Synthesizing Parallel Imaging Applications using the CAP Computer-Aided Parallelization tool[1]

B. A. Gennart, M. Mazzariol, V. Messerli, R.D. Hersch
Ecole Polytechnique Fédérale, Lausanne

Abstract. Imaging applications such as filtering, image transforms and compression/decompression require vast amounts of computing power when applied to large data sets. These applications would potentially benefit from the use of parallel processing. However, dedicated parallel computers are expensive and their processing power per node lags behind that of the most recent commodity components. Furthermore, developing parallel applications remains a difficult task : writing and debugging the application is difficult (deadlocks), programs may not be portable from one parallel architecture to the other, and performance often comes short of expectations.

In order to facilitate the development of parallel applications, we propose the CAP computer-aided parallelization tool which enables application programmers to specify at a high-level of abstraction the flow of data between pipelined-parallel operations. In addition, the CAP tool supports the programmer in developing parallel imaging and storage operations. CAP enables combining efficiently parallel storage access routines and image processing sequential operations. This paper shows how processing and I/O intensive imaging applications must be implemented to take advantage of parallelism and pipelining between data access and processing. This paper's contribution is (1) to show how such implementations can be compactly specified in CAP, and (2) to demonstrate that CAP specified applications achieve the performance of custom parallel code. The paper analyzes theoretically the performance of CAP specified applications and demonstrates the accuracy of the theoretical analysis through experimental measurements.

1 Introduction

Imaging computations such as filtering, image transforms, compression/decompression and image content indexing [3,4] require, when applied to large data sets (such as 3-D medical images, satellite images and aerial photographs), vast amounts of computing power. Such applications would potentially benefit from the use of parallel processing. However, dedicated parallel computers are expensive and their processing power per node lags behind that of the most recent commodity components. Moreover, developing parallel applications remains a difficult task : writing and debugging the application is difficult (deadlocks), programs are not portable, and performance often comes short of expectations.

In order to facilitate the development of parallel applications, we propose the CAP computer-aided parallelization tool which enables application programmers to specify at a high-level of abstraction the flow of data between pipelined-parallel operations.The CAP environment supports the programmer in developing parallel imaging applications. The CAP environment features (1) support for the parallel storage of large data sets ; (2) an image library supporting 1-bit, 8-bit, 16-bit, 24-bit images, as well as the division of images in tiles of user-defined size ; (3) the CAP language extension to C++ which allows to write deadlock-free portable pipelined-parallel applications, and combine parallel storage access routines and image processing operations.

This paper shows how processing and I/O-intensive imaging applications can be implemented to take advantage of parallelism and pipelining between data access and processing operations. This paper's contribution is (1) to show how such implementations can be compactly specified using the CAP set of flow control instructions, and (2) to demonstrate that CAP specified applications achieve the performance of custom code. The paper analyzes theoretically the performance of CAP specified applications and demonstrates the accuracy of the theoretical analysis through experimental measurements. To implement I/O intensive applications, large 2D (resp. 3D) images are divided into square (resp. cubic) subsets with good locality called *tiles*. Two kinds of applications are considered in this paper : *neighborhood-independent* operations, and *neighborhood-dependent* operations. Neighborhood-independent operations are operations where no data must be exchanged between tiles to compute the resulting final image.

Section 2 shows how large images are divided in tiles for storage and processing purposes. Section 3 describes in general terms the *process-and-gather* operation, i.e. the problem of applying a neighborhood-independent operation to selected tiles stored on disk(s) and gathering the processed tiles in a single address space. It shows the ideal execution

---

1. Preliminary version

schedule for performing a process-and-gather operation and analyzes theoretically the performance of such a schedule. Section 4 shows how process-sand-gather operation is specified in CAP. Section 5 describes the *exchange-process-and-store* operation, i.e. the problem of applying a neighborhood-dependent operation to tiles stored on disk(s) and storing the result back to disk(s). Section 6 lists performance results for the process-and-gather and exchange-process-and-store parallel operations.
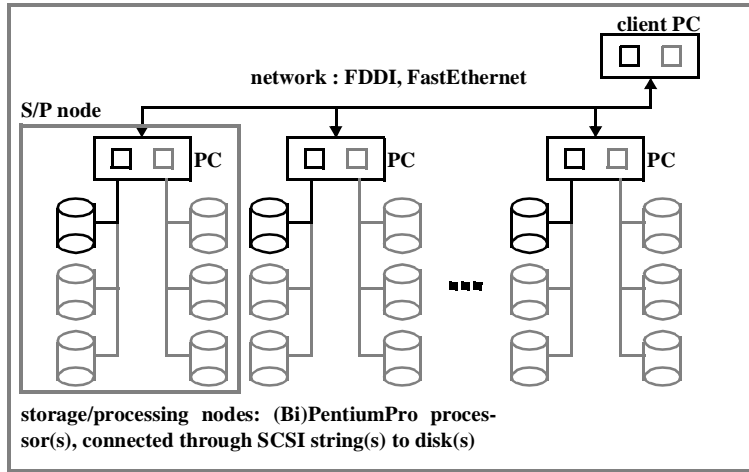

