Long parallel algorithm design *vs.* quick parallel implementation

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

1 Motivations

Some applications have a parallel natural algorithm well adapted to modern MIMD parallel computers, while others seem at first look to be sequential and need a parallel algorithm design. In this last case, we have to choose between doing just a parallel implementation of a poor parallel algorithm, or doing a parallel algorithmic effort before the parallel implementation.

As today it is possible to quickly parallelize sequential source code using OpenMP, the temptation is great to avoid to design new parallel algorithms. This paper relates some new and previous experiences [2], and points out the difference between parallel algorithmic and parallel implementation, and some contributions of OpenMP.

2 Straightforward parallel implementation of image processing

We are currently working on robotics projects, and we need to detect special marks in the environment to check the position of the robots. We use marks based on P-similar function curves [8], that can be detected with few computations, and are very scarce in natural environment. Even if the sequential execution time is short (less than 1s) we need to reduce it to the maximum, as it is just a part of a complex robot management program that has to run in real time. So, we need to run this program on a parallel computer.

But we cannot use a big parallel machine to manage all the robots, shared with many other users: execution time would depend on the load of the machine, and would not be always in real time. We need a dedicated parallel computer for each robot. Moreover, we cannot buy a big parallel computer for each robot, this would be too expensive. We have to use small parallel computers, and to implement high efficient parallel programs in order to obtain very small execution times on few processors. Finally, today we use a cheap 4 processor SGI-VWS-540 parallel computer, with Pentium III-Xeon processors and Windows-NT OS.

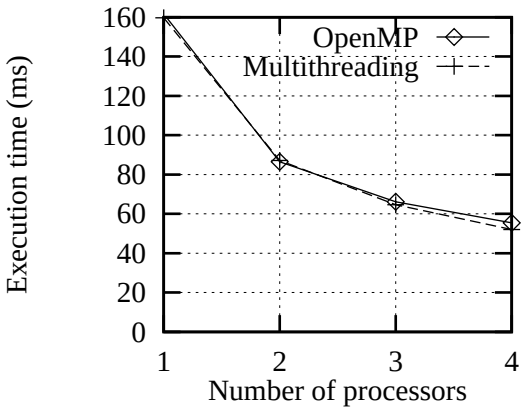

Figure 1: Execution times of two straightforward parallel implementations of P-similarity detection (image processing): OpenMP implementation and classic multithreaded one, on SGI-Origin2000 parallel computer

P-similarity detection is well adapted to these MIMD parallel machines: image lines can be processed concurrently without communications, and we can group several lines per task in order to obtain coarse grained parallelism. Natural parallelism of the algorithm can be easily and efficiently exploited, no algorithmic effort is needed. We have made two parallel implementations, using OpenMP (Kai compiler) and classical multithreading technics based on native Windows threads.

We obtained very good and close results for both implementations as shown in figure 1. Efficiency was close to 75% on four processors (limited by a residual serial fraction), and OpenMP implementa-

Implementation	Total line number	Overhead
Sequential	3443 lines	–
OpenMP	3447 lines	4 lines
Multithreaded	3631 lines	188 lines

Table 1: Number of lines of P-similarity detection implementations, with OpenMP and classical multithreading

tion was easier (development time and source code were reduced, see table 1). This is an example of a regular computation application, with natural parallelism well adapted to MIMD parallel computers. No parallel algorithmic effort is needed, just a parallel implementation effort, and OpenMP can be used successfully to obtain high efficiency and minimize development time.

3 Simple parallelisation of Kohonen neural network

Kohonen neural network ([3]) is another example of regular algorithm containing natural parallelism. It is a fine-grained algorithm, but it can be easily parallelized on modern parallel machines by grouping several computations in one task. Some care is needed to avoid cache coherence contention and, if the data size is sufficiently large, to find a good load balancing among the processors.

A cycle of simulation is divided in three parts. The first part is the computation part, where each neuron computes its distance to the input data independently of the other neurons. Several neuron computations can be grouped in one task. The second part is the detection of the winning neuron, where the neuron which has the minimum distance (the winner) is found. This is a sequential bottleneck inherent to the Kohonen algorithm. Note that the `REDUCTION` clause of OpenMP cannot be used in this case, because we do not need to know the minimum distance, but the neuron whose this minimum is associated. Finally, the third is the update part, where the neurons in the neighbourhood of the winner update their weights. Again, the neurons can be grouped in one task to increase the grain of parallelism. But because only some neurons are concerned, a partitioning which balance the load among the processors might be necessary (figure 2). Others types of partitioning are shown in [7].

