;i<HorSize;i++)
    OutputP->world[0][i]=prevworld[1][i];       // send line 1 from prevworld
}

leaf operation ComputeServerT::sendBottomBorder
  in void* InputP
  out WorldT*  OutputP
{
  OutputP = new WorldT(1,HorSize);
  OutputP->firstRow=0;
  for (int i=0;i<HorSize;i++)
  {
  // send last valid line from prevworld
     OutputP->world[0][i]=prevworld[VertSize][i];
  }
}

void ComputeServerT::mergeBorders( void* resultP, WorldT* borderP, NeighbourT myBorder)
{
  if (borderP->cap_TypeIndexF()==capTokenT::cap_TypeIndex) return;
  for (int i=0;i<HorSize;i++) {
       int indexPrev=0;
    if (myBorder==TopBorder) indexPrev=0;
    else if (myBorder==BottomBorder) indexPrev=VertSize+1;
    else printf("\n error in border type");         // last line
    prevworld[indexPrev][i]=borderP->world[0][i];
  }
  printf ("mergeborders %s %i\n", cap_NameS, (int) myBorder) ;
  //printArray(prevworld, VertSize+2, HorSize);
}

leaf operation ComputeServerT::ComputeStep
in void* InputP
out void* OutputP
{ // computes one iteration in prevworld and places it in currworld
  // traverses its 8 neighbours + itself
  int iv, ih, kv, kh, jv, jh;
  int count;
  OutputP = new capTokenT();                          // necessary, empty token
  for (iv=1;iv<=VertSize;iv++) {                      // top and bottom lines not computed
    for (ih=0; ih<HorSize; ih++) {
      // traverses its 8 neighbours + itself
      count = 0;
      for (kv=-1; kv<=1; kv++) {
        for (kh=-1; kh<=1; kh++) {
          jv = (iv+kv);
          jh = (ih+kh+HorSize) % HorSize;             // works horizontally in wrap around mode,
                                                      // % requires pos number
          // printf("\n jv,jh = %i , %i ", jv, jh);
          if ((jv==iv)&&(jh==ih)) ; else count=count+prevworld[jv][jh];
        }
      }
      switch( count ) // The law of life is applied here
      {
        case 2:   currworld[iv][ih]=prevworld[iv][ih]; break;      // remains alive
        case 3:   currworld[iv][ih]=ALIVE; break;                  // remains or becomes alive
        default:  currworld[iv][ih]=DEAD;                          // remains or becomes dead
      }
    }
  }
  // now copy currworld into prevworld
  for (iv=1;iv<=VertSize;iv++)                                     // top and bottom lines not computed
    for (ih=0; ih<HorSize; ih++)
      prevworld[iv][ih]=currworld[iv][ih];
    printf ("\n after process %s iteration \n", processP->cap_NameS) ;
  //printArray(prevworld, VertSize+2, HorSize);
}

leaf operation ComputeServerT::GetSubWorld (int threadIx)
// sends prevworld to main program
in void* InputP
out WorldT* OutputP
{
  int i,k;
  OutputP = new WorldT(VertSize,HorSize);
  OutputP->firstRow = localFirstRow;
  for (i=1;i<=VertSize;i++)
    for (k=0;k<HorSize;k++) {
      OutputP->world[localFirstRow+i-1][k]=prevworld[i][k];
      // OutputP->firstRow=threadIx * (TOT_VERT_SIZE/NUMBER_OF_COMPUTE_SERVERS);
    }
    printf ("\n process %s collect results \n", processP->cap_NameS) ;
  //printArray( prevworld, VertSize+2, HorSize) ;
}

void MergeSubWorld (WorldT* resworldP , WorldT* partworldP, int index)
{ // every cell of partworld must be copied into corrrect hor tile of resworld
  int k=0;
  for (int i=partworldP->firstRow;i<((partworldP->firstRow)+(partworldP->vertSize));i++)
    for (int j=0;j<partworldP->horSize;j++)
    {
      resworldP->world[i][j] = partworldP->world[i][j];
    }
}