FIGURE 1. Commodity component based parallel architecture

2 System support for managing large images

2.1 Hardware architecture

The hardware we consider consists of multiple PentiumPro PC's connected through a commodity 1000Mb/s network such as FDDI or Fast Ethernet (Figure 1). The PCs run the WindowsNT operating system and communicate using the TCP/IP-based MPS message-passing system developed by the authors. Each PC represents a storage/processing node (S/P node) consisting of one or two processors connected through its PCI internal bus to one or more disks. The client requesting some image processing operation is also located on a PC. This platform scales from a single PC architecture with one processor and one disk, to a multiple-PC multiple-disk architecture. For the purpose of comparing modeled and measured performance figures, we will assume single processor PCs. We assume that both the disk and the network can access memory through DMA (Direct Memory Access). While this hypothesis is accurate for disks[2], network interfaces based on the TCP/IP protocol consume a lot of processing power[3].
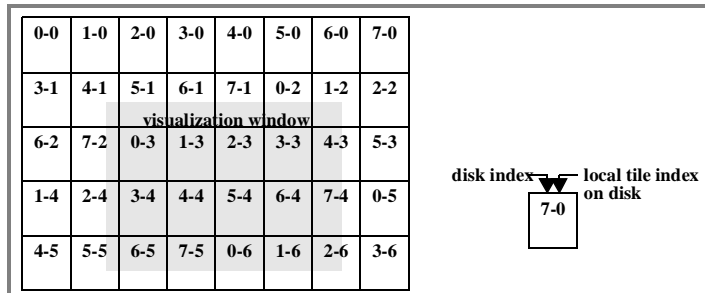

FIGURE 2. Image tiling and visualization window

2.2 Software architecture

Large images are divided in square tiles which are stored independently, possibly on multiple disks. Pixmap image tiling is routinely used for the internal representation of images in software packages such as PhotoShop. Square tiles enable accessing image windows efficiently, with a good data locality. The CAP imaging library provides data types and functions for splitting images in tiles, and allocating tiles to disks. Figure 2 shows an image divided in tiles, as well as a visualization window covering part of the image. Figure 2 also shows a possible allocation of tiles to disks, assuming an image striped over 8 disks. The allocation index consists of the disk index, as well as the local tile index on the disk. For example, the bottom right tile in Figure 2 is allocated on disk 3, and is the 6th tile on that disk. The

---

2. Our experience shows that for disk throughputs of up to 45 MB/s (throughput achieved by connecting 15 disks to 5 SCSI boards, the processor utilization remains below 15%.

3. Our experience shows that a PentiumPro processor reaches a 100% utilization rate when the Fast-Ethernet throughput reaches 5MB/s.

distribution of tiles to disks is made so as to ensure that direct tile neighbors reside on different disks. We achieve such a distribution by introducing, between two successive rows of tiles (and between two successive planes of tiles in the case of 3-D images), offsets which are prime to the number of disks.

The data types required in this example are the *WindowT* and the *TileT* classes, provided in the CAP imaging library (Program 1). The *WindowT* class fields are a file name, the window position within an image, the window size, the window data, and a pixel descriptor (pixel size in bit, color scheme (gray level, RGB, ...). The *WindowT* class is used both to specify a window request parameters (in which case the data field is empty), as well as the window itself. The *TileT* class fields give its position, its size, the tile data, a pixel descriptor, as well as the index of the disk where the tile is stored, and the local index of the tile on the disk.

```
1   class WindowT {                          1   class TileT {
2     char* FileName ;                        2     int DiskIndex
3     int PositionX, PositionY ;              3     int LocaTileIndex ;
4     int SizeX, SizeY ;                      4     int PositionX, PositionY ;
5     char* DataP ;                           5     int SizeX, SizeY ;
6     PixelDescriptorT Descr ;                6     char* DataP ;
7   } ;                                       7     PixelDescriptorT Descr ;
                                              8   } ;
```

PROGRAM 1. CAP imaging-package data-types

Program 2 lists a simple sequential program which performs a process-and-gather operation. It assumes that the whole window to be processed is in memory. The while loop (lines 12 to 16) repeatedly calls the *GetNextTileInWindow* routine, until it returns 0. At each iteration, the *GetNextTileInWindow* routine returns the next window tile (*nextP* parameter), based on the window description (*windowP* parameter) and the previous window tile (*prevP* parameter). The first time the *GetNextTileInWindow* is called, the *prevP* parameter is 0. In the body of the while loop (lines 13 to 15), the new tile is processed using the user-defined *ProcessTile* routine. In this simple program, all tiles are processed independently. When the tile is processed, the result is merged into the final window using the *MergeAndAddTile* routine. This simple routine handles correctly neighborhood-independent and linear filtering operations (see section 3.1) The *GetNextTileInWindow* and the *MergeAndAddTile* are provided by the CAP imaging library. The *ProcessTile* routine is user-defined.

```
1    int GetNextTileInWindow (WindowT* windowP, TileT* prevP, TileT* &nextP) ;
2    void MergeAndAddTile (WindowT* windowP, TileT* tileP) ;
3    void ProcessTile (TileT* inputP, TileT* &outputP) ;
4
5    void ProcessWindowByTile (WindowT* inputP, WindowT* &outputP)
6
7    {
8      TileT* prevP = 0 ;
9      TileT* nextP ;
10     TileT* processedTileP ;
11     outputP = new WindowT (...) ;
12     while (GetNextTileInWindow (inputP, prevP, nextP)) {
13       ProcessTile (nextP, processedTileP) ;
14       MergeAndAddTile (outputP, processTileP) ;
15       prevP = nextP ;
16     }
17   }
```