We have implemented this algorithm in Fortran-

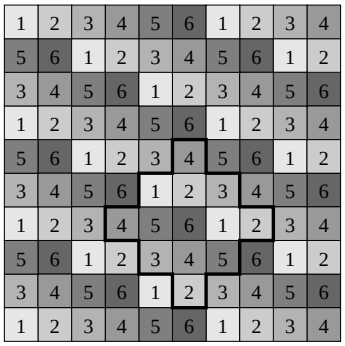

Figure 2: Data partitioning principle of a 10×10 Kohonen map parallelization trying to maximize the load balancing: the neurons are cyclically distributed among the threads (6 on this example)

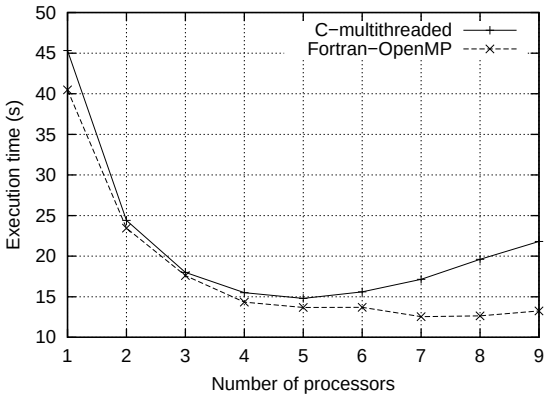

Figure 3: Execution times of the Kohonen map implementations in Fortran-OpenMP and C-multithreaded on SGI-Origin2000 parallel machine

OpenMP and C-multithreaded, both on an SGI-Origin2000 [2]. The input data is a 16×16 matrix, and the neuron map is 10×10. We obtained relatively similar execution performances: speed up close to 3 on 5 processors, and growing up to 3.2 on 7 processors for the Fortran-OpenMP version (figure 3). But the OpenMP version was written much faster than the C-multithreaded one, and, as shown in figure 3, is more scalable than the multithreaded one.

The Kohonen algorithm does not need an algorithmic work, but a minimum effort in parallel implementation. The results show that this algorithm is better suited to parallelisation with OpenMP, both in development time and in execution performance.

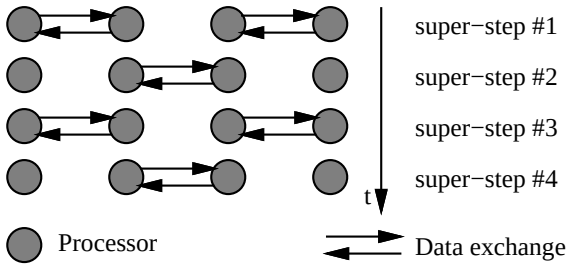

Figure 4: Odd-even bubble sort applied to a list of four elements

4 New algorithmic design of parallel bubble sort

4.1 Natural parallel implementation

Bubble sort is an example of totally sequential algorithm. The data input of any comparison depends on the results of the previous comparisons. But the *odd-even* bubble sort [4] interchanges the order of comparisons and may be parallelized.

This algorithm sorts a data list of length N, while running N super-steps, and each super-step running $N/2$ elementary steps. Figure 4 illustrates this principle on a list of an even number of elements (4 in this example). The algorithm begins with an odd super-step (number 1), and each element of the initial list is associated to one of its neighbors, in order to form $N/2$ couples. Each couple of elements are compared and exchanged if necessary. This constitutes the $N/2$ compare-exchange operations, or $N/2$ elementary steps, of the first super-step. At the next super-step (number 2), each element of the list is associated with its other neighbor, in order to form $N/2-1$ couples (one couple in the example of figure 4). As previously, each couple of data is sorted, and super-step number 2 is composed of $N/2-1$ elementary steps. Following super-steps are alternatively identical to the first or the second super-step, and the data list is sorted at the end of the super-step number N.

We have made an OpenMP parallel implementation of this algorithm, adding only 3 lines to the sequential code: one `#include <omp.h>` line and two `#pragma omp parallel for` lines. Figure 5 shows the main routine of the source code.

