Comparison of OpenMP and Classical Multi-Threading Parallelization for Regular and Irregular Algorithms

Eugen Dedu, *dedu@ese-metz.fr*
Stéphane Vialle, *vialle@ese-metz.fr*
Supélec, Metz campus
2 rue Edouard Belin
57070 Metz, France

Claude Timsit, *Claude.Timsit@supelec.fr*
Supélec, Gif-sur-Yvette campus
Plateau de Moulon
91192 Gif-sur-Yvette, France

Abstract

The new emerging Distributed Shared Memory architecture promises to be more scalable than Symmetric Multiprocessor architecture, and leads to a regain of interest for parallel shared-memory programming paradigms. This paper compares two such important paradigms: classical multi-threading and multi-threading based on compiler directives (with OpenMP). Several implementations of regular and irregular algorithms, taken from artificial intelligence field, were made on an SGI-Origin2000 (a DSM architecture) and compared both in terms of development time *and of* execution time. *Finally, we identify the most appropriate paradigm for each kind of algorithm.*

Keywords: *Parallelism, Threads, OpenMP, Regular algorithms, Irregular algorithms, Multi-agent systems, Artificial neural networks.*

1. Motivations and objectives

This paper compares parallel programming paradigms for regular and irregular problems. To illustrate this, two classic problems of Artificial Intelligence (AI) field have been chosen: an artificial neural network (ANN) as a regular computation problem, and a multi-agent system (MAS) as an irregular one. They are both composed of multiple autonomous entities that run concurrently, and are time-consuming in many cases. Their natural parallelism could be exploited to run on parallel computers, aiming at the decrease of their execution times or the increase of their data size. After the description of available parallelization paradigms on shared-memory parallel computers, some of these models and their algorithms will be briefly introduced in sections 3 and 4.

Today, Distributed Shared Memory (DSM) architectures [4, 10] exist, and are scalable up to several hundreds of processors, as Origin2000 of SGI. Parallel computers supporting shared-memory paradigm are no more limited to a small number of processors, as classic manufactured Symmetric Multiprocessors (SMP) architectures (see [3]). We chose to explore the shared memory parallel programming, as DSM architectures seem to have a promising future.

Three main shared-memory parallel paradigms exist: communicating processes, classic multi-threading and multi-threading based on compiler directives (for example OpenMP). They have different development times and execution performance. Among these three paradigms we search for the most adapted for regular and irregular algorithm implementations. Threads automatically share a global address space and must explicitly allocate the thread-private space, while processes keep their private address space and must explicitly allocate shared memory to share data [8]. Thus, multi-threading usually appears to be more appropriate to parallelize applications doing computations on shared data. Because of this we will ignore processes and we will compare only classic multi-threading and OpenMP.

2. Programming paradigms for shared-memory multiprocessors

2.1. Multi-threading programming evolution

The best way to use the shared-memory computers seems to share the memory between tasks. This avoids the memory copies necessary in standard message passing implementations. Nevertheless, the trade-off is about the synchronization operations. In classical multi-tasking, task creation, synchronization and communication are handled by the programmer and the OS libraries. These operations are very costly, which led to implementation of lightweight processes: the threads [8]. But multi-tasking remains relatively complex. So, to alleviate this problem the use of compiler directives, which had been successfully tested on SIMD and vector machines, has emerged again.

The low-level programming needed by the classic threads and the lack of portability of compiler directives ([2]) led major parallel machine manufacturers, such as Compaq, HP, Intel, IBM, SGI and Sun, and other software developers to propose the OpenMP standard[1], which is better suited for large scientific applications. This standard, based on other essays like X3H5, has been designed to allow an easier programming, and to be architecture-independent and efficient on shared-memory architectures, which some programmers consider as the main features of an ideal parallel programming paradigm ([11]). OpenMP is a set of compiler directives, along with some library functions[2] ([2]). The compiler directives allow mainly parallel sections (and particularly the automatic loop decomposition), single sections (executed by only one thread), critical sections, and barriers. The scope attributes of variables are controlled by several clauses, like private, shared and reduction (see below). The OpenMP library provides functions to control and query the parallel execution environment, and lock functions.

