Dynamic load balancing of parallel cellular automata

Marc Mazzariol, Benoit A. Gennart, Roger D. Hersch
Ecole Polytechnique Fédérale de Lausanne, EPFL*

ABSTRACT

We are interested in running in parallel cellular automata. We present an algorithm which explores the dynamic remapping of cells in order to balance the load between the processing nodes. The parallel application runs on a cluster of PCs connected by Fast-Ethernet.

A general cellular automaton can be described as a set of cells where each cell is a state machine. To compute the next cell state, each cell needs some information from neighbouring cells. There are no limitations on the kind of information exchanged nor on the computation itself. Only the automaton topology defining the neighbours of each cell remains unchanged during the automaton's life.

As a typical example of a cellular automaton we consider the image skeletonization problem. Skeletonization requires spatial filtering to be repetitively applied to the image. Each step erodes a thin part of the original image. After the last step, only the image skeleton remains. Skeletonization algorithms require vast amounts of computing power, especially when applied to large images. Therefore, skeletonization application can potentially benefit from the use of parallel processing.

Two different parallel algorithms are proposed, one with a static load distribution consisting in splitting the cells over several processing nodes and the other with a dynamic load balancing scheme capable of remapping cells during the program execution. Performance measurements shows that the cell migration doesn't reduce the speedup if the program is already load balanced. It greatly improves the performance if the parallel application is not well balanced.

Keywords : parallel cellular automata, dynamic load balancing, image filtering, skeletonization, Computer-Aided Parallelization (CAP), cluster of PCs

1 INTRODUCTION

We are interested in running in parallel cellular automata. We present an algorithm which explores the dynamic remapping of cells in order to balance the load between the processing nodes. The parallel application runs on a cluster of PCs (Windows NT) connected by Fast-Ethernet (100 Mbits/sec).

A general cellular automaton[1,2] can be described as a set of cells where each cell is a state machine. To compute the next cell state, each cell needs some information from neighbouring cells. There are no limitations on the kind of information exchanged nor on the computation itself. Only the automaton topology defining the neighbours of each cell remains unchanged during the automaton's life.

Let us describe a simple solution for the parallel execution of a cellular automaton. The cells are distributed over several threads running on different computers. Each thread is responsible for running several automaton cells. Every thread applies successively to all its cells a 3 steps algorithm : (1)(2) exchange (send and receive) neighbouring information, (3) compute the next cell state. If communications are based on synchronous message passing, the whole system is synchronized at exchange time because of the neighbourhood dependencies. Due to the serial execution of communications, computations and multiple synchronizations, some processors remain partly idle and the achievable speedup does not scale when increasing the number of processors.

Improved performance can be obtained by running communications asynchronously. One can then overlap data exchange with computation. Neighbouring information is received during the computation of the previous step and sent during the computation of the next step. This solution offers improved performances, but still does not achieve a linear speedup. Like in the skeletonization problem, discussed in section 2, the computation load may be highly data dependent and may considerably vary from cell to cell. Furthermore, the parallel application may run on heterogeneous processors inducing a severe load balancing problem. Due to the neighbouring dependencies, cells consuming more computation time slow down the whole system. To reach an optimal solution we need a flexible load balancing scheme.

* Contacts : Marc.Mazzariol@epfl.ch, RD.Hersch@epfl.ch

One solution is to allow each cell to be dynamically remapped during program execution. One or more cells may be displaced from overloaded threads to partly idle threads. Cell remapping requires 3 steps after terminating the computation of the cell to be remapped : (1) notify every thread about the decision to remap a given cell, (2) wait for acknowledgement from all threads and (3) remap the cell. Step (2) ensures that the neighbourhood information for the remapped cell is redirected towards the target thread. In the applications we consider, the overhead for remapping a cell is insignificant compared with the computation time. For the sake of load balancing, we will present in section 4 a trategy for cell remapping.