First, we have run this program on a small set of data on a small parallel computer (a 4-processors SGI-VWS-540), and we have obtained an efficiency close to 92%. Second, we have run the same program on a larger data set (2 millions of floating point numbers) on a larger parallel computer (an SGI-Origin2000, with 64 processors). Figure 6 illustrates the significant execution time decrease of this benchmark.

```
for (step = NbDataT; step > 0; step--) {
  if (step % 2 == 0) {      /* Even steps */
    #pragma omp parallel for private(i,buff)
    for (i = 0; i < NbDataT-1; i += 2)
      if (TabData[i] > TabData[i+1]) {
        buff = TabData[i];
        TabData[i] = TabData[i+1];
        TabData[i+1] = buff;}
  } else {                  /* Odd steps  */
    #pragma omp parallel for private(i,buff)
    for (i = 1; i < NbDataT-1; i += 2)
      if (TabData[i] > TabData[i+1]) {
        buff = TabData[i];
        TabData[i] = TabData[i+1];
        TabData[i+1] = buff;}
  }
}
```

Figure 5: Main routine of the straightforward OpenMP implementation of the odd-even bubble sort: only two OpenMP directives have been added

Performances seem great, but execution times remain high. We had to give up sorting the larger set of data on less than 8 processors: execution times were too large. This is the medium result we obtain with a quick parallel implementation of the bubble sort algorithm (using OpenMP), and a small parallel algorithmic effort (odd-even bubble sort in place of classic bubble sort). In the next paragraph we introduce another parallel bubble sort algorithm, which needs greater algorithmic effort than parallel implementation effort, but supplying better performances.

4.2 New parallel algorithm design

A better algorithm based on the parallel bubble sort principle consists of exchange and compare only P macro-elements on P processors, in P super-steps. Each macro-element is an ordered list of N/P elements, loaded and sorted sequentially on a processor [4]. For this sequential and initial step an optimized sequential sort has to be used, such as quick-sort.

At each step, each processor exchanges its local ordered list with one of its neighbor, and extracts the N/P less or greater data of the two ordered lists of length N/P, as illustrated on figure 7. This

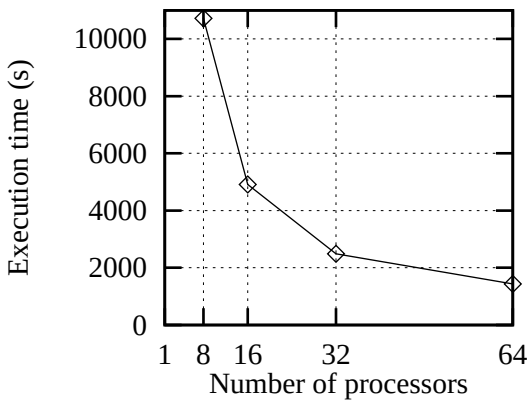

Figure 6: Decreasing execution time of the odd-even bubble sort implemented with OpenMP on an SGI-Origin2000, and applied to sort 2 millions of floating point numbers

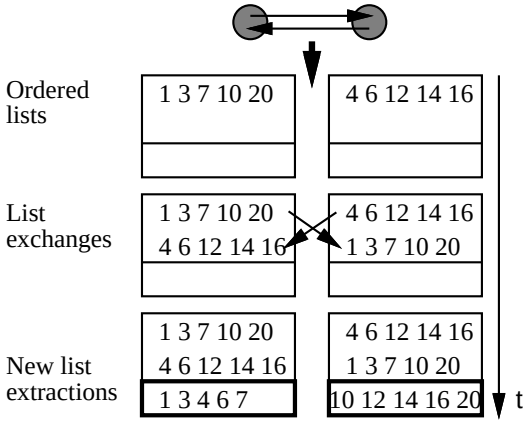

Figure 7: Details of one comparison-exchange operation of the optimized parallel bubble sort

Implementation	Total line number	Overhead
OpenMP	1399 lines	–
Multithreaded	1802 lines	+403 lines

Table 2: Number of lines of the optimized parallel bubble sort implementations

is an easy and fast operation, since a simple comparison of the two ordered list heads supplies the less element. Then it is extracted from its list, and the two new list heads are compared again, and supply the second less element. The process stops when N/P elements have been extracted. Identical mechanism on queue lists allows to extract easily the N/P greater elements on the neighbor processor.

We have implemented this algorithm on a shared memory machine, an SGI-Origin2000, using MPI and multithreading paradigm, and more recently using OpenMP. MPI implementation was not very efficient as message passing was longer than memory sharing, but multithreaded and OpenMP implementation performances were high and close. Figure 8 shows the short and close execution times of both versions, sorting a list of 8 million floating point numbers.

