Parallelisation of Wave Propagation Algorithms for Odour Propagation in Multi-Agent Systems

Eugen Dedu[1], Stéphane Vialle[1], and Claude Timsit[2]

[1] Supélec, 2 rue Édouard Belin, 57070 Metz, France
dedu@ese-metz.fr, Stephane.Vialle@supelec.fr
[2] University of Versailles, 45 avenue des États-Unis, 78035 Versailles Cedex, France
Claude.Timsit@prism.uvsq.fr

Abstract. One of the algorithms used in multi-agent systems is based on the wave propagation model. This article discusses some sequential (recursive, iterative, and based on distance) and parallel methods (frontier exchanging, domain decomposition changing, private environments, and mutex-based) to implement it. The mixing between these sequential and parallel methods is also shown, and the performance of some of them on two shared-memory parallel architectures is introduced.

Keywords: parallel algorithms, wave propagation model, multi-agent systems.

1 Introduction

The situated multi-agent systems allow to simulate populations of agents. An agent is defined as an entity found in an environment, which senses it by its percepts and act upon it through its effectors [2]. One of the goals of the simulation of populations of agents is to discover emergence: individual behaviours of agents which give efficient global behaviours. The behaviour uses agent percepts. One of the percepts used in multi-agent systems is the "odour", simulated by a value in each square proportional to its intensity [8]. These values, called potential in the following, are spread by resources through the environment. A square may be influenced by several resources. For our model, we have chosen that, when a square is influenced by several resources, it receives the value of the *strongest* potential (another hypothesis is taken in [8], where overlapping potentials add). The potentials allow agents to find the way to resources in environment. As the potential propagation represents in some cases a great part of the execution time of the simulator [8], we studied their parallelisation.

This article is divided as follows. Firstly, we describe the wave propagation model, used to propagate the potential of resources. Secondly, some sequential methods to implement it are presented. General parallelisation methods and their mixing with sequential ones are presented afterwards. Finally, the performance of two parallel implementations, based on one sequential method, is described.

2 Wave Propagation Model

The potential propagation consists of spreading decreasing potentials from a resource. It allows agents to find the way to resources by following increasing potential. We have made the assumption in our model that the potential in each square is equal to the potential of the resource minus the distance to it: $p(d) = p_R - d$. We use a 4-connectivity of squares. In an environment without obstacles, the distance between two squares is simply $|dy| + |dx|$. However, in the presence of obstacles, this formula is no longer appropriate. The wave propagation model allows to surpass obstacles (Figure 1).

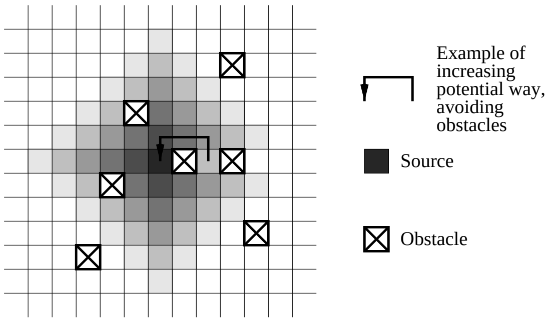

Fig. 1. Illustration of the potential spread by a resource.

3 Sequential Methods

3.1 Recursive Potential Computing

These methods create the potential fields by propagating successively the potential of each resource. The propagation of each resource is done recursively.

Recursive with depth-first propagation. In this method, the propagation starts from each resource and is recursively spread on the environment through a depth-first mechanism while decreasing the value of the current potential ($P(S)$ is the potential in square S):

```
clear (set to 0) the potential of all the squares
**for all** square $S$ containing a resource **do**
  prop-square $(S, P(S))$

**procedure** prop-square (square $S$, potential $p$)
**if** $P(S) < p$ **then** {$S$ needs to be processed}
  $P(S) \leftarrow p$
  **if** $p > 1$ **then**
```

```
for all neighbour $N$ of $S$ do
  if $S$ is not an obstacle then
    prop-square ($N$, $p-1$)
```

The key point of this method is that the call to recursive function `prop-square` gives a *depth-first* propagation. The depth-first recursion is simple to implement, since it is automatically provided by modern programming languages. However, this is a case where the depth-first search is not efficient. The reason is that the potential of some squares is successively updated. Figure 2 presents such an example, where the neighbours are taken in N, E, S, W order (potential of resource is 4). The potential of the square at right of the resource receives two values: It is firstly set to 1, then to its correct value 3. Moreover, an update of the potential of a square generates also a recursive update of all its neighbours[1]. Generally, the greater the potential of the resource, the bigger the number of squares which receive multiple updates, and the greater the number of updates which are needed in average for each square.