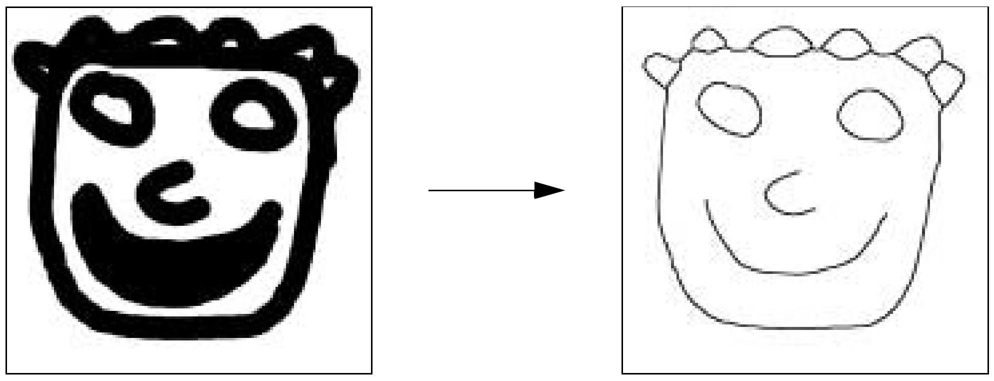
As a typical example of a cellular automaton, we consider the image skeletonization problem[3,4]. Skeletonization requires spatial filtering to be repetitively applied to the image. Each step erodes a thin part of the original image. After the last step, only the image skeleton remains. Skeletonization algorithms require vast amounts of computing power, especially when applied to large images. Therefore, skeletonization application can potentially benefit from the use of parallel processing.

To parallelize image skeletonization, we divide the original image into tiles. These tiles are distributed across several threads. Each thread applies successively the skeletonization algorithm to all its tiles. Threads are mapped onto several processors according to a configuration file. Tiles cannot be processed independently from their neigbouring tiles. Before each computation step, neighbouring tiles need to exchange their borders. In addition, each computation step depends on the preceding step.

Section 2 presents the image skeletonization algorithm. Section 3 develops a parallelization scheme. Section 4 shows how to load balance the application by cell remapping. The performance analysis is presented in section 5.

2 IMAGE SKELETONIZATION ALGORITHM

Image skeletonization consists of extracting the skeleton from an input black and white image. The algorithm erodes repeatedly the image until only the skeleton remains. The erosion is performed by applying a *5x5* thinning filter to the whole image. The thinning filter is applied repeatedly, thinning the input image pixel by pixel. The algorithm ends once the thinning process leaves the image unchanged. Figure 1 shows a skeletonized image.



Original image **Skeletonized image**

Figure 1. Skeletonization algorithm example

Since several skeletonization algorithms exist, let us describe the one providing excellent results[4]. Let $TR(P_1)$ be the number of white to black (0 $\rightarrow$ 1) transitions in the ordered set of pixels $P_2, P_3, P_4, ..., P_9, P_2$ describing the neighbourhood of pixel P_1 (Fig. 2). Let $NZ(P_1)$ be the number of black neighbours of P_1 (black = 1).

P_1 is deleted, i.e. set to background (white = 0) if :

$$\left.\begin{array}{ll} & 2 \le NZ(P_1) \le 6 \\ \text{and} & TR(P_1) = 1 \\ \text{and} & P_2 \cdot P_4 \cdot P_8 = 0 \text{ or } TR(P_2) \ne 1 \\ \text{and} & P_2 \cdot P_4 \cdot P_6 = 0 \text{ or } TR(P_4) \ne 1 \end{array}\right\} \qquad (1)$$

The process is repeated as long as changes occur. This algorithm is highly data dependent. One thinning filter step modifies only small parts of the input image and leaves the major part unchanged. In the next section we take advantage of this fact to improve the algorithm.

P_3	P_2	P_9
P_4	P_1	P_8
P_5	P_6	P_7

Figure 2. Neighbourhood of pixel P_1

2.1 Improvement of the image skeletonization algorithm

To improve the image skeletonization algorithm, we divide the input image into cells (or tiles). The program maintains a list of living and dead cells. A cell is dead if further applications of the thinning filter leave the cell unchanged. A cell is alive if it is not dead. The algorithm applies the thinning filter to each living cell, decides if the cell is still alive and if necessary updates the list of living and dead cells.

This algorithm improves considerably the performance of the skeletonization since a significant part of the image is removed from the computation. What should be the cell size ? A small cell size ensures a fine grain selection of the living and dead parts of the image, but increases the cell management overhead. The cell size should be chosen so as to keep the management overhead time small compared with the average computation time.