PROGRAM 2. Sequential processing of a window, tile by tile

The next sections show more sophisticated imaging programs which in a pipelined-parallel manner access tiles stored on disks and perform on it processing operations.
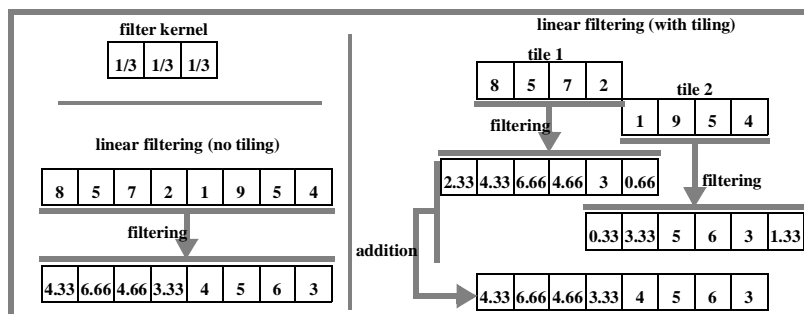

FIGURE 3. Applying a linear filtering operation to a tiled 1-D image

3 The parallel process-and-gather operation

3.1 Problem description

A process-and-gather operation consists of reading tiles from the disk(s), performing a neighborhood-independent operation on the tiles, gathering the processed tiles in a single address space, and merging the tiles to form the final visualization window. The S/P nodes perform the neighborhood-independent operation, and send the processed tiles to the client PC, where processed tiles are merged to form the final image. Linear filtering fits the process-and-gather

scheme (Figure 3). The linear operation is performed on each tile, assuming that pixels beyond the tile border are set to 0. The linear filtering operation generates enlarged tiles. After filtering, tiles overlap. When merging the tiles to form the final image, the overlapping part of the tiles are added together, leading to the correct final result. As an example, we filter a 1-D gray-level image, by averaging pixels with a 3-by-1 convolution kernel (Figure 3). In Figure 3, a 3-by-1 convolution kernel is applied to an 8-pixel vector, with and without tiling. Both situations assume that pixels outside the range of the vector are 0. With tiling, the filter is applied to two 4-pixel vector slices, and both vector slices grow to 6 pixels after filtering. The overlapping parts of the vector slices are then added at tile-merging time to recover the correct 8-pixel vector.

3.2 Modelled single-PC execution schedule

This hardware configuration consists of a single PC reading data from the disks, performing the neighborhood-independent operations on all tiles, and merging the processed tiles. We assume that the disks read tiles faster than the processor can handle them (Figure 4).
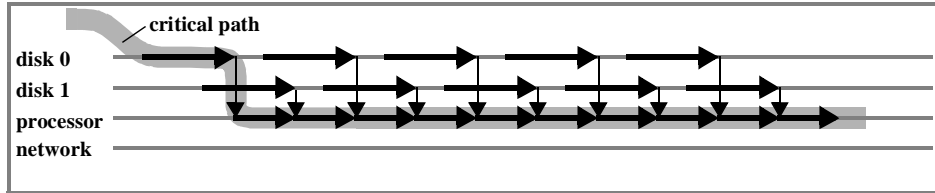

FIGURE 4. Execution schedule (single PC, multiple disks, processor bottleneck)

The schedule described in Figure 4 guarantees that the PC processor is busy at all times, after the first tile has been fetched. In this model, all disk accesses but the first one are performed while the processor is busy.

3.3 Modelled multiple-PC execution schedule

Figure 5 shows the ideal execution schedule for a multiple S/P node situation. We assume that the disks read tiles faster than the processors can handle them, that the network transfers processed tiles faster than the processor can produce them, and that the client PC merges processed tiles faster than the network can transfer them.
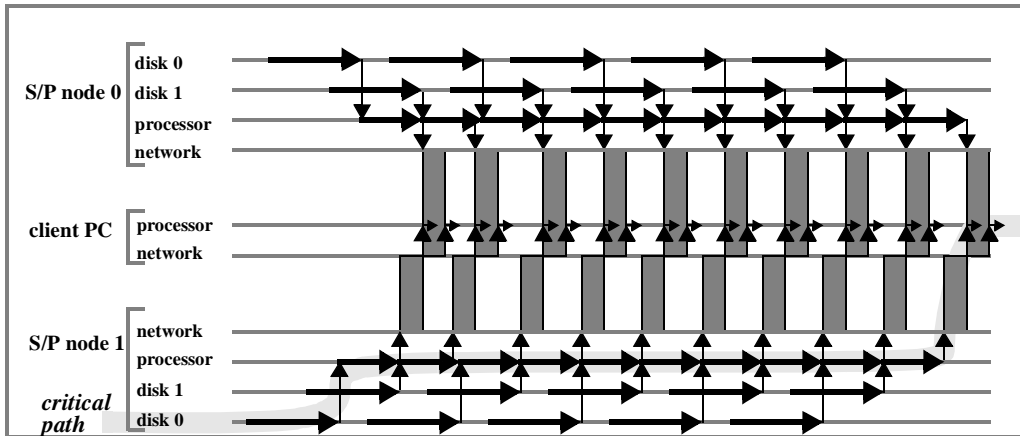

FIGURE 5. Execution schedule (multiple single-processor PCs with two disks, slave processor bottleneck)

In Figure 5, horizontal arrows represent disk access and processing operations ; vertical arrows represent ordering between operations ; gray boxes represent data transfers between PCs. The critical path is represented as a smooth light-gray line. As in section 3.2, this schedule ensures that all disk transfers but the first one, all network transfers but the last P (where P is the number of S/P nodes) and all merging operations but the last one are performed while the S/P node processors perform the neighborhood-independent tile computations.