It is worth-while to note that both models do not affect the serial optimizations, like loop-unrolling[3] and continuous memory access, which optimize the pipeline and cache memory usage. Some of these optimizations are done by the compilers quite efficiently, others need to be done by the programmer. Other problems arising in parallel programming, like cache conflicts and false sharing[4], can be avoided by the operating system or by the programmer himself, but are identical with multi-threading and OpenMP.

The next sections describe the strengths of both models.

2.2. OpenMP programming major features

OpenMP is aimed to be a portable, simple and efficient parallel programming model on shared-memory architectures ([2]).

It is of a higher level than the classic threads, and offers some useful functionalities frequently used in parallel programs. The creation of the threads is implicit and the domain of every thread in loop decomposition[5] is automatically calculated, unlike classic multi-threading. Here is a basic example of loop decomposition, accessing continuously the memory, in classic threads and OpenMP:

```
// Classic threads version
// size: the number of values assigned to every
//   thread (the last may have more)
size = last_index / nb_threads;
first = MY_TID * size;
last = (MY_TID != nb_threads-1) ?
    first + size : last_index;
for (i=first ; i<last ; i++)
  array[i] = ...;
-----------
// OpenMP version
// automatic loop decomposition
#pragma omp parallel for
for (i=0 ; i<last_index ; i++)
  array[i] = ...;
```

Due to the `pragma` directives (which are not taken into account by the compilers which do not recognize them) the sequential and the parallel version are the same, unlike in classical thread programming. This leads to an important issue: an (already) existing sequential program can be very easily parallelized by adding OpenMP directives to it. For instance, a loop with independent computations in a *C* program can be parallelized simply by adding the next directive before it:

```
#pragma omp parallel for
```

This is very useful for legacy-code or for already written programs which need a rapid and error-less parallelization. Several directives accept clauses that control the scope attributes of the variables among the threads. For example, adding a `private` clause to the directive above generates a local and private copy of the variable `i` for each thread:

```
#pragma omp parallel for private (i)
```

OpenMP also provides some high-level clauses frequently used in parallel programs, such as `reduction`, which applies the same operation on different variables and accumulates the result in a variable[6], and `single`, which starts a region executed by only one thread. They may be implemented in an optimized way by the compiler; for example a tree-based parallel reduction or a first-thread execute for single regions. In [2] other features are presented, like the ability to directly access memory throughout the system, fast shared-memory locks and its scalability and performance.

2.3. Classic multi-threading programming major features

Contrary to OpenMP, the classic threads need some effort to allow the features presented above. On the other

[1]Available since October, 1997

[2]See "OpenMP *C* and *C*++ [or *Fortran*] Application Program Interface", available at `www.openmp.org`

[3]Serial optimization which consists of transforming an n-loop into an n/k-loop by explicitly writing k times the body of the loop

[4]The sharing of the same cache line by two variables, which may lead to unwanted cache conflicts in a parallel execution

[5]Method of parallelizing a loop consisting of allocating an interval of the index of the loop to every thread

[6]Reduction example:
`for (i=0 ; i<100 ; i++) sum = sum + func(i);`

hand, being at a lower level, all OpenMP functionalities can be expressed by threads. Additionally, thread expression power is greater than in OpenMP. For instance, the Linux Pthreads library[7] provides three types of mutexes and provides semaphores, unlike OpenMP.

Classic thread programming allows a finer granularity of parallelism, so a finer control of parallel regions. For example, the barriers or the critical sections may concern only some of the threads. This is more difficult to achieve in OpenMP, where the barrier directives are bound to all the threads. This may be useful for a domain decomposition, where the threads need to synchronize only with their neighbours and not with all the threads. This can be written in *C* with a multi-threading library as follows:

```
// synchronization point for only 2 threads:
//   number 0 and 2
if ((mytid == 0) || (mytid == 2))
  barrier(2);
```

This lack of flexibility leads us to the idea that OpenMP is more suitable for data parallelism than for task parallelism.