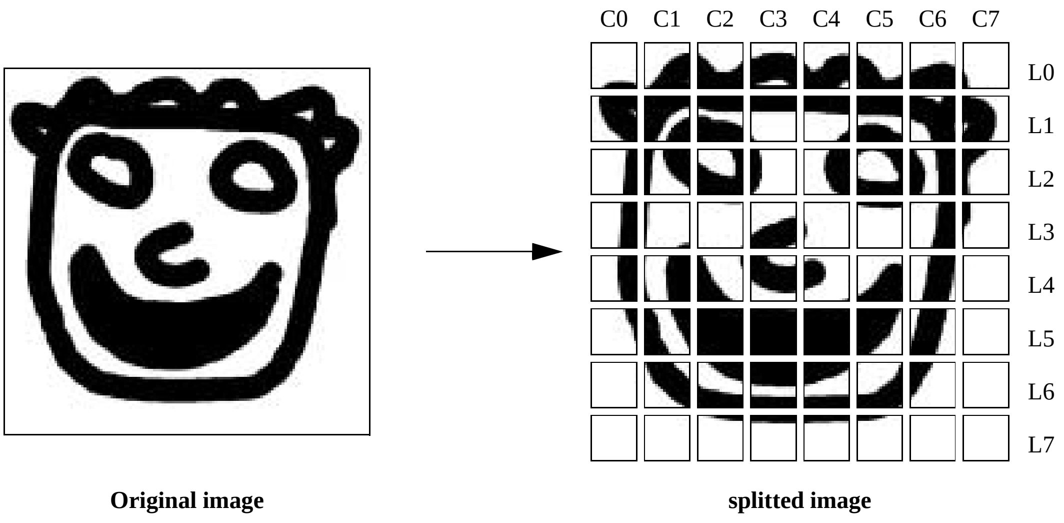


Figure 3. Division of the input image into cells

3 PARALLEL SKELETONIZATION WITH STATIC LOAD DISTRIBUTION

To parallelize the skeletonization algorithm, the cells are uniformly distributed over *N* processing nodes. Each processing node applies repeatedly the thinning filter to all its living cells. Since the cells can not be processed independently from their neighbours, the processing nodes may need to exchange neighbouring information before applying the thinning filter to a given cell. The program ends once all the cells of every processing node are dead. This parallelization scheme ensures that all processing nodes are performing the same task.

Initially the cells are distributed in a round robin fashion over the *N* processing nodes. For example if there are 4 processing units, the cells (Fig. 3) are distributed in row major order modulo the number of nodes (*(L0, C0)->P0, (L0, C1)->P1, ...*).

More generally, if there are *LMax* lines, *CMax* columns and *N* processing nodes, the distribution of the cells over the processing nodes in function of the line and column numbers *(L, C)* is given by :

$$NodeIndex(L, C) = (L \cdot CMax + C) \text{ Modulo } N \tag{2}$$

In the parallel algorithm, the overhead for the exchange of information between neighbouring cells increases since communication and synchronization is needed between processing nodes responsible for adjacent cells. The parallel program requires therefore larger cell sizes.

To develop the parallel application, we use the Computer-Aided Parallelization (CAP) framework, which allows to manage the neighbourhood dependencies and the data flow synchronization. The CAP Computer-Aided Parallelization framework[5,6] is specially well suited for the parallelization of applications having significant communication and I/O bandwidth requirements. Application programmers specify at a high level of abstraction the set of threads present in the application, the processing operations offered by these threads, and the flow of data and parameters between operations. Such a specification is precompiled into a C++ source program which can be compiled and run on a cluster of distributed memory PCs. A configuration file specifies the mapping between CAP threads and operating system processes possibly located on different PCs. The compiled application is executed in a completely asynchronous manner : each thread has a queue of input tokens (serializable C++ data structures) containing the operation execution requests and their parameters. Network I/O operations are executed asynchronously, i.e. while data is being transferred to or from the network, other operations can be executed concurrently by the corresponding processing node. If the application is compute bound, in a pipeline of network communication and processing operations, CAP allows to hide the time taken by the network communications. After initialization of the pipeline, only the processing time, i.e. the cell state computation, determines the overall processing time.

Each processing node contains two threads (*IOThread, ComputeThread)* running in one shared address space. The address space stores the data relative to all its cells. The *IOThread* runs the asynchronously called *ReceiveNeighbouringInfo* and *SendNeighbouringInfo* functions. The *ComputeThread* runs the *ComputeNextCellStep* function. The *ReceiveNeighbouringInfo* function receives the neighbouring information from the other cells and puts it in the local processing node memory. The *SendNeighbouringInfo* function sends the neighbouring information from the local cells to the neighbouring cells. The *ComputeNextCellStep* applies the skeletonization filter to the given cell. Before starting the computation, the *ComputeNextCellStep* waits for the required neighbouring information and until the current neighbouring information has been sent. Once the filter is applied, the *ComputeNextCellStep* determines if the cell is still alive, and if necessary updates the list of living and dead cells.