3.4 Theoretical performance analysis

As described in Figure 5, the critical path in the pipelined-parallel process-and-gather operation consists of one disk access, $\lceil T/P \rceil$ tile processing steps, P network transfers and one ***MergeAndAddTile*** operation, where T is the number of tiles in the window, and P the number of S/P nodes in the architecture. We assume that the tile size does not change significantly during the neighborhood-independent computation. The time required to read a tile from a disk is written as $t_d = l_d + \tau_d \cdot \text{TileSize}^2$ where l_d is the disk latency and $1/\tau_d$ is the disk throughput. The time required to transfer data over the network is written as $t_n = l_n + \tau_n \cdot \text{TileSize}^2$ where l_n is the network latency and $1/\tau_n$ is the network throughput. The time required to process a tile is written as $t_p = \tau_p \cdot f(\text{TileSize})$ where τ_p is the unitary computation time and f gives the complexity of the algorithm as a function of the tile size. The time required to merge a tile into the visualization window is $t_m = \tau_m \cdot \text{TileSize}^2$. The duration of the process-and-gather operation is (Equation 1) :

$$T = t_d + \left\lceil \frac{T}{P} \right\rceil \cdot t_p + P \cdot t_n + t_m \tag{1}$$

The assumptions behind the execution schedule of Figure 5 are that tile accesses are faster than tile processing steps ($t_d < D \cdot t_p$), where D is the number of disks per S/P node, that P network transfer times are faster than a single tile processing step ($P \cdot t_n < t_p$), and that merging a tile into a window is faster than a network transfer step ($t_m < t_n$).

4 CAP specification of the process-and-gather operation

4.1 The computer-aided parallelization framework

In order to speedup the development of parallel applications and to specify parallel I/O and processing operations at a high level of abstraction, we use the Computer-Aided Parallelization (CAP) tool. This tool enables application programmers to hierarchically specify the *macro dataflow* between *operations* performed on tiles (file stripe parts). Operations are segments of sequential code performed by a single execution thread and characterized by input value and output values. The input and output values of an operation are called *tokens*. In the context of this paper, tokens consist of tile data and additional application-dependent parameters. The macro dataflow specifies how tokens are routed between the operations of the parallel program. In addition, synchronization points (also used for merging intermediate results) specify which tokens must be available before the next operation can start (Figure 6).
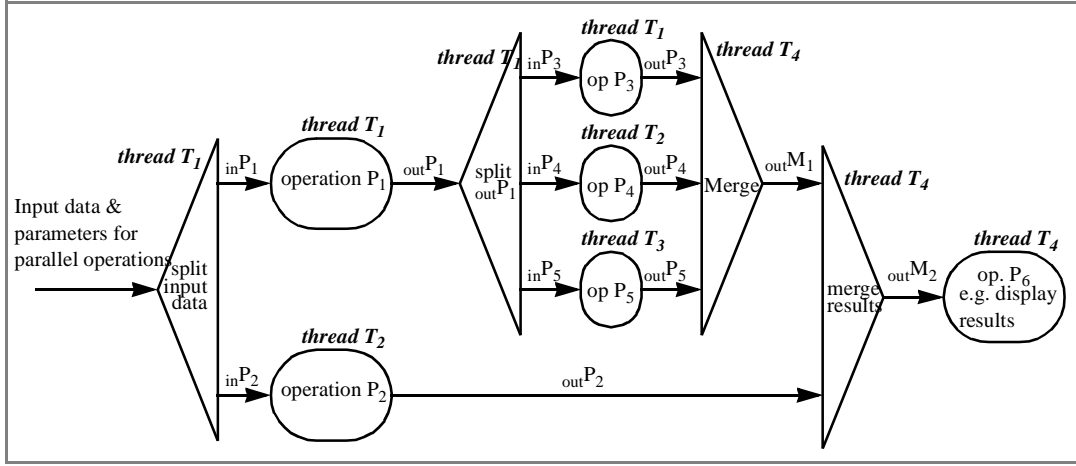

FIGURE 6. Graphical CAP macro dataflow specification

In a graphical CAP specification, parallel operations are displayed as parallel horizontal branches, pipelined operations are operations located in the same horizontal branch. Figure 6 assumes a parallel program consisting of 4 threads T_1, T_2, T_3, and T_4. In the macro data flow graph of Figure 6, the input token enters the graph from the left. It is divided into two parts, $_{in}P_1$ and $_{in}P_2$, which undergo operations P_1 and P_2. Operation P_1 is performed by thread T_1. Operation P_2 is performed by thread T_2. The result of operation P_1 is $_{out}P_1$. $_{out}P_1$ is divided into three tokens $_{in}P_3$, $_{in}P_4$, $_{in}P_5$, which undergo operations P_3, P_4 and P_5 in parallel (threads T_1, T_2 and T_3). The results of operations P_3, P_4 and P_5 are merged into a single token, $_{out}M_1$, which is in turn merged with $_{out}P_2$ to form $_{out}M_2$. $_{out}M_2$ is fed to operation P_6. If several tokens enter the macro data flow graph of Figure 6, they are processed in a pipelined fashion.