As previously shown, OpenMP implementation is shorter (see table 2) and performances are identical, so OpenMP seems to be the best way to implement this new parallel algorithm. We have not used the `#pragma omp parallel for` directive, as we did for basic bubble sort parallelization issued from sequential implementation. We have used one `#pragma omp parallel` directive to run threads in parallel, and some `#pragma omp barrier` to synchronize the list exchanges and list manipulations instead. This is a parallel programming style close to the classic multithreading one; differences are mainly syntactic. But simplicity of OpenMP syntax is highly appreciable to minimize development times.

Comparing figures 6 and 8, we notice that the new parallel algorithm is significantly faster than the basic odd-even bubble sort version: execution times are 1000 times less on a 4 times greater problem! This is the positive result of the great parallel algorithmic effort that we made to change the basic odd-even bubble sort algorithm, to introduce local quick-sort on local data, and to exchange and compare some ordered lists in place of elementary elements. The performance increased a lot, but the parallel algorithmic effort became higher than the parallel implementation one and the development time increased significantly.

This example shows that modern and powerful parallel implementation tools, such as OpenMP, allow to quickly parallelize some sequential source codes, but do not dispense with designing new and faster parallel algorithms. Opposite the P-similarity detection described in section 2, some applications have not efficient and natural parallel algorithms.

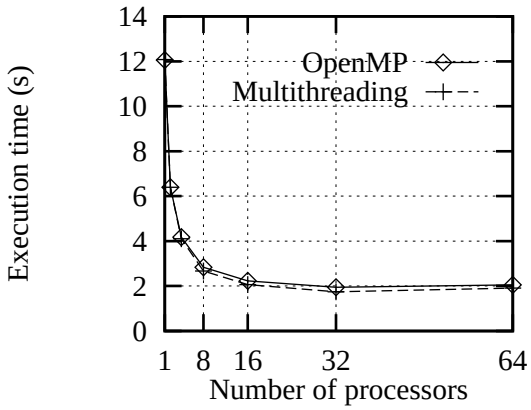

Figure 8: Identical execution times on an SGI-Origin2000, of the new algorithm of parallel bubble sort, applied to a list of 8 millions of floating point numbers, and implemented both with classic multithreaded and OpenMP

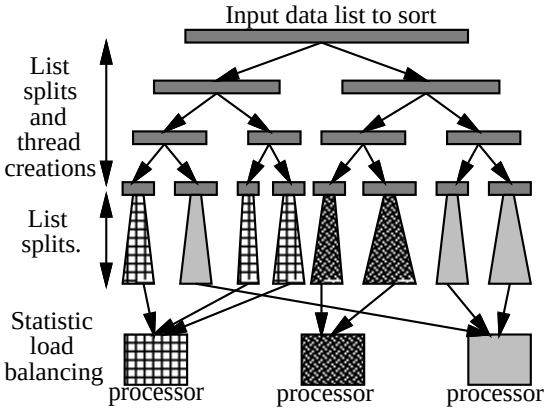

Figure 9: Natural quick-sort parallelization: recursive calls are associated to thread creations until a fixed deep, and tasks are statistically load balanced among the processors

5 Attempt of natural parallelization of quick sort

Some very efficient parallel sort algorithms exist, more efficient than the optimized parallel bubble sort of the previous section. One of the best known is the *hyperquicksort* [4]. It is based on the classical quick-sort principles (pivot values usage, and list split) and is run on a hypercube network of processors, doing local quick sort on each processor, and minimizing processor communications. We have implemented it both with message passing and memory sharing paradigm, and we have obtained very good performance until 64 processors on an SGI-Origin2000. No doubt we could implement it using OpenMP, as we have done for the optimized parallel bubble sort. But it remains a complex and not obvious parallel sort, and we wanted to compare its performances to the ones of a straightforward parallelization of the sequential and well known quick-sort algorithm, as we did in section 4 with different versions of parallel bubble sort.