3. Parallelization of a regular algorithm: Kohonen map

3.1. Kohonen neural network introduction

The ANNs represent a powerful tool in AI. Their interest domain ranges from pattern recognition to game theory and human brain simulation. There are many kinds of ANNs, but the implementation presented in this paper uses a two-dimensional SOM (Self Organizing Map) Kohonen network (see [7]).

A Kohonen network is formed by a map of neurons connected to the input area (figure 1). If the map is a two-dimensional array and the input array is multi-dimensional, the network may be thought as a way to visualize multi-dimensional images.

Like any other ANN, the use of the Kohonen network follows two steps: the *learning* step and the *testing* step. While learning, the input data are sequentially and repeatedly used until the ANN converges. During a cycle, one input data is used as input of the ANN, and the weights of some neuron connections are updated according to a formula like (1). The involved neurons are the closest neuron to the input data and its topological neighbours. The distance between (the weights of) a neuron and the input vector is defined as follows:

$$d_{out} = \sum_{in} (w_{in,out} - x_{in})^2$$

[7]See http://www.ap.univie.ac.at/users/havlik/Linux/documentation/FAQ/Linux_Threads

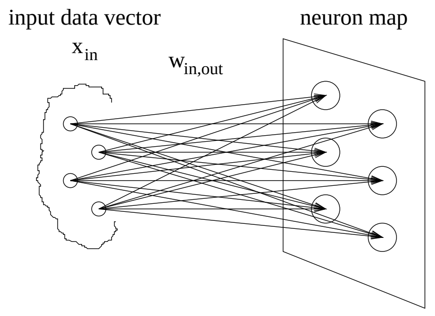

Figure 1. An example of a Kohonen network: a 2D neuron map connected to an input data vector

where x_{in} is the element of index in of the input data, and $w_{in,out}$ is the weight between the neuron *out* and the element x_{in}. The formula of the Kohonen map tends to bring the output map closer to the input data by changing the weights of the network connections:

$$w_{in,out} = w_{in,out} + \eta(x_{in} - w_{in,out}) \quad (1)$$

where η, the *learning-rate factor*, is a number between 0 and 1 which gives the speed of convergence.

While testing, the weights do not change, and the output of the ANN is used as the response of the ANN to the given input data.

3.2. Kohonen ANN parallelization issues

As seen in section 3, the Kohonen ANN uses a *static* network, where every output neuron is connected to every input neuron ([7]).

A straightforward parallel Kohonen algorithm consists of the following loop:

1. update the parameters: the learning-rate factor and the radius of the neighbourhood;
2. run all neuron computations: each neuron computes its distance to the current input data;
3. synchronization barrier for all neurons;
4. find the closest neuron to the current input;
5. run all weight update computations for the closest neuron and its neighbours;
6. synchronization barrier for all neurons;
7. increment cycle counter.

The steps 2 and 5 are highly parallel, but with fine-grain parallelism which is not adapted to MIMD architecture. To parallelize this algorithm, an efficient partitioning may be conceived, where each thread processes several neurons and each neuron is processed by only one thread. Many such types of partitioning insure an efficient load-balancing for step 2. But for step 5 the involved output neurons are *in a neighbourhood*, and the domain decomposition has to be chosen so that *each* neighbourhood contains neurons processed by a *maximum* of threads. The binding of the neurons to the threads is done consecutively: the first neuron is processed by the first thread, the second by the second thread and, generally, the nth neuron is processed by the thread number $n\ modulo\ p$, where p is the total number of threads. Figure 2 shows such a partitioning for 6 threads, on a 2D Kohonen output map. A more exhaustive study which introduces many domain decompositions and parallelization issues of Kohonen map is presented in [9].