The CAP specification of a parallel program is described in a simple formal language, an extension of C++. This specification is translated automatically into a C++ source program. At program startup time, the CAP runtime allocates the program threads to the available processors, using the information stored in a configuration file [1]. The macro data flow model which underlies the CAP approach has also been used successfully by the creators of the MENTAT parallel programming language [2].

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

CAP threads are grouped hierarchically. In the context of this paper, the *CapServerT* thread hierarchy (Program 3) consists of a client thread running on the client node (line 3) and two sets of threads running on the S/P nodes (lines 4 and 5). The *TileServer* threads perform I/O operations (*ReadTile* and *WriteTile*, lines 16 to 19) and the *ComputeServer* threads perform computations on the tiles extracted from the disks (e.g. filtering, lines 25 and 26). Each S/P node comprises one *ComputeServer* thread and as many *TileServer* threads as disks. The CapServerT thread hierarchy can

perform two parallel operations : the process-and-gather and the exchange-process-and-store operations. Section 4.2 and 5.3 specify the behavior of these two operations.

```
1    process CapServerT {
2    subprocesses:
3      ClientProcessT Client;
4      TileServerT TileServer[NDISK];      // NDISK:total nb of disks
5      ComputeServerT ComputeServer[NSTOR]; // NSTOR:total nb of
6    operations:                          // storage/processing nodes
7      ProcessAndGather (FilterT filter)
8        in WindowT Input out WindowT Output;
9      ExchangeProcessAndStore (FilterT filter)
10       in WindowT Input out void Output;
11     ...
12   };
13
14   process TileServerT {
15   operations:
16     ReadTile
17       in TileReadingRequestT Input out TileT Output;
18     Writetile
19       in TileWritingRequestT Input out void Output;
20     ...
21   };
22
23   process ComputeServerT {
24   operations:
25     Filtering (FilterT filter)
26       in TileT Input out TileT Output;
27   };
```

PROGRAM 3. Parallel storage and processing system threads

4.2 Cap specification of the process-and-gather operation

Program 4 is the CAP specification of the process-and-gather operation declared in Program 3, line 7 and 8. This program applies in a pipeline-parallel manner a linear filter to all tiles within a window specified by the *WindowT Input* class instance. The *WindowT* class consists of a window position, and a file name (see section 2.2). The CAP *pipeline* expression semantics (Program 4, lines 4 to 7) is to perform in parallel the body of the pipeline (lines 5 and 6). The pipeline expression iteratively calls the *GetNextTileInWindow* routine (line 4, first pipeline parameter) until it returns 0. Each token generated by the call is immediately (before the next token is generated) sent to the appropriate *TileServer* thread which reads a tile (line 5). The tile is then processed by the ComputeServer thread (line 6). Tiles sent to different S/P nodes are processed in parallel. Tiles sent to the same S/P node are processed in pipeline : the *TileServer* thread fetches the next tile, while the *ComputeServer* is processing the previous tile. When a tile has been processed, it is returned to the Client PC (line 4, third initialization parameter) where it is merged using the *MergeTile* routine (line 4, second initialization parameter) into the final window (line 4, fourth initialization parameter).

```
1    operation CapServerT::ProcessAndGather (FilterT filter)
2      in WindowT Input out WindowT Output
3    {
4      pipeline (GetNextTileInWindow, MergeTile, Client, WindowT Window)
5        ( TileServer[thisTokenP->DiskIndex].ReadTile
6        >-> ComputeServer[thisTokenP->DiskIndex].Filtering (filter)
7        )
8    }
```

PROGRAM 4. CAP process-and-gather operation

Program 4 performs in a pipeline-parallel manner the operations performed sequentially by Program 2. It specifies all communications and synchronizations required to implement the execution schedule of Figures 4 and 5. If a single PC with two disks is available, Program 4 follows the schedule of Figure 4. If two S/P nodes with a total number of 4 disks are available, it follows the schedule of Figure 5.

5 The exchange-process-and-store operation

5.1 Problem description

We consider the situation where the source image resides on disks, and the target image is written to disk(s). Before filtering can be performed on a tile, tile sides must be exchanged : a tile must receive pixels from its 8 neighboring tiles. The number of exchanged pixels depends on the filtering operation. The width of the border exchanged between tiles is defined as *w* and is application dependent (Figure 7).

We consider that the bottleneck are the processors. We select an execution schedule where the processors are always busy. We must ensure that the required data to compute a given tile is in memory when the computation starts, i.e. the required data is read from the disks, and exchanged between the various S/P nodes before the computation is started.

To achieve this result, we use a three-step pipeline (Figure 8). The first step consists of reading from the disk(s) into memory data for the next computation step. No data is exchanged between S/P nodes during the first step. The second step consists of processing tiles. The third step consists of writing back to the disks tiles processed during the previous computation step. The second pipeline step is itself divided in two parts : (1) each S/P node computes the *central part*

of the tile and in parallel reads the neighboring tiles' borders from the other S/P nodes ; (2) each S/P node computes the *border* of the tile after it has received the neighboring tile sides. The tile central part is defined as the part of the tile that is not affected by the neighboring tile sides. The tile border is defined as the part of the tile that is affected by the neighboring tile sides. The index of the S/P node processing a tile is equivalent to the tile disk index. For example, in Figure 8, tile 3-5 is stored as local tile 52 on disk 2. This tile is processed by S/P node 2. In the present allocation of tiles to disks, adjacent tiles on the same row are processed by different S/P nodes.
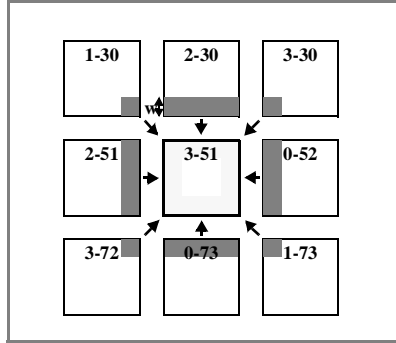

