PARALLEL IMPLEMENTATION OF AN ANT COLONY OPTIMIZATION METAHEURISTIC WITH OPENMP

Pierre Delisle[(1)], Michaël Krajecki[(2, 3)], Marc Gravel[(1)], Caroline Gagné [(1)]

(1) Département d'informatique et mathématique, Université du Québec à Chicoutimi, Chicoutimi, Québec, Canada, G7H 2B1
pdelisle@uqac.uquebec.ca, mgravel@uqac.uquebec.ca, caroline_gagne@uqac.uquebec.ca

(2) Université de Reims Champagne-Ardenne, LERI
BP1039, F-51687 Reims Cedex2
(michael.krajecki@univ-reims.fr)

(3) Collège Militaire Royal du Canada,
CP 17000, Succursale Forces, Kingston, Ontario, Canada, K7K 7B4

ABSTRACT

This paper presents a parallel implementation of an ant colony optimization metaheuristic for the solution of an industrial scheduling problem in an aluminum casting center. The usefulness and efficiency of the algorithm, in its sequential form, to solve that particular optimization problem has already been shown in previous work. However, even if this method, as well as metaheuristics in general, offers good quality of solution, it still needs considerable computational time and resources. Moreover, the structure of the algorithm makes it well suited for parallelization, so a considerable improvement can be achieved that way. In this paper an efficient and straightforward OpenMP implementation on a shared memory architecture is presented and the main algorithmic and programming issues that had to be adressed are discussed. The code is written in C and the application has been executed on a Silicon Graphics Origin2000 parallel machine.

1 INTRODUCTION

In many industrial scheduling situations, exact optimization algorithms require overlong solutions times and cannot produce an acceptable or even feasible solution in the time available. Metaheuristics have been shown to offer successful solution strategies for that kind of problems. More specifically, Gravel *et al.* [2001] have presented an efficient representation of a continuous horizontal casting operation, a real scheduling problem encountered in an aluminium foundry, using an ant colony optimization metaheuristic. The algorithm (in its original sequential form) has been implemented in an application that is now used in the foundry. However, even if it has proven to be a viable solution, it is still very demanding in computational time and resources and this is particularly true as the problem size increases. Moreover, the generation and evaluation of ants (solutions), which are the main parts of the algorithm and its principal source of computational cost, offers a low dependancy degree, so the structure of the algorithm makes it well suited for concurrent execution. For those reasons, a parallel approach was justified and an implementation was made to study the performance in execution time that could be obtained that way.

We already have shown the interest of using OpenMP for the parallelization of irregular Applications (Habbas *et al.* [2000]). In this previous work we have studied an exact algorithm for CSP resolution. We hope to show in this paper that this approach can be applied to the present Ant Colony Optimization implementation.

In the first section of this paper the Ant Colony Optimization algorithm (ACO) is presented. Then, the choice of using a shared memory model and the OpenMP environment is explained, the parallel implementation of the ACO is detailed and some algorithmic and programming issues that had to be adressed during the parallelization process are discussed. Finally, some results are presented to show the performance of the resulting parallel application.

2 ANT COLONY OPTIMIZATION ALGORITHM (ACO)

The first ant colony optimization metaheuristic (ACO), called ant system (Colorni *et al.* [1991], Dorigo [1992]), was inspired by studies of the behavior of ants (Deneubourg *et al.*, [1983]; Deneubourg & Goss, [1989]; Goss *et al.*, [1990]). Ants communicate among themselves through

*pheromon*e, a substance they deposit on the ground in variable amounts as they move about. It has been observed that the more ants use a particular path, the more pheromone is deposited on that path and the more it becomes attractive to other ants seeking food. If an obstacle is suddenly placed on an established path leading to a food source, ants will initially go right or left in a seemingly random manner, but those choosing the side that is in fact shorter will reach the food more quickly and will make the return journey more often. The pheromone on the shorter path will therefore be more strongly reinforced and will eventually become the preferred route for the stream of ants. The works of Colorni *et al.* [1991], Dorigo *et al.* [1991], Dorigo *et al.* [1996], Dorigo & Gambardella, [1997], Dorigo & Di Caro, [1999] offer detailed information on the workings of the algorithm and the choice of the values of the various parameters.