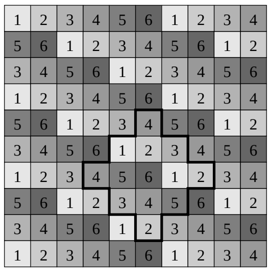

Figure 2. Example of domain decomposition optimizing the load-balancing for 6 threads

The calculations are independent, this avoids the majority of cache conflicts. Also, each neuron data (its weights, its input and its output) can be aligned on cache lines, in order to avoid the false sharing. The data occupied by the network uses a small memory space, so the processor cache is sufficient. For example, a Kohonen network with 100 input neurons and 256 output neurons has $100 \times 256 = 25000$ weights, or about $100kbytes$ of data. [1] introduces a specialized parallel library for artificial neural network implementation based on multi-threading, which deals also with serial optimizations and false sharing avoiding on shared-memory multi-processors.

Despite the features presented above, there are other trade-offs in the parallelization of the Kohonen map algorithm: there are two synchronization points per cycle and a sequential bottleneck at step 4. These may limit the speed-up obtained by its implementation.

3.3. Kohonen ANN performance

The comparative performance between *Fortran*-OpenMP version and *C*-threads version is shown in figure 3.

For the OpenMP version we used the *Fortran* language because the SGI *C* compiler was not OpenMP compliant at the time when the program was written (1999, February) and we had not the time to write a *C*-OpenMP version.

The application used for the tests is fine-grained. It is a 10×10 neuron map connected to a 16×16 input data array. As seen in the execution time curves, there is a small gap between the sequential times of the two versions, due to the language difference: *Fortran vs. C* (see [6] for such a comparison). Their performance is comparable until 5 processors, with a speed-up of about 3, which is the maximum speed-up of the *C*-threads version. Beyond this number, the *Fortran*-OpenMP version yields better results, with a maximum speed-up for 7–8 processors ($SUp_{max} = 3.4$). So *Fortran*-OpenMP version is a bit more scalable than *C*-thread version. The efficiency curve is relatively good until 4 processors, with a value of 70%.

As the results show, there is no major difficulty (beyond the trade-offs inherent to the algorithm) in implementing a parallel version of the Kohonen map. This is due to the regularity of its calculations, as shown in section 3.2.

4. Parallelization of irregular algorithms: a multi-agent system

4.1. Situated multi-agent system introduction

The goal of a situated multi-agent system (SMAS) is to simulate a society of agents in a real world, i.e. where natural constraints exist. This allows to model behaviours of agents for a maximum efficiency or to emerge yet unknown behaviours. Though the definition of an agent is not yet world-wide agreed, a situated agent can be thought as an entity being in an environment (figure 4), capable of autonomous actions to fulfill its goals, using its perceptions of the environment and communicating with other agents ([5]).

For example, an SMAS may be used to model a set of robots, as the implementation used in this paper. The real world is modeled by an environment of $M \times N$ squares containing obstacles (walls), where robots have to carry all the ore from mines to factories. Every wall, mine or factory occupies one square. During a cycle, a robot may stay, may move, may take ore from a mine or may drop the ore it has into a factory. If several robots try to enter the same square, a spatial conflict appears. The system will let then only one to enter the square, the others have to stay. The robots are guided by a potential field simulating the odour, which is a decreasing function of the distance between the robot and