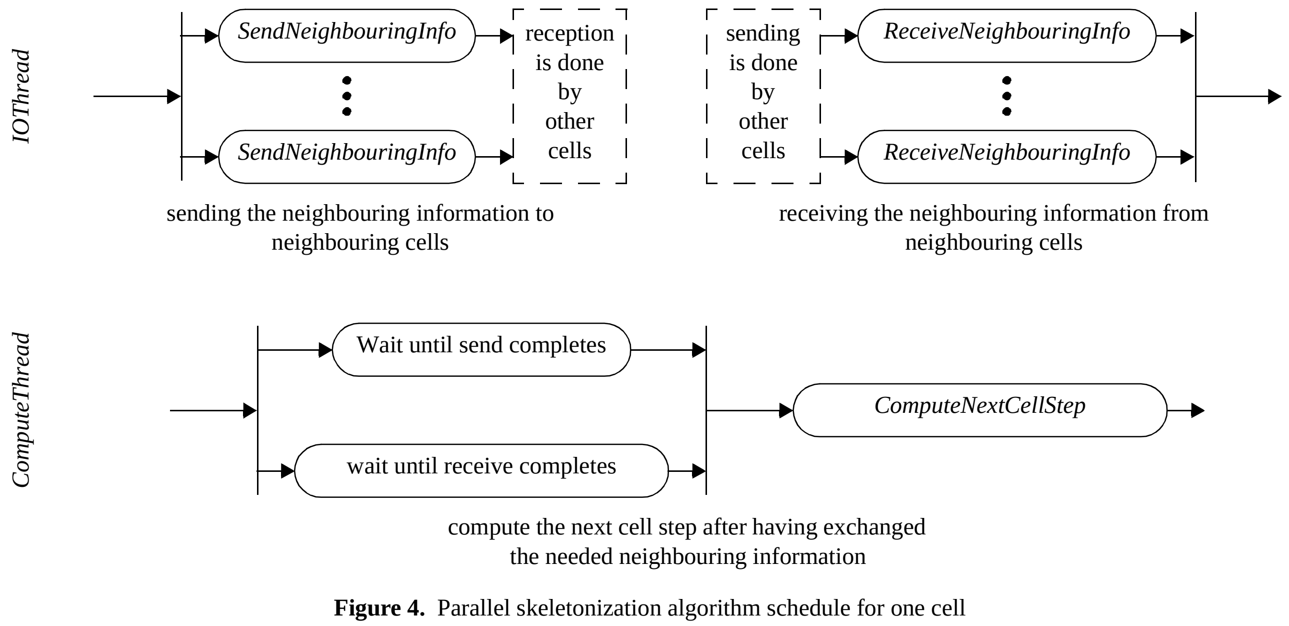

IOThread
SendNeighbouringInfo
SendNeighbouringInfo
reception is done by other cells
sending is done by other cells
ReceiveNeighbouringInfo
ReceiveNeighbouringInfo
sending the neighbouring information to neighbouring cells
receiving the neighbouring information from neighbouring cells
ComputeThread
Wait until send completes
wait until receive completes
ComputeNextCellStep
compute the next cell step after having exchanged the needed neighbouring information

Figure 4. Parallel skeletonization algorithm schedule for one cell

Each processing node runs the same schedule. The schedule comprises a main loop executing for all the living cells of the local address space one running step. A running step consists of executing asynchronously the *SendNeighbouringInfo*, *ReceiveNeighbouringInfo* functions and in parallel the *ComputeNextCellStep* function for the given cell. Figure 4 shows a schematic view of the schedule. The CAP tool allows the programmer to specify this schedule by appropriate high-level language constructs[6].

When the program starts, all the cells are at step zero. Each time the thinning filter is applied to a cell, the cell step is incremented by one. Because of the neighbouring dependencies, the differences between the step of a given cell and of its neighbours is at most one. Therefore, during the program execution, some cells are waiting for their neighbours to perform the computation of the next step. If all the cells of a processing node are waiting, the processor becomes idle, reducing the overall performance. To avoid as much as possible such a situation, the parallel algorithm is improved by computing first the cell with the smallest step value on each processing node.

While some cells are sending their neighbouring information or waiting for the reception of neighbouring information, other cells could potentially keep the processor busy. This argument fails if there is just one cell per processing node or if the cellular automaton topology implies that every cell is depending on all other cells. In order to run computations in parallel with communications, one may partially compute a cell without knowing the neighbouring information. Cell computation may start while receiving the neighbouring information from other cells[7].