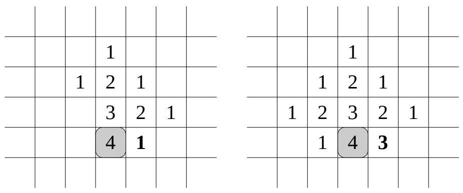

Fig. 2. Two steps during the depth-first propagation of a resource. The square at the right of the resource updates its potential twice.

Recursive with breadth-first propagation. We have implemented a similar method, which avoids the successive unuseful updates (as given by the previous method) by doing a *breadth-first* propagation. The breadth-first search mechanism and the recursion were simulated with a (FIFO) queue that stores the elements used in recursion.

Because this method uses the breadth-first search, each square is modified only once. As such, the number of updates of squares is equal to the number of squares in the potential field.

3.2 Iterative Potential Computing

These methods sweep several times the whole environment, updating each square when necessary.

[1] For the sake of precision, the number of times each square (except the squares in N direction) is updated is $\left\lfloor \frac{k+1}{2} \right\rfloor$, where k is the final potential of the square.

Iterative with fixed potential. This method is presented by Bouton [1], who has worked in our team. It works by firstly putting the potential of each resource in its square. Then, during the first iteration, all the environment is swept in order to find all the squares containing the greatest potential p. Each time a square with potential p is found, all its neighbours having potential less than $p - 1$ are given a potential of $p - 1$. During the second iteration, all the squares with potential $p - 1$ are found and their neighbours with a lower potential are given potential $p - 2$. The iterations continue until the potential 1 is reached. At this step, the propagation is completely finished on all the environment.

This method has the advantage to be simple, so it does not add execution overheads. Nevertheless, the high disadvantage of this method is that during each step all the environment is swept, which leads to a lot of unnecessary processed squares.

Iterative with variable potential. This is similar to the previous method. The difference between them is that, during each step, instead of processing only squares with a given potential p, this method compares each square with its neighbours, updating it if one of them has a higher potential.

3.3 Distance-Storing Methods

These methods are characterised by the fact that each square stores not potentials, but *distances* to resources. This is possible because the place of resources is fixed during the simulation.

Distance-storing of all influent resources. In this method, each square stores the identifier (a unique number) of every resource which can influence its potential during the simulation, and the distance to it. When the potential of a square needs to be known, the influence of each resource on it can be simply calculated by using the distances it stores and the actual potential of each concerned resource. Then the strongest potential can be chosen.

This method has the advantage that the potential can be computed on demand (only for the needed squares), which can be very fast. As an example, if agents are found on 1% of the squares, then the potential of a maximum of 5% of the squares is calculated (the square itself and its four neighbours). Another advantage of this method is that the computations for each square are independent, so no special parallelisation method (to avoid parallelisation conflicts) is needed. Nevertheless, its drawback is that it has a high memory requirement, because it stores in each square information about every resource which can influence it.

Distance-storing of the most influent resource. This method looks like the previous one. However, instead of storing in each square the identifier of *all* the resources which can influence it, it stores the identifier of only the most influent one. The most influent resource is the resource which gives the potential of the square, i.e. it gives the maximum potential in the case of overlapping fields.

This method is more difficult to implement than the previous one. However, since only one resource is stored in each square, it needs less memory space. In conclusion, it seems to be a very good trade-off between memory requirements and execution time. Nevertheless, it seems to be very difficult to find an algorithm which updates the resource frontiers when resource potentials evolve.