FIGURE 7. Borders exchanged during a neighbourhood dependent operation

Figure 8 shows the activity pattern during a computation step. During the algorithm step of Figure 8, S/P node 0 (resp. 1, 2, 3) reads tile 6-4 (resp. {3-4, 7-4}, {4-4, 8-4}, 5-4) from disk, processes tile 4-2 (resp. 5-2, 6-2, 7-2), and writes tile 5-1 (resp. 6-1, 7-1, 4-1) to disk. For the next computation step, the activity pattern is shifted one tile to the right. The CAP environment features a tile cache keeping loaded tiles in memory. The tile cache works according to a LRU (least-recently used) scheme. Provided that the cache can store at least 24 tiles, the required tiles will all be in memory during a given computation step. The typical tile size being 50KB, 24 tiles represent 1.2MB allocated on 4 S/P nodes, well below the typical memory size of current PCs. The additional buffer required for temporary results is not larger than one tile on each S/P node.
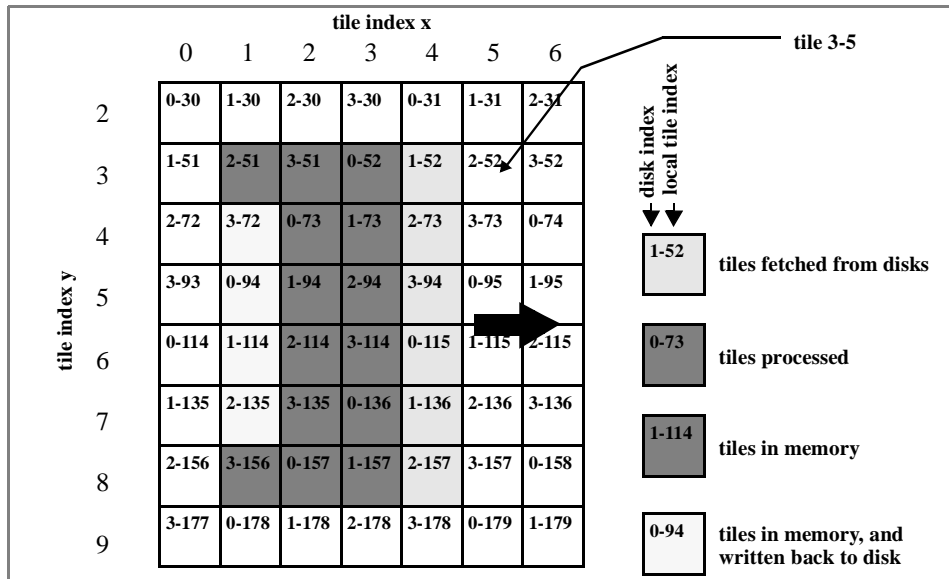

FIGURE 8. Activity pattern among image tiles during a computation step

A number of trade-offs must be taken into considerations :

1. The basic activity pattern can be adapted so that each S/P node processes more than one tile during each computation step. This reduces the number of synchronizations during the course of the algorithm, but increases the pipeline startup cost.

2. For the same reasons, an increase in tile size reduces the number of synchronizations during the execution of the algorithm, but increases the pipeline startup cost.

3. As explained in section 2.2, the tile allocation scheme selected in this paper ensures that neighboring tiles are allocated on different S/P nodes, to improve load balancing. This in turn increases the number of communications required during each computation step. An alternative tile allocation scheme could optimize communications over load balancing, and would be easy to specify in CAP.

4. Alternative pipelining schemes can be considered. A four step pipeline (reading tile(s) from disk, tile border exchange, tile processing, writing the tile back to disk) would simplify the processing step, but could increase the pipeline startup cost.

5.2 Theoretical performance analysis

During each computation step, four activities are carried out simultaneously. Each S/P node reads on the average one tile from disk ; communicates with the other S/P nodes to get at most 4 tile sides and 4 tile corners ; computes one tile ; writes one tile to the disk. Assuming that tile processing operation is a computation intensive operation (as opposed to data-intensive), we aim to keep the S/P node processors busy at all times by reading in advance data from the disks.

The time required to read a tile from the disks is written as $t_d = l_d + \tau_d \cdot \text{TileSize}^2$ where l_d is the disk latency and $1/\tau_d$ is the disk throughput. The time required to transfer data over the network is written as $t_n = l_n + \tau_n \cdot \text{DataSize}$ where l_n is the network latency and $1/\tau_n$ is the network throughput. The time required to process a tile is written as $t_p = \tau_p \cdot f(\text{TileSize})$ where τ_p is the unitary computation time and f gives the complexity of the algorithm as a function of the tile size. The width of a tile border is defined as w. The number of S/P nodes is defined as P. The number of disks per S/P node is D. The number of tiles in the window is T. Considering this, the conditions that the processing time during each step be superior to both the disk access and the network transfer times are formulated as follows (Equations 2 and 3) :

$$\text{disk access time} < \text{processing time} : \quad (l_d + \tau_d \cdot \text{TileSize}^2) < \tau_p \cdot f(\text{TileSize}) \tag{2}$$

$$\text{network transfer time} < \text{processing time} : \quad 4(l_n + \tau_n \cdot \text{TileSize} \cdot w) + 4(l_n + \tau_n \cdot w^2) < \tau_p \cdot f(\text{TileSize}) \tag{3}$$