If the total computation load is evenly distributed over the processing nodes, the parallel algorithm can potentially keep all the processors busy. However, in the case of the skeletonization algorithm, the computation time is highly data dependent. To keep all processors busy, we need to balance dynamically the computation load.

4 DYNAMIC LOAD BALANCED PARALLEL SCHEME

For load balancing, we need to remap the cells during program execution. In order to migrate a cell from one address space to another, we need to maintain the load (or the inverse of the load, i.e. an idle factor) for each processing node. A simple way of computing the load is presented here. Let *A* be the average step value of all cells

$$A = \frac{1}{NumberOfCells} \sum_{AllCells} CellStepValue \tag{3}$$

For each processing node, we compute an *IdleFactor* by adding the signed differences between the processing node cell step values and the average step value *A*. When the processor is idle, the *IdleFactor* is set to a *MaxIdleFactorValue* minus the number of cell in the processing node[†].

$$IdleFactor = \begin{cases} \displaystyle\sum_{AllCells} (CellStepValue - A), & \text{if the processor is not idle} \\ MaxIdleFactorValue - NumberOfCells, & \text{if the processor is idle} \end{cases} \tag{4}$$

A negative *IdleFactor* indicates that the cell step values of the corresponding processing node are behind the other processing nodes. A processing node with a strongly negative *IdleFactor* is overloaded and slows other processing nodes which run its neighbouring cells. A positive *IdleFactor* indicates that the corresponding processing node is ahead of the others. The processor of such a processing node may soon become idle since the neighbouring dependencies with the cells of other processing nodes will put it in a wait state. To balance the load, a cell from the processing node having the most negative *IdleFactor* should be remapped to the processing node having the largest positive *IdleFactor*. The *IdleFactor* is evaluated periodically, every time a new cell migration is performed. Between two cell migrations, a specific *IntegrationTime* allows the system to take advantage of the previous cell migration.

In order to compute the *IdleFactor*, one thread, called *MigrationThread*, is added in each processing node. Periodically, a new token is generated and traverses all the *MigrationThreads* of every processing nodes. The token is generated in processing node *P0*, then it visits all processing nodes in the order : *P1, P2, ..., PN* and back to *P0*. The migration token makes three full traversals in order to allow the parallel system to decide which cell to remap. During the first traversal, the migration token collects the number of living cells of each processing node and the sum of their step values. This information is distributed to all the processing nodes during the second traversal. During this same traversal, every node computes its *IdleFactor*. This *IdleFactor* is collected by the migration token and distributed over all processing nodes during the third and last traversal. Then every node decides in a distributed manner which processing nodes are involved in the migration.

† A processing node having no cell to process should have a higher *IdleFactor* than processing nodes with cells waiting for neighbouring information.

The processing node from which the migration starts, migrates the cell with the smallest step value. In order to perform the migration, the *IOThread* broadcasts to every processing node the migration cell destination and waits for acknowledgment. Once the *IOThread* receives acknowledgments from every processing node, no further information for the migrating cell will be received on the current processing node. The *IOThread* sends the cell data and all the previously received neighbouring information to the destination processing node. The migration is done. The time period between each migration cycle is set by the *IntegrationTime* parameter. If the *IntegrationTime* is too short, the processing nodes will waste time for performing useless cell migrations. In the worst case, a too short *IntegrationTime* results in migrating all the cells of a processing node leaving it without any cell. If the *IntegrationTime* is too large, then the processors may become idle before receiving a migrated cell.

Experiments show that it is difficult to find an a good *IntegrationTime*. In order to improve the cell remapping strategy, let us introduce the notion of *stability*. A processing node is *stable* if the difference between the *CellStep* values within a processing node is at most one :

$$\text{Processing node is } stable \Leftrightarrow Max(CellStepValue) - Min(CellStepValue) \leq 1 \qquad (5)$$

A processing node is *unstable*, if it is not *stable*. Since the *ComputeNextCellStep* function processes first the cells with the smallest *CellStep* value, the *stable* state is a permanent state if no cell migration occurs. Without cell migration, each *unstable* processing node will sooner or later reach the *stable* state. The migration cell emission and receiving processing nodes are determined by the *IdleFactor*. In order to improve the cell migration strategy, we take into account two migration rules avoiding in some special cases the migration of cells. We do not migrate the cell if the cell receiving processing node is in an unstable state. This rule avoids to carry out consecutive migrations to the same cell receiving processing node. We also do not migrate if the migrated cell will leave the receiving processing node in a stable state. This rule avoids migration if the receiving processing node has no major advance compared with the emission processing node. These two rules are not applied if the receiving processing node is detected to be idle. The *stability* information is exchanged in the same way as the *IdleFactor*.