4 Parallelisation Methods

This section deals with parallelisation methods which can be applied to the sequential algorithms described above. Some of the combinations between the parallelisation methods and the sequential algorithms are possible without any modification, others are inefficient, while others are not possible. Their mixing is presented in Table 1 and will be detailed below.

Table 1. Mixing between the parallelisation methods and the sequential ones.

Parallelisation\Sequential method	Recursive	Iterative	Distance-all	Distance-most
Fixed domain partitioning	ok	ok	first stage only	ok
Changing domain partitioning	ok	ok	first stage only	ok
Processor-private environments	ok	inefficient	inefficient	ok
Mutex-based	ok	ok	no	ok

4.1 Fixed Domain Decomposition

This is the classical domain decomposition parallelisation [4]. The basic principle is that each processor is affected to a different domain. The number of domains is equal to the number of processors, and each processor is bound to a distinct domain in our environment. We have used a horizontal decomposition, where each domain has the same number of lines[2] of environment.

The complete propagation is done in three stages (Figure 3). The first stage propagates the potential of all the resources in each domain *separately*, setting the correct potential on each square of its domain. This can be done with any of the sequential methods, according to Table 1. This stage of propagation is sufficient when the sequential method based on storing all the influent resources is used.

The second stage copies the frontiers to a memory accessed by other processors, called *buffer* in the following. No synchronisation point is needed, since there is no sharing conflict: each processor reads and writes its own data (domain and buffers).

The third stage repropagates in each domain separately the potentials of all the frontiers. Four points need to be discussed in this stage, presented in the

[2] Or differing by 1 line if the number of lines is not divisible by the number of domains.

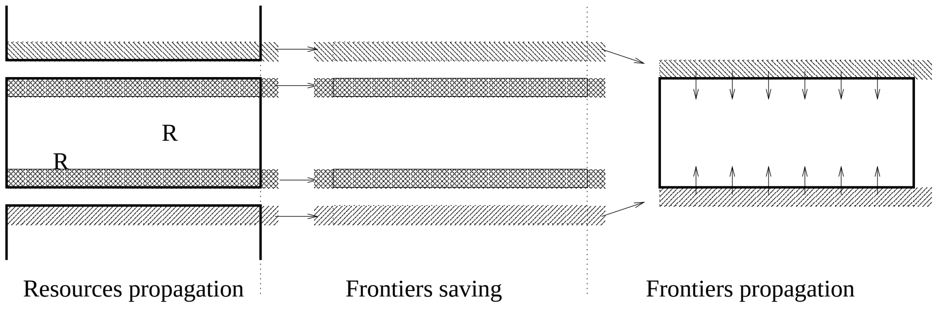

Fig. 3. For each processor, the propagation is completely done in three stages: resources propagation, frontiers saving (only 2 frontiers in this figure), and frontiers repropagation, the last two stages being repeated several times.

following. Firstly, a synchronisation point is mandatory to cope with the sharing of the buffers.

The second point involves the main part of this stage: the repropagation of the frontiers. This is similar to the first stage (propagation), except that the propagation starts from all the points of the domain frontiers (and not from resources, as some of the propagation methods).

Thirdly, the second and the third stages are repeatedly executed until no change of potential is done on frontiers during the third stage. At this moment, the propagation is entirely done in all the environment.

Figure 4 (left case) presents an example where two repropagations are necessary. During propagation, the obstacles prevent the square X to receive the potential from resource R. A first repropagation allows intermediate squares to receive correct potential values, and only the second repropagation can put the right potential into square X. The other two examples in Figure 4 present cases when a domain is interposed between resource R and square X.

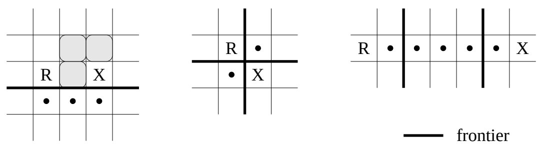

Fig. 4. Examples where two repropagations are needed for square X to receive the correct potential value from resource R (suppose the only resource influencing X is R).