The pipeline startup cost is the cost of preloading two columns of P+2 tiles. The pipeline termination time is the cost of writing back one column of P tiles. The total computation time with P S/P nodes consists of the pipeline startup cost, the computation time, and the pipeline termination time (Equation 4).

$$T = \left\lceil \frac{2(P+2)}{P \cdot D} \right\rceil \cdot t_d + \left\lceil \frac{T}{P} \right\rceil \cdot t_p + t_d \tag{4}$$

Equation 4 shows that, with 4 S/P nodes, the pipeline startup cost is at most the cost of loading three tiles in memory (D = 1), and can be reduced to the cost of loading one tile (D >= 3). Provided the number of tiles is large, the relative startup cost can become very small. The trade-off in the tile size is that (1) the larger the tile size, the smaller the overhead due to synchronization and communications ; (2) the smaller the tile size, the smaller the overhead due to pipeline startup cost.

```
 1   operation CapServerT::TileFiltering (FilterT filter, int P, int i, int j, int k)
 2     in WindowT Input
 3     out void Output
 4   {
 5     parallel (ComputeServer[k], TileBordersT Result) (
 6       ( void, GetTileBordersFromNeighbours (P, i, j, k), SetTilesBorders )
 7       ( void, ComputeServer[k].ProcessTileCentralPart, void )
 8     ) >-->
 9     ComputeServer[k].ProcessTileBorders ;
10   }
11
12   operation CapServerT::ComputationStep (FilterT Filter, int P, int i, int j)
13     in WindowT Input
14     out void Output
15   {
16     parallel (Client, void Result)
17       ( DuplicateWindow, GetNextStepTiles (P, i, j), void )
18       ( DuplicateWindow, SavePreviousStepTiles (P, i, j), void )
19       ( DuplicateWindow
20       , indexed
21          (int k = 0 ; k < P ; k++ )
22        parallel ( DuplicateWindow, void, Client, void Result )
23          ( ComputeServer[k].TileFiltering (filter, i+k, j) )
24       , void
25       ) ;
26   }
27
28   operation CapServerT::ExchangeProcessAndStore (FilterT filter, int P)
29     in WindowT Input
30     out void Output
31   {
32     for ( int i = 0 ; i < VerticalNumberOfBands (Input) ; i += P )
33       (    GetNextStepTiles (P, i, -2)
34       >-> GetNextStepTiles (P, i, -1)
35       >-> for (int j = 0 ; j < HorizontalNumberOfBands (Input) ; j++)
36            ( ComputationStep (filterT, P, i, j)
37       >-> SavePreviousStepTiles (P, i, HorizontalNumberOfTiles (Input))
38       ) ;
39   }
```

PROGRAM 5. CAP specification of the exchange-process-and-store operation

5.3 CAP specification

Program 5 is the CAP specification of the exchange-process-and-store operation (lines 28 to 38). The program consists of a double sequential iteration. The first loop (lines 32 to 38) iterates on all *tile bands* in the **Input** window, a tile band being a P-tile high horizontal window slice (where P is the number of S/P nodes in the architecture). The second

loop iterates on all vertical tile slices within a single tile band. Before each step of the first iteration can start, we must ensure that all the required tiles are in memory (lines 32 and 33). After each step of the first iteration, we must write back the P tiles of the last window column (line 37).

```
 1   operation CapServerT::GetNextStepTiles (int P, int i, int j)
 2     in WindowT Input
 3     out void Output
 4   {
 5     indexed
 6       (int k = -1 ; k <= P ; k++ )
 7     parallel ( GetTileIndices (P, i, j+1, k), void, Client, void Result )
 8       ( if (thisTokenP->DiskIndex != -1)
 9         ( TileServer[thisTokenP->DiskIndex].ReadTile )
10       ) ;
11   }
12
13   operation CapServerT::SavePreviousStepTiles (int P, int i, int j)
14     in WindowT Input
15     out void Output
16   {
17     indexed
18       (int k = 0 ; k < P ; k++ )
19     parallel ( GetTileIndices (P, i, j-1, k), void, Client, void Result )
20       ( if (thisTokenP->DiskIndex != -1)
21         ( TileServer[thisTokenP->DiskIndex].WriteTile )
22       ) ;
23   }
```

PROGRAM 6. GetNextStepTiles and SavePreviousStepTiles operations

Each computation step (lines 12 to 26) consists of three parts, performed in parallel : get the tiles for the next computation step (line 17) ; save the tiles computed during the previous iteration (line 18) ; let each of the compute servers apply the filter to one tile (lines 20 to 23). The parallel construct semantics (line 16 to 25) is to initialize in the client address space (line 16, first initialization parameter) a void output token for synchronization purpose (line 16, second initialization parameter) and perform in parallel the three construct bodies (line 17, line 18, and line 19 to 24). Each parallel-construct body consists of three parts : a *split-function* name, a CAP expression, and a *merge-function* name. The split-function indicates how to create the parallel-construct body-input token from the parallel-construct input token. The merge-function indicates how to merge the parallel-construct body-output token into the parallel-construct output-token. At line 17, the split-function *WindowDuplicate* duplicates the window parameters ; the merge-function is void, indicating that the parallel-construct body-output is used for synchronization purposes only. The third parallel-construct body (lines 20 to 24) is itself a CAP instruction, the indexed-parallel expression. The indexed parallel expression executes in parallel instances of its body (line 23), as many times as expressed in the index specification (line 23).