5 PERFORMANCE MEASUREMENT

The performance measurements were carried out on three input images : a balanced input image, a highly unbalanced input image and a slightly unbalanced input image. The balanced input image (Fig. 5) consists of a repetitive pattern ensuring an evenly distributed computation load. In the highly unbalanced input image (Fig. 6), according to the cell distribution (eqn. 2), the non-empty cells are distributed unevenly across the processing nodes. One of two processing nodes receives empty cells which require only one computation step. The slightly unbalanced input image (Fig. 7) is an intermediate case between the balanced and the highly unbalanced input images. The balanced and unbalanced input images are of size 2048x2048 pixels (8 bits/pixel) and splitted into 16x16 cells, incorporating 128x128 pixels. The slightly unbalanced input image is of size 1024x1536 pixels (8 bits/pixel) and splitted into 8x12 cells, incorporating 128x128 pixels..

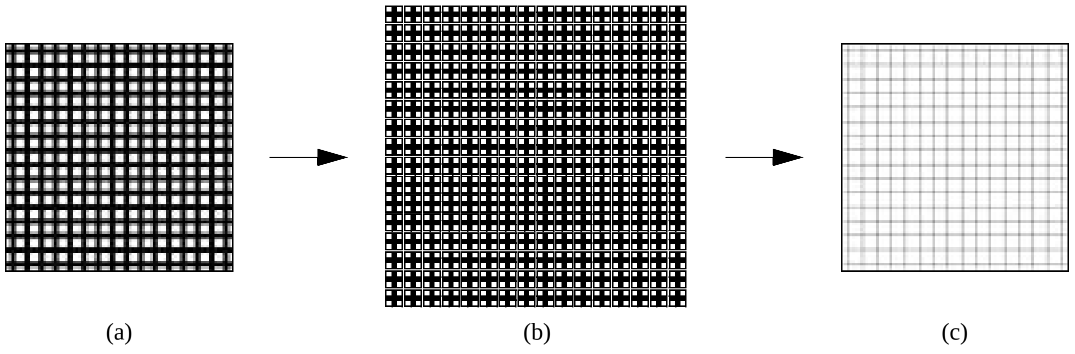


Figure 5. Example (a) of a balanced input image, (b) after segmentation into cells and (c) after skeletonization

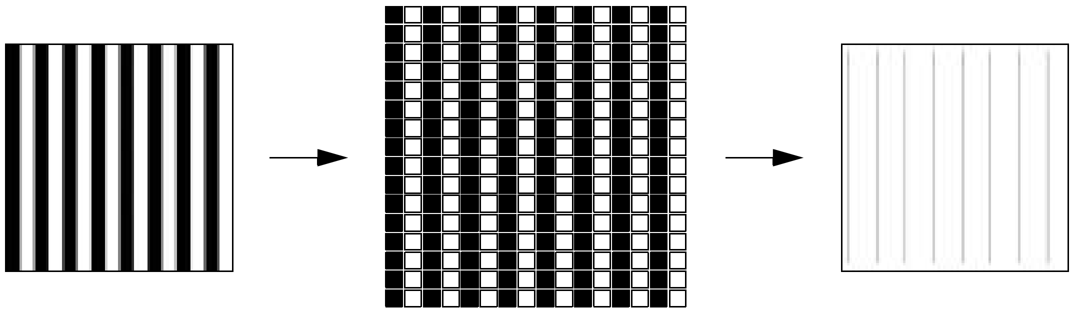


Figure 6. Example (a) of a highly unbalanced input image, (b) after segmentation into cells and (c) after skeletonization