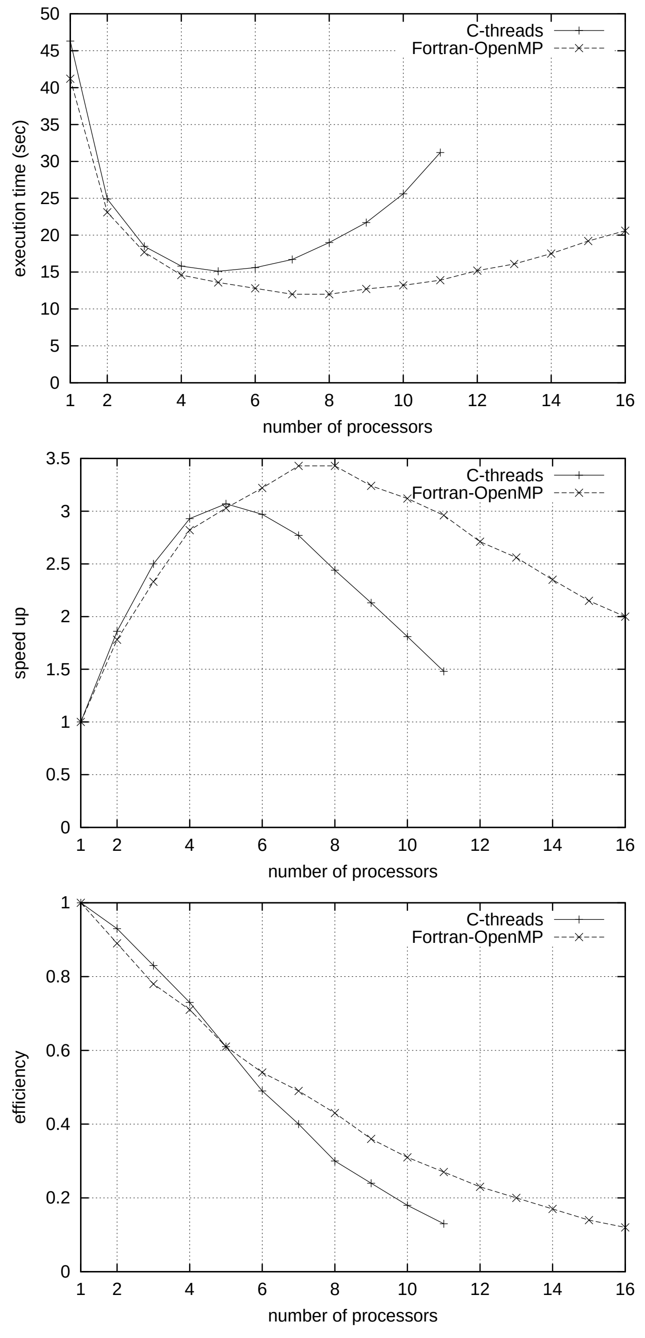

Figure 3. Comparative results of C-thread and Fortran-OpenMP implementations of Kohonen map

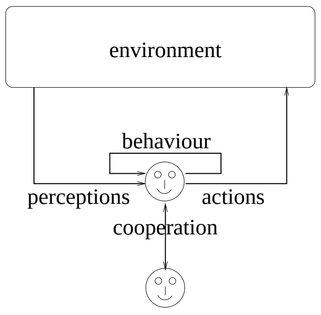

Figure 4. Illustration of the agent concept

the mine/factory. To avoid the walls, the implementation of this potential field uses a wave propagation algorithm (see below).

4.2. SMAS parallelization issues

Unlike ANNs, the SMASs are generally more complex and dynamic. The application is dynamic due to the agents which move, and also to the environment, which changes its state. The agents and the environment are mainly the parts which may be parallelized. They may have a different partitioning, but, in order to minimize the cache conflicts, the same domain-partitioning was chosen. However, the load-balancing becomes a goal difficult to achieve. Sometimes an agent may stay (doing nothing), sometimes it may be looking for a source. When choosing a domain-partitioning, the agents may migrate from a domain to another, and the potential of a source, which changes in time, may spread on several domains. A dynamic load-balancing may be imagined, but this is not easy to implement and leads to cache conflicts. Several synchronization points are needed in order to solve the spatial conflicts and to correctly propagate the potential of the sources. The dynamic nature of the SMAS results in other actual parallelism issues, like false sharing and cache conflicts.

Due to their complexity, the data needed by these applications fits no longer in the processor caches. A common example is a 512×512 environment with 4096 agents, which may occupy several *Mbytes* of memory. Thus, a careful algorithm has to be used in order to make the best use of the caches.