The fourth point involves the number of synchronisations needed during this method. Since each buffer is read by a neighbour processor at the middle of the third stage, and written afterwards by its own processor (in the second stage), another synchronisation point seems to be needed. Nevertheless, the double-buffer technique avoids the use of a second synchronisation:

```
write own frontiers
i = 0 {the variable i is private to each processor}
repeat
   write own buffers[i]
   synchronisation point
   read neighbours' buffers[i]
   i = 1 - i {choose the other set of buffers}
   write own frontiers
until no change of potential on any square of the frontiers
```

4.2 Changing Domain Decomposition

The fixed domain decomposition method needs several repropagations to complete the potential spreading in the environment. The purpose of the changing domain decomposition is to reduce the number of repropagations.

The distinctive feature of this method is that the domain decomposition changes when the frontiers are repropagated. Figure 5 presents such a case, with 4 processors used by the propagation and 3 processors used during the repropagation. The new frontiers are now located at some distance from the old ones. If the distance between the old and the new frontiers is larger than the potential field length of any resource, the potentials of the new domain frontiers do not change, thus no synchronisation point is necessary.

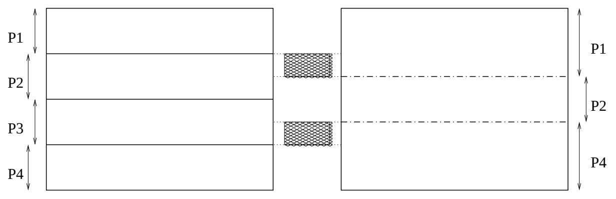

Fig. 5. The repropagation changes the decomposition in order to move off the frontiers.

A disadvantage of this method is that the cache is not well exploited. Since a lot of squares are processed by different processors during propagation and repropagation, cache conflicts appear [5], degrading the performance.

4.3 Processor-Private Environments

As a first step (Figure 6), each processor processes a part of the resources, and updates its potential field on a *whole* processor-private copy of the environment. After that, as the second step, the environment is updated using the processor-private copies of the environment: each square of the environment receives a potential equal to the maximum potential of the corresponding squares of the processor-private environments. This solution has higher memory requirements (for the private environments) and leads to a lot of cache misses for the second step, since each processor reads private environments.

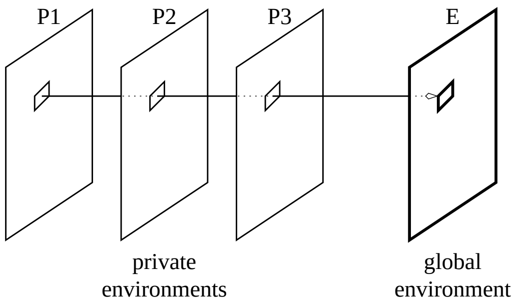

Fig. 6. Example of data decomposition for the wave propagation algorithm (3 processors).

4.4 Mutex-Based Parallelisation

In this method, each processor propagates the potential of a part of the resources (data decomposition). To solve the concurrent access to squares influenced by several resources, a mutex (variable used to ensure a *mut*ual *ex*clusion region) per square or per group of squares is used. This leads to numerous and expensive mutex operations.

5 Performance

We have implemented two parallelisation methods: fixed domain partitioning, and processor-private environments. Four sequential implementations were taken into account: recursive with depth-first and breadth-first propagation, and iterative with fixed and variable potential. As shown in Table 1, it is inefficient to use processor-private environment with iterative methods. Therefore, we chose to use the recursive breadth-first method for the two parallelisation methods above.

The programming language used was the `C` language. The SMP machine was a Workgroup 450 Sun server[3] with 4 processors running Linux, the compiler used was gcc (GNU Compiler Collection), and the parallel library was Posix threads [6], linux threads[4] implementation. The DSM [7] machine was an Origin 2000 [3] with 64 processors running Irix, the compiler used was its native one, MIPSPro, with Irix native multi-threading library.

[3] `http://www.sun.com/servers/workgroup/450`
[4] `http://pauillac.inria.fr/~xleroy/linuxthreads`

For these tests, an environment with 512x512 squares was used, containing 10% obstacles. The number of resources was 1% of the squares, and their potential was 16.