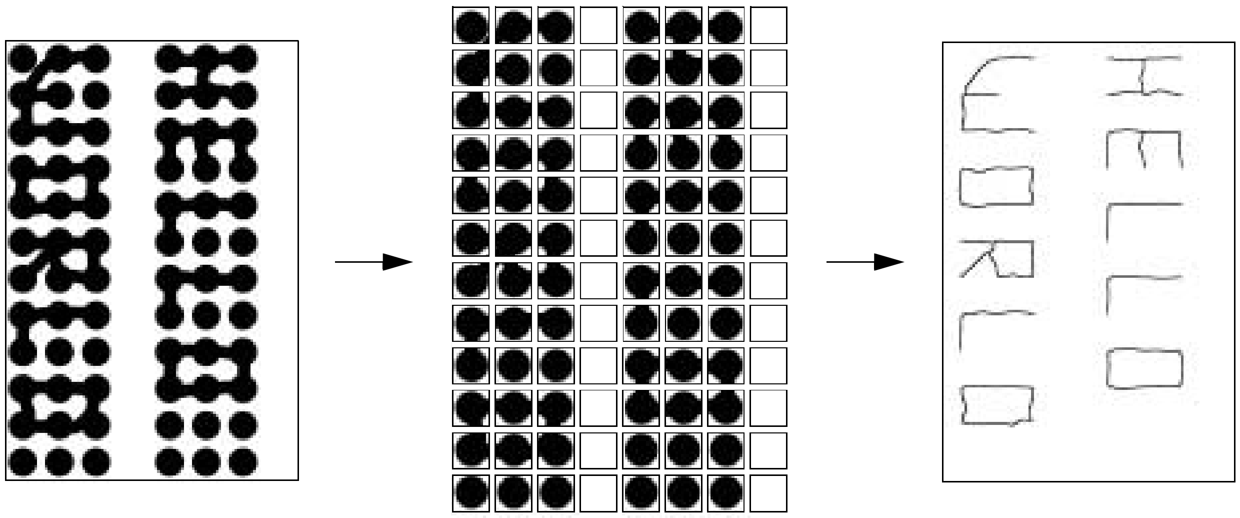


Figure 7. Example (a) of a slightly unbalanced input image, (b) after segmentation into cells and (c) after skeletonization.

The *N* processing nodes are all Bi-Pentium Pro 200 MHz PCs running under WindowsNT 4.0. They are connected through a Fast Ethernet switch.

Figure 8 shows the speedup for the balanced input image, the highly unbalanced input image and the slightly unbalanced input image. The measurements are done for 1 to 10 processors. The performances of the algorithms with and without the cell migration scheme are compared. The measurements for the dynamically load balanced algorithm are done with an *IntegrationTime* of 0.5 sec between each cell migration.

In the case of the balanced input image, there is no significant performance difference between the two algorithms. For such an input image, the cell migration is useless since the load is perfectly balanced between the processing nodes. The results show that the overhead induced by the management of the cell migration is low. The parallelization does not provide a linear speedup because the neighbouring information exchange consumes processing resources (CPU power for the TCP/IP communication protocol).

In the case of the highly unbalanced input image, the performances are considerably improved by dynamic load balancing. Without cell migration the efficiency (speedup/*N*) is approximately 50% since one processor of two becomes idle. Implementing the cell migration allows the parallel program to reach approximately the same speedup with a balanced or an unbalanced input image.

In the case of the slightly unbalanced input image, the performances are improved by using the dynamically load balanced algorithm. Since in the input image, about one out of four cells is empty, the theoretically maximal performance improvement factor obtained by the dynamically load balanced algorithm is $4/3 = 1.33$. Our algorithm reaches a performance improvement factor of 1.22. The difference is due to the fact that some time is needed until the system reaches a load balanced state.