The irregularity of this system is exemplified by the use of the wave propagation algorithm, which may be the main time-consuming part of this SMAS. Its parallelization is described in the next section.

Wave propagation algorithm

The reason for using this algorithm is that it tries to better simulate a real hypothesis: the intensity of the potential (odour for example) of a source in a square of the environment is proportional to the distance to that source, with avoidance of obstacles (figure 5). If a square belongs to the

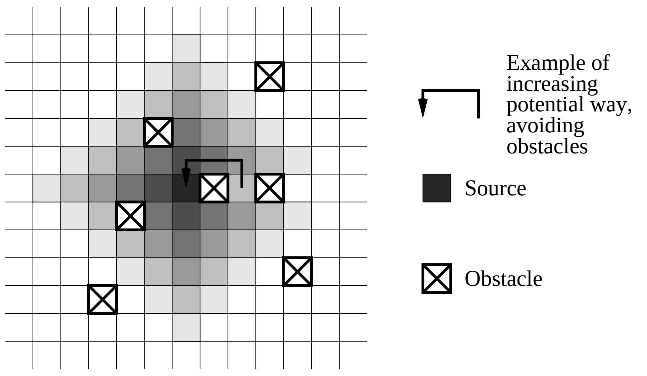

Figure 5. Wave propagation algorithm illustration

potential field of several sources, the maximum potential is used by the square. This allows an agent to avoid obstacles, following increasing potentials. But the parallelization of this algorithm is a difficult task, because the potential of a square may be influenced by several sources, and the sources may be processed by different threads. Three variants of its parallelization have been explored:

1. domain decomposition: the environment is decomposed in several domains. Every thread works on a distinct domain and updates the potential of its squares. The potential field of a source may spread on several domains. Several boundary exchanges then may need to take place, each of them requiring a synchronization point for several threads.

2. data decomposition with thread-private environments (figure 6): as a first step, every thread works on a

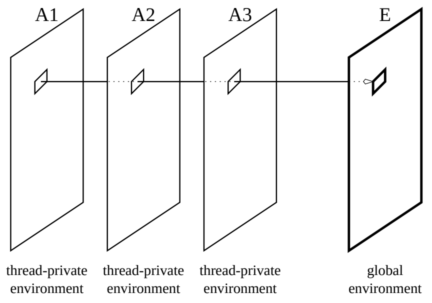
A1
A2
A3
E
thread-private environment
thread-private environment
thread-private environment
global environment

Figure 6. Example of data decomposition for the wave propagation algorithm (3 threads)

part of the sources, and updates its potential field on a whole thread-private copy of the environment. After that, as the second step, the environment is updated using the thread-private copies of the environment: every square of the environment has a potential equal to the maximum of the corresponding squares of the thread-private environments. This solution needs a lot of memory (for the private environments) and leads to many cache conflicts for the second step.

3. data decomposition with mutexes: every thread propagates the potential of a part of the sources. To solve the concurrent access to a square influenced by several sources, a mutex (variable used to ensure a *mut*ual *ex*clusion region) per square or per group of squares is used. This leads to expensive operations needed by mutex operations.

The dynamic nature of such kind of irregular applications may decrease the performance of their implementations on actual parallel machines and makes difficult their OpenMP implementation. We encountered important difficulties during OpenMP implementation of our SMAS and we stopped it after the same development time accorded to the *C*-thread version.

4.3. SMAS performance

The performance of the C-thread implementation is shown in figure 7.

The time decreasing is not negligible: in the best case the sequential time can be decreased by 60%, and the implementation scales well: the speed-up curve is closed to a line until 14 processors. But its maximum value, equal to 2.7, needs 16 processors and it is far from the ideal speed-up ($SUp(P) = P$), so efficiency is low, as we can see on the efficiency curve which decreases quickly.