The tile filtering process consists of in parallel applying the filter to the tile central part (line 7) and fetching the tile sides from the neighboring S/P nodes (line 6). Once the tile borders have been fetched, the filter can also be applied to the tile borders (line 9).

```
 1   enum NeighbourT {
 2     TopNeighbour, TopRightNeighbour, RightNeighbour, BottomRightNeighbour,
 3     BottomNeighbour, BottomLeftNeighbour, LeftNeighbour, TopLeftNeighbour
 4   } ;
 5
 6   token TokenBordersT {
 7     DataT Sides[8] ;
 8   } ;
 9
10   operation CapServerT::GetTileBordersFromNeighbours (int P, int i, int j, int k)
11     in void Input
12     out TileBordersT Output
13   {
14     indexed
15       ( NeighbourT n = TopNeighbour ; n <= TopLeftNeighbour ; n++ )
16     parallel ( void, SetBorder (n), ComputeServer[k], TileBordersT Result ) (
17       ( TileServer[DiskIndex(P,i,j,k,n)].GetBorder (P,i,j,k,n) ) ;
18   }
```

PROGRAM 7. GetTileSidesFromNeighbours operation

The input token to the exchange-process-and-store operation is a window description (image name, size, position, but no data). The window description is duplicated and sent to each compute server (*DuplicateWindow* routine, lines 19 and 22). A copy of the window is also sent to the routine in charge of getting the next step tiles (line 17) and saving the previous step tiles (line 18). The CAP procedures for fetching the next step tiles, saving the previous step tiles and getting the tile sides from the neighboring S/P nodes are specified in programs 6 and 7.

Program 6 describes the *GetNextStepTiles* and *SavePreviousStepTiles* operations. The *GetNextStepTiles* operation consists of reading in parallel from disk all tiles related to a given computation step described by indices i and j. There are P+2 tiles involved in the *GetNextStepTiles*, indexed from -1 to P. The *GetTileIndices* routine returns the disk index and the local tile index on the disk for each tile. It sets the disk index to -1 for tiles outside the visualization window. The

CAP if construct checks that the disk index is not -1 (line 8) before calling the appropriate TileServer thread. The *SavePreviousStepTiles* consists of writing in parallel all tiles from a previous computation step (lines 13 to 23).

Program 7 is the CAP specification of the *GetTileSidesFromNeighbours* operations. It consists of fetching from neighboring tiles in parallel the appropriate tile side or corner. The index of the neighbor that stores a tile is given by the *DiskIndex* function. Each tile server supports a *GetBorder* operation (line 17) which reads a tile part, and returns it as its output token. The tokens are transferred to *ComputeServer[k]* (line 16, third *indexed-parallel* parameter) which requested the tile parts and gathered using the *SetBorder* routine into a single *TileBordersT* token.

6 Performance results

(* In this section we will present and discuss performance results, and compare them to the theoretical models of sections 3.4 and 5.2. *)

7 Conclusions

This paper shows that CAP enables the compact specification of pipelined-parallel imaging applications. The CAP environment is not restricted to the process-and-gather and exchange-process-and-store operations described in this paper. It can be applied to any imaging algorithm, including *non-oblivious* algorithms[4]. Once the imaging library is available, the implementation and test effort for the two applications described in this paper is of the order of days. The generated programs run on PCs under WindowsNT and on Sun workstations under Solaris. With a limited effort, reusable and customizable parallel code can be produced. The programs generated use as runtime systems either the MPS communication library developed by the authors, or the familiar PVM communication package.

(* comments on the performance results *)

The CAP imaging library supports the subdivision of images in tiles. The CAP language handles the communication and synchronization of messages in a parallel program. These two features of the CAP environment free the programmer to concentrate on the algorithm(s) to be applied to the image. Once the algorithm has been designed, the programmer can either reuse existing CAP programs and modify the processing operation performed on each tile, or create new CAP programs to handle new parallel execution schedules.

The CAP language has also been applied to the parallelization of linear algebra algorithms (matrix multiplication and LU decomposition), and to the visualization of 3-D tomographic images. In the field of linear algebra, speedups of 18 with 20 processing nodes, and 8.4 with 10 processing nodes have been demonstrated for the matrix multiplication and the LU factorization respectively. The experimental analysis of the parallel 3-D tomographic imaging application (plane extraction) has shown that the application achieves near linear speed-ups for a hardware architecture consisting of up to 4 PCs (3 S/P nodes and a client) and up to 27 disks working in pipeline parallel fashion. Both applications have demonstrated that the overhead of CAP is very low.

References

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

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

[3] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker and C. Faloutsos, "The Q-BIC project: querying images using colour, texture and shape", *IBM Research Report RJ 9203 (81511)*, Feb.1 1993.

[4] W. Equitz, "Image searching in a shipping product", *Proc. Conf. IS&T/SPIE-Storage and Retrieval for Image And Video Databases III*, pp. 186-196, Feb 1995, San Jose, California.

---

4. Oblivious algorithms are algorithms whose execution flow is independent of the content of the data being processed.