For each test, four executions were done, and the execution time of the slowest processor in each execution was kept. The average time among these was taken into account for experimentation results. The execution time of the propagation part in sequential is 350ms for DSM machine, and 650ms for SMP machine. The speed-ups are presented in Figures 7 and 8. We can see that the speed-ups on both machines are similar. Also, for the given parameters of simulation, the fixed domain partitioning gives better performance than processor-private environments. The reason can be the cache misses which appear in the latter method.

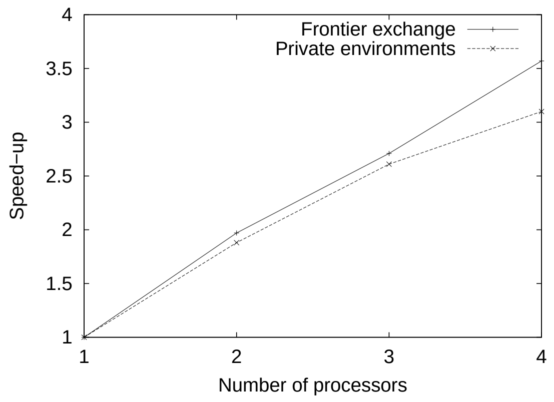

Fig. 7. The speed-up on an SMP machine.

6 Conclusions

This article has introduced four parallelisation and six sequential algorithms of the wave propagation model. We have experimented two parallelisation algorithms, using an efficient and compatible sequential algorithm for the processing of their sequential part. We have obtained good speed-ups up to four processors on both SMP and DSM architectures: speed-up greater than 3 on 4 processors. But performance decreases using more processors. Therefore, our future work will be guided to the following directions:

– Optimising the implementations, such as reducing useless cache misses.
– Implementing and evaluating algorithms with higher memory requirements, such as the method based on distance-storing of all influent resources.
– Implementing and evaluating other parallelisation methods.

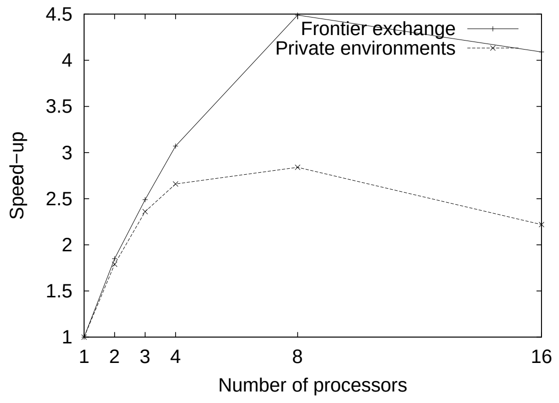

Fig. 8. The speed-up on a DSM machine.

But an original and promising direction seems to be the tolerance of minor errors in the potential propagation, which would decrease the execution times without affecting agent performance.

Acknowledgements

Support for this research is provided in part by a grant from *Région Lorraine* (France), and access and support to Origin 2000 supercomputer are provided by *Charles Hermite Centre* (France).

References

1. L. Bouton. ParMASS : un simulateur de systèmes multi-agents parallèle. Master's thesis, Université de Nancy-I, Supélec, LORIA, June 1998.
2. J. Ferber. *Les systèmes multi-agents. Vers une intelligence collective.* InterEditions, 1995.
3. J. Fier. Performance tuning optimization for Origin2000 and Onyx. Available at `http://techpubs.sgi.com/library/manuals/3000/007-3511-001/html/O2000Tuning.0.html`.
4. I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley Publishing Company, 1995.
5. M. D. Hill and J. R. Larus. Cache considerations for multiprocessor programers. *Communications of the ACM*, 33(8):97–102, Aug. 1990.
6. B. Nichols, D. Buttlar, and J. P. Farrel. *Pthreads Programming.* O'Reilly & Associates, Sept. 1996.
7. J. Protić, M. Tomašević, and V. Milutinović. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology*, 4(2):63–79, Summer 1996.
8. G. M. Werner and M. G. Dyer. BioLand: A massively parallel simulation environment for evolving distributed forms of intelligent behavior. In H. Kitano and J. A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 316–349. MIT Press, 1994.