5. Performance comparison

Situated multi-agent systems and Kohonen maps have different natural parallelism, and introduce different parallel algorithmic issues. A Kohonen neuron needs a few computations, and uses only local variables (its weights, its inputs and its output). It has a natural fine grain parallelism. Moreover, more complex neural networks exchange data between neurons, for example between associative Kohonen maps, and have a natural message passing paradigm. At the opposite, an agent can need a lot of computations, and uses a data structure modeling the environment shared by all the agents. This global data structure can be an important source of memory contention and false sharing. Finally, situated multi-agent systems present a natural intensive memory sharing paradigm.

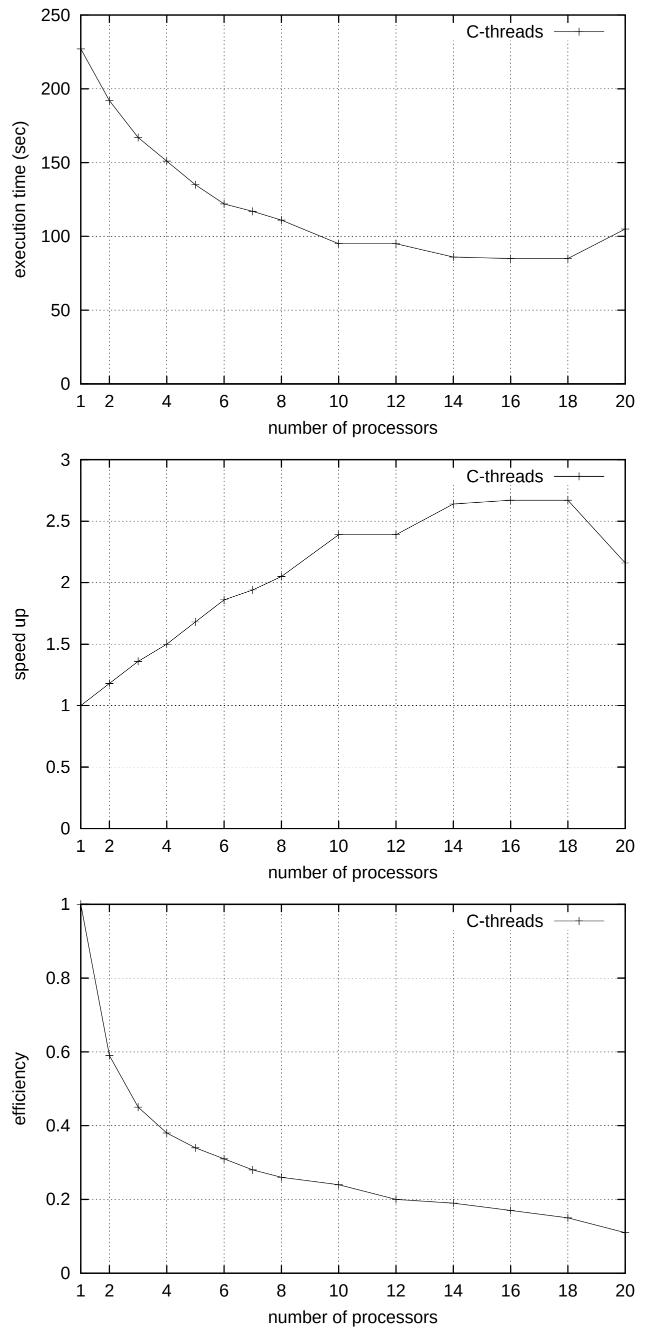

Figure 7. Results of C-thread implementation of a Situated MAS on an SGI-Origin2000

Implementation	Regular algorithm: Kohonen map C-threads	Regular algorithm: Kohonen map Fortran-OpenMP	Irregular algorithm: Situated MAS C-threads	Irregular algorithm: Situated MAS C-OpenMP
Maximum speed-up	3	3.5	2.7	–
Optimal thread number	5	7–8	16–18	–
Parallelization complexity	middle	low	high	–
Development time	2 weeks	1 week	5 weeks	> 5 weeks
Source code lines	450	400	950	850