operation ParallelServerT::parallelInitPartWorld
  in WorldT* InputP
  out WorldT* OutputP
{
  indexed
    (int i = 0 ; i < NUMBER_OF_COMPUTE_SERVERS ; i++)
  parallel ( SplitWorld, MergeWorld, Main, WorldT Result (thisTokenP->vertSize,thisTokenP->horSize))
  ( Sequential[i].initPartWorld(i)
  ) ;
}

operation ParallelServerT::ExchangeBorders (int partIx)
// this operation asks in parallel the neighbours to send their borders,
// merges them into the prevworld
// the parallel construct is lauched in the Main thread;
// where Sequential is called, it acts on the slave threads
// exchange of borders works in wraparound mode
```

```
  in void* InputP
  out void* OutputP
{
  parallel (Sequential[partIx], void result) (                     // indicates merge in Sequential
    (void
    , ifelse (partIx>0)
      (Sequential[partIx-1].sendBottomBorder)                      // partIx>0
      (Sequential[NUMBER_OF_COMPUTE_SERVERS-1].sendBottomBorder)   // partIx==0
    , mergeBorders(TopBorder)
    )
    (void
    , ifelse (partIx<NUMBER_OF_COMPUTE_SERVERS-1)          // partIx<NUMBER_OF_COMPUTE_SERVERS-1
      (Sequential[partIx+1].sendTopBorder)                 // partIx==NUMBER_OF_COMPUTE_SERVERS-1
      (Sequential[0].sendTopBorder)
    , mergeBorders(BottomBorder)
    )
  ) ;
}


operation ParallelServerT::Automaton(int nbIterations)
// on Main, sequences the indexed parallel ExchangeBorders and indexed parallel ComputeStep

  in void* InputP
  out void* OutputP
{
  // single iteration first
  for (int it=0;it<nbIterations;it++) (             // iterations of ExchangeBorders->ComputeStep
    indexed                                         // to synchronize exchange of borders
      (int i=0; i<NUMBER_OF_COMPUTE_SERVERS; i++)
    parallel (void,void,Main,void output)
      ( ExchangeBorders (i) )
    >->
    indexed
      (int j=0; j<NUMBER_OF_COMPUTE_SERVERS; j++)
    parallel (void,void,Main,void output)
      ( Sequential[j].ComputeStep)
  )
  ;
}




operation ParallelServerT::GetWorld
// gathering the tiles (sub worlds) into the final world
  in void* InputP
  out WorldT* lastResultP
{
    indexed                                         // collects the partial worlds (result)
      (int j=0; j<NUMBER_OF_COMPUTE_SERVERS; j++)
    parallel (void, MergeSubWorld, Main, WorldT lastResult(TOT_VERT_SIZE,HOR_SIZE))
      ( Sequential[j].GetSubWorld (j) )
    ;
}

ParallelServerT ParallelServer ;                    // instantiation de ParallelServerT

void get_world(WorldT* startWorldP, char* fileNameS)                /* Get input */
{
  FILE* fP ;
  int Row,Column;
  char ch;
  fP = fopen (fileNameS, "r") ;
  if (fP == 0) {
    fprintf (stderr, "unable to open file %s\n", fileNameS) ;
    exit (1) ;
  }
  for( Row=0; Row < startWorldP->vertSize; Row++ ) {
    for(  Column=0; Column < startWorldP->horSize+1; Column++ ) {   // +newline char
      ch= (char) getc (fP);
      if (ch=='O') startWorldP->world[Row][Column] = 1;
      else if (ch=='.') startWorldP->world[Row][Column] = 0;
      else if (ch == ’\n’ ) continue;
      else { fprintf(stderr,"Error in Data!\n"); exit (1); }
    }
  }
  fclose (fP) ;
}

parameter int NumberOfIterations = 1 ;              // lifePar -noi 1 -wfn wch1.in
parameter char* WorldFileName = "wch1.in" ;

int main (int argc, char** argv)
{
  NB_ITERATIONS = NumberOfIterations ;
  if (NB_ITERATIONS < 1) {printf("\n neg number of iterations");exit(1);}