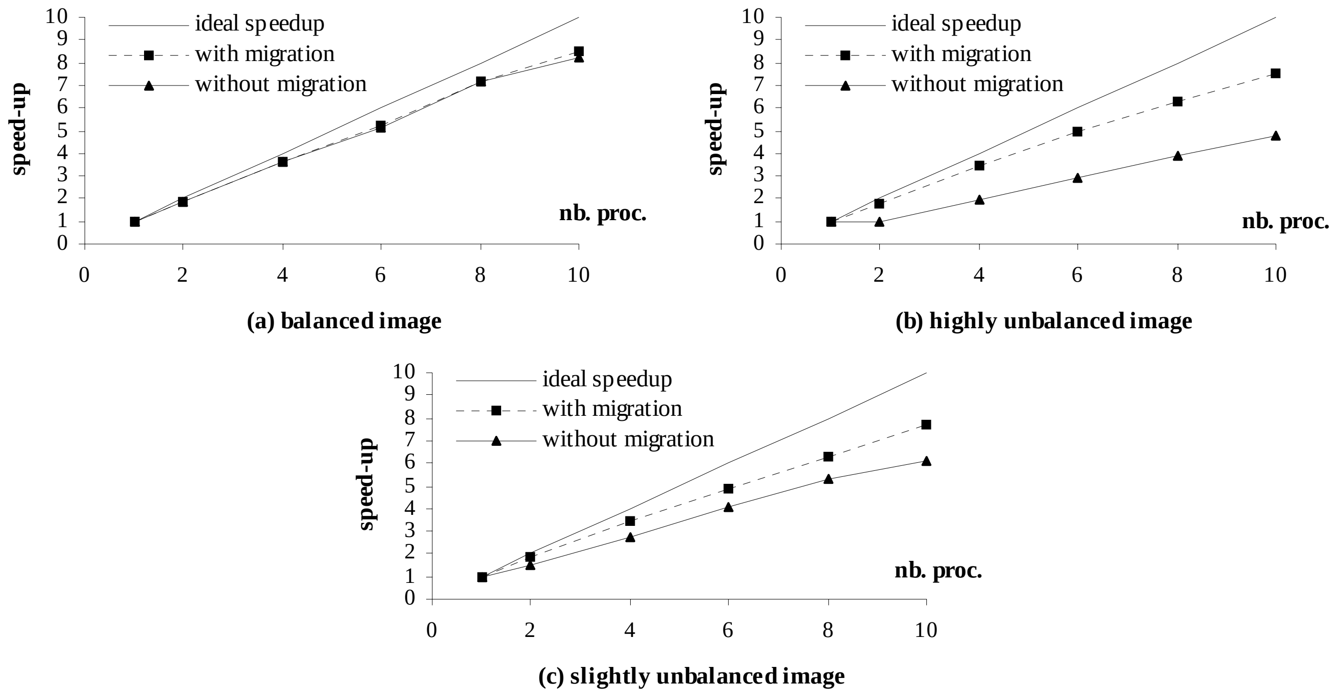


Figure 8. Speedup for (a) the balanced input image, (b) the highly unbalanced input image and (c) the slightly unbalanced image.

Globally, performances are improved by dynamic cell remapping. No improvement is expected in the case of a well balanced input image. A major improvement is achieved in the case of an unbalanced input image. The presented results closely match the expected results.

6 CONCLUSIONS

We are interested in the parallelization of cellular automata. Our experiment is based on a particular image skeletonization method. We have developed a parallellization algorithm which can be easily applied to other cellular automata. We explore two parallelization methods, one with a static load distribution consisting in splitting the cells over several processing nodes and the other with a dynamic load balancing scheme capable of remapping cells during the program execution. Performance measurements show that the cell migration doesn’t reduce the speedup if the application is already load balanced. It improves the performance if the parallel application is not well balanced.

Cellular automata have a wide range of applications : matrix computation, state machines, Von Neumann automata, etc. Many problems can be expressed as cellular automata. Developing from the scratch a custom parallel application requires a large effort. This paper shows the possibility of developing first a generic parallel cellular automaton and, on top of it, parallel applications making use of the cellular automaton program interface. This approach reduces the programming effort without loosing efficiency.

ACKNOWLEDGEMENT

We thank Gilles Ritter for having written the first version of a simple parallel skeletonization algorithm. This research is supported by the Swiss SPP-ICS research program, grant 5003-51332.

REFERENCES

1. Moshe Sipper, Evolution of Parallel Cellular Machines, LNCS 1194, Springer Verlag, 1997
2. Toffoli and Margolus, Cellular Automata Machines : A New Environment for Modeling, MIT Press, 1987
3. Robert J.Schalkoff, Digital Image Processing and Computer Vision, Wiley, 1989, p. 320
4. Anil K. Jain, Fundamentals of Digital Image Processing, Prentice Hall, 1989, p. 382
5. V. Messerli, R. D. Hersch, “Parallelizing I/O intensive Image Access and Processing Applications”, IEEE Concurrency, Vol. 7, No.2, April/June 1999, 28-37
6. B. Gennart, R. D. Hersch, “Computer-Aided Synthesis of Parallel Image Processing Applications”, Proceedings Conf. Parallel and Distributed Methods for Image Processing III, SPIE Int. Symp. on Opt. Science, Denver, July 1999, SPIE Vol-3817, 48-61
7. B. Gennart, R. D. Hersch, “Synthesizing parallel imaging applications using the CAP Computer-Aided Parallelization tool”, Proceedings Conf. Storage & Retrieval for Image and Video Databases VI, San Jose, Jan. 98, SPIE Vol-3312, 446-458