Table 1. Implementation development information

These applications have different natural parallelism and they need special parallel implementation mechanisms to run efficiently by avoiding the problems presented above (memory contention, false sharing and fine-grained computations). This leads the programmer to deviate from their natural parallelism:

- Firstly, best results of Kohonen map parallelization were obtained processing *several* neurons by task, with loop decomposition in OpenMP (see section 3.3);

- Secondly, our situated multi-agent system has needed an explicit multi-threading parallelization, based on domain-decomposition of the environment, and not on a partitioning of the agents among the processors. As seen in section 4.2, its irregular computations could not be easily expressed with OpenMP.

Table 1 summarizes information about the implementations of an ANN as a regular problem, and an MAS as an irregular problem. No theoretical comparison was done, but from these experiments we can deduce that:

1. The execution times of both classic multi-threading and OpenMP implementations are comparable for regular applications.

2. The development time with OpenMP is much less than with classic threads in regular applications. So in our experience OpenMP is more user-friendly than explicit multi-threading, as we have learned it in very short time and have successfully used it for Kohonen parallelization.

3. For irregular applications the higher-level of OpenMP leads to difficulties in programming. Sometimes, the use of foreign parallel functions is necessary, so we think that OpenMP still remains limited to regular computations.

For concreteness, we present here the experimental conditions. The applications were run on a supercomputer SGI-Origin2000 [3], with 64 processors MIPS R10000 at 195MHz, each with 4MB L2 cache. The operating system was Irix64, version 6.5.6f. The programs were written in C and Fortran and compiled with the native compiler, `MIPSPro`, version 7.3.1m. The compiler optimization flag used was `-Ofast=ip27` (maximum standard optimization) and the tests were made in exclusive (mono-user) mode.

6. Conclusions

The new-emerging DSM architecture leads to a regain of interest for shared-memory programming paradigm. We have compared two such parallel programming paradigms: OpenMP and classical multi-threading. The comparison was done on two kinds of applications, a regular one and an irregular one, both in terms of development time and execution time.

Based on the tests shown in this paper (section 5) and on our experience, both with us and with some students at Supelec, we have arrived to the following conclusions:

- For regular applications, multi-threading and OpenMP yield the same execution times, but OpenMP programming needs a development time about twice smaller than classical thread programming. We advise to use OpenMP for parallelization of regular applications, as it provides fast parallel executions and is easy to learn and to use.

- Irregular applications are much more difficult to parallelize. The multi-threading paradigm seems to be nevertheless much simpler to program than the OpenMP one because of its expression power.

Acknowledgments

Support for this research is provided in part by a grant from *Région Lorraine* (France), and access and support to Origin2000 supercomputer are provided by *Charles Hermite Center* (France).

References

[1] Y. Boniface, F. Alexandre, and S. Vialle. A library to implement neural networks on MIMD machines. In *EuroPar-1*, 1999.

[2] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Jan.-Mar. 1998.

[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] A. Grujić, M. Tomašević, and V. Milutinović. A simulation study of hardware-oriented DSM approaches. *IEEE Parallel & Distributed Technology*, 4(1):74–83, Spring 1996.

[5] N. Jennings and M. Wooldridge. Applications of intelligent agents. In N. Jennings and M. Wooldridge, editors, *Agent Technology. Foundations, Applications, and Markets*. Springer, 1998.

[6] L. Kale. Programming languages for CSE: The state of the art. *IEEE Computational Science & Engineering*, 5(2):18–26, Apr.-June 1998.

[7] T. Kohonen. *Self-Organizing Maps*. Springer, second edition, 1997.

[8] B. Nichols, D. Buttlar, and J. P. Farrel. *Pthreads Programming*. O'Reilly & Associates, Sept. 1996.

[9] I. Pitas, editor. *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*. John Wiley & Sons, 1993.

[10] J. Protić, M. Tomašević, and V. Milutinović. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology*, 4(2):63–79, Summer 1996.

[11] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.