Quick sort is an efficient sorting algorithm containing natural parallelism: it is based on recursive data list decomposition that can be processed concurrently. Then we can create a new task each time a recursive call is done. Load balancing among the P processors will be statistically assured, as each processor will run a lot of tasks. However, in order not to create too many tasks nor too small tasks, we limit the task creation to a fixed depth of the recursive call, as shown in figure 9. We have already successfully used this way of parallelization, based on large number of task creations instead of recursive calls and statistical load balancing, on an N-queen problem [5]. So, we wanted to use this strategy to quickly parallelize a sequential quick-sort source code, using OpenMP and its nested parallelism capabilities, allowing any thread to create new threads.

But the two OpenMP compilers we used (Kai and SGI compilers) do not support yet nested parallel sections, so we limit the splitting deep of the recursive call tree to 1, creating only 2 threads. We obtained an efficiency ranging from 92% to 67% when sorting random lists on 2 processors on an SGI-Origin200 or an SGI-VWS-540, depending on the input data list. There were not enough threads to insure a statistical load balancing. But the OpenMP implementation of this strategy was easy, and it would be very interesting to compare its performances to more complex implementations as hyperquicksort ones, when nested parallelism will be available in OpenMP compilers. We claim that nested parallelism could be very useful to quickly parallelize some algorithms based on recursive calls.

6 Complex parallel algorithm of Multi Agent System.

We have developed a situated multi-agent system that simulates a set of robots evolving on a building floor. We use stochastic mechanisms to model

Application	Parallel algorithmic effort ?	Is OpenMP as efficient as multithreading ?
P-similarity detection	None	Yes
Kohonen neural net	Small	Yes
Optimized bubble sort	Great	Yes
Recursively parallelized quick sort	None	Need nested parallelism

Table 3: Main results of our experimentations: execution efficiency and parallel algorithmic effort.

uncertainties of observations and actions, and we model simultaneity of actions such as moving conflicts [1]. This application has a natural parallelism: the agents act in parallel. But one agent does not make enough computations to be an entire process or thread, and simultaneity model leads to grouping in a same task all the agents engaged in a same conflict (one conflict has to be solved sequentially). So, number and size of tasks change when agents move, and we have to use a dynamic load balancing mechanism to efficiently parallelize our multi agent system, and to process several tasks in a same thread.

We have implemented this situated multi-agent system on a distributed shared memory computer (an SGI-Origin2000), using a multithreading paradigm and a double *work-pool* mechanism to do dynamic load balancing [6]. This is an example of a very complex parallel algorithm, really different from natural and agent-based parallelism: the implementation uses threads (that do not match agents), shared counters and semaphore-based synchronization. It seems possible to use OpenMP to implement this application, for example by putting conflicts in a shared table, and by using parallel loop (`#pragma omp parallel for`) with the `dynamic` scheduling option, but we have not yet experimented this solution. However it will not allow a straightforward parallel implementation of this application. This will remain a not obvious parallelization, and parallel algorithmic effort will be greater than parallel implementation effort.

7 Conclusion

OpenMP allows to decrease development time of many parallelizations, especially for regular computation algorithms[2], without loss of performances, compared to classic multithreading paradigm (see table 3). But if the natural algorithm of the application does not contain parallelism adapted to MIMD modern computers, then we have to design a new and efficient parallel algorithm rather than look for an efficient implementation of the natural one.

Parallel algorithm design and parallel implementation are two different steps of parallel programming, and OpenMP simplifies parallel implementation but does not dispense with parallel algorithmic effort when necessary.

Acknowledgments

Authors thank Vincent Rieger and Cédric Rose for their implementations and benchmarks of P-similarity detection, and the Charles Hermite Center (France) for accesses to its SGI-Origin2000.

References

[1] M. Bouzid, V. Chevrier, S. Vialle, and F. Charpillet. A stochastic model of interaction for situated agents and its parallel implementation. In *ACIDCA*, Monastir, Tunisia, 2000.

[2] E. Dedu, S. Vialle, and C. Timsit. Comparison of OpenMP and classical multithreading parallelization for regular and irregular algorithms. In *SNPD*, Reims, France, May 2000.

[3] T. Kohonen. *Self-Organizing Maps.* Springer, 1997.

[4] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing.* The Benjamin/Cummings Publishing Company, 1994.

[5] Y. Lallement, T. Cornu, and S. Vialle. Application developpement under ParCEL-1. In *PPAI-95*, Montreal, Canada, 1995.

[6] B.P. Lester. *The art of parallel programming.* Prentice Hall, 1993.

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

[8] D. Scharstein and A. Briggs. Fast recognition of self-similar landmarks. In *IEEE Workshop on Perception for Mobile Agents*, June 1999.