  WorldT* fullworldP = new WorldT (TOT_VERT_SIZE,HOR_SIZE) ;
  WorldT* resultP ;

  WorldT* lastResultP;

  int i;
  get_world (fullworldP,WorldFileName) ;
  printWorld(fullworldP);
  call ParallelServer.parallelInitPartWorld in fullworldP out resultP ;
  printWorld(resultP);

  capTokenT* inputP = new capTokenT;
  capTokenT* outputP;
  call ParallelServer.Automaton(NB_ITERATIONS) in inputP out outputP;

  inputP = new capTokenT;
  call ParallelServer.GetWorld in inputP out lastResultP;
  printWorld(lastResultP);

  return 0 ;
}
```

Appendix 2

```
class complex{
  public:
  double i;
  double j;

  complex(double i = 0.0, double j = 0.0){
    this->i = i;
    this->j = j;
  }

  void Set(double i, double j){
    this->i = i;
    this->j = j;
  }

  friend complex operator+(complex a,complex b){
    return complex(a.i+b.i,a.j+b.j);
  }

  friend complex operator*(complex a, complex b){
    return complex(a.i*b.i-a.j*b.j,a.i*b.j+a.j*b.i);
  }

  double Magnitude(){
    return sqrt(i*i + j*j);
  }
};
```

Appendix 3

```
// ImageFile.h, SV, RDH 8.9.98

class ImageFile {
  FILE *file;
  char * filename;
  int sizeX;
  int sizeY;
  int nbBytesPerPixel;

public:
  ImageFile(char * name,int x, int y, int pixelSize); // constructor: initialization
  int SaveImage(unsigned char * buffer);
  int Open();
  void SaveHeader();
  void SaveData(unsigned char * buffer);
  void SaveDataChunk(int position, int size, unsigned char * chunk);
  void SaveLineContiguous(int size, unsigned char * linebuffer);
  void Close();
};

inline ImageFile::ImageFile(char * name,int x, int y, int pixelSize){
  filename = name;
  sizeX = x;
  sizeY = y;
  nbBytesPerPixel = pixelSize;
}

inline int ImageFile::SaveImage(unsigned char * buffer){    // saves image
  if (Open()==1) return 1;
  SaveHeader();
  SaveData(buffer);
  Close();
  return 0;
}

inline int ImageFile::Open(){
  //opens the output file
  if ((file = fopen(filename,"wb")) == NULL) {
    cout << "The file " << filename << " could not be opened\n" ;
    return 1;
  }
  //cout << "The file " << filename << " was successfully opened !\n" << flush;
  return 0;
}

inline void ImageFile::SaveHeader(){
    // generates the bytemap header parameters sizeX,sizeY, numberOfIntensity levels
     // for the PGM or PPM output bytemap file.

    fputs("P",file);

    /* Test if here are three bytes per pixel */
    if (nbBytesPerPixel==3)
        fprintf(file,"%c\n", ’6’);
    else
        fprintf(file,"%c\n", ’5’);

  fprintf(file, "%u %u\n",sizeX,sizeY);
    fprintf(file, "%u\n", 255);
}

inline void ImageFile::SaveData(unsigned char * buffer){
  fwrite(buffer, sizeof(unsigned char), sizeX * sizeY * nbBytesPerPixel, file);
}

inline void ImageFile::SaveDataChunk(int position, int size, unsigned char * chunk){
  // saves one image scanline into the file according to “position”
  fseek(file,position,SEEK_CUR);
  fwrite(chunk,sizeof(unsigned char),size,file);
  fseek(file,-(position+size),SEEK_CUR);
}
```

```
inline void ImageFile::SaveLineContiguous(int size, unsigned char * linebuffer){
  // saves a scanline contiguously onto a file
  fwrite(linebuffer,sizeof(unsigned char),size,file);
}

inline void ImageFile::Close(){
  fclose(file);
  //cout << "The file " << filename << " was successfully closed !\n" << flush;
}
```

RDH/9.4.99