In the multiple-objective scheduling problem treated in this paper, we must determine the processing sequence for a set of orders where setup times are sequence-dependent. Our formulation is based on the well-known traveling salesman problem (TSP). Each order to be processed is represented as a "city" in the TSP network. When an ant moves from city *i* to city *j*, it will leave a *trail* analogous to the pheromone on the edge (*ij*). The trail records information related to the previous use of edge (*ij*) and the higher this use has been, the greater is the probability of choosing it once again. For the scheduling problem, the pheromone trail will contain information based on the number of times ants chose to make jobs (*ij*) adjacent. We will explain later how the trail is initialized and modified.

At time *t*, from an existing partial job sequence, each ant *k* chooses the next job to append using a probabilistic rule $p_{ij}^{k}(t)$ based on *visibility* (η_{ij}) and on the *intensity of the pheromone trail* ($\tau_{ij}(t)$). For the scheduling problem, the visibility is defined by a matrix that aggegates information on each of the four objectives to minimize. This matrix represent the visibility information analogous to the D matrix in the TSP. At initialization of the algorithm, the trail intensity for all job pairs (*ij*) is initialized to a small positive quantity τ_0. Parameters α and β are used to vary the relative importance of the trail intensity and the visibility. To ensure that a job that already been placed in the sequence being constructed is not again selected, a tabu list is maintained. Each ant *k* will have its own tabu list $tabu_k$ recording the ordered list of jobs already selected.

At any given time, more than one ant constructs a job sequence and a *cycle* is completed when each of the *m* ants has completed their construction. The version of the algorithm proposed in this paper carries out an updating of the trail intensity at the end of each cycle. This allows us to update the trail according to the evaluation of the solutions found in the cycle. Let the evaluation on the most important objective (h') found by the kth ant be $L_k^{h'}$. The contribution to the update of the pheromone trail for ant *k* is the calculated as follows: $\Delta\tau_{ij}^{k}(t) = Q/\ L_k^{h'}(t)$, where Q is a system parameter. The updating of the trail is also influenced by an evaporation factor (1-ρ) that diminishes the trail present during the previous cycle.

The reader may consult Gravel *et al.* [2001] for the details of these adaptations to the original ACO to make it fit to the actual industrial problem. To treat the multiple-objective optimization problem, all nondominated solutions found by the metaheuristic are stored in a quadtree (Finkel & Bentley, [1974]). If a solution is dominated, it will be eliminated during the quadtree insertion process, and if it dominates other solutions already in the quadtree, then the insertion process will remove them before the solution is inserted in its correct position.

Then, for clarity issues regarding the next section, the ACO algorithm can be formulated (see figure 1) in the following way, which is less formal than the original specification, but simpler and closer to the way it has been implemented.

Figure 1 **Sequential implementation of the ACO**

```
NC = 0;
Initialize τij matrix;
Initialize quadtree;
While (NC < NCMax) and (Not Stagnation
Behaviour)
  Initialize Δτij matrix;
  For each ant k do
    Build a job sequence;
    Evaluate solution k on each objective;
    Update Δτij matrix according to solution k;
    Insert solution k into the quadtree;
  Update τij matrix according to Δτij matrix;
  NC = NC + 1;
```

3 PARALLEL IMPLEMENTATION OF THE ACO

As far as we know, few references can be found about parallel implementations of the ACO at this time. Moreover, work that has been done in this field is mostly related to message passing MIMD architectures, which presents different issues compared to shared memory architectures.

Bullnheimer *et al.* [1998] have proposed a synchronous parallel implementation of the Ant System for the message passing model. The authors outline the considerable cost of communications encountered and the existence of a synchronization procedure that cannot be neglected. Talbi *et al*. [1999] have developed a similar parallel ACO algorithm that is combined with a local tabu search to solve the quadratic assignment problem (QAP).

In this paper we hope to show that a design for a shared memory model is more accurate in order to reduce the cost of the parallelization altough the synchronization procedure cannot be avoided.

There is also an issue about concurrent update of the information that emerges when using a shared memory model but it can be easily resolved so it is possible to achieve good performances in this environment. With the availiability of OpenMP (OpenMP [1998]) it is possible to experiment this approach and the parallelization of the existing sequential C code can be made in an easy, straightforward way.

The reader may notice that our goal in this paper is to improve the execution time of the algorithm without altering its behavior. Improvement of the quality of the solutions found by the ACO with parallel mechanisms is another part of this project and will not be detailed in this paper.

3.1 The "natural" parallelism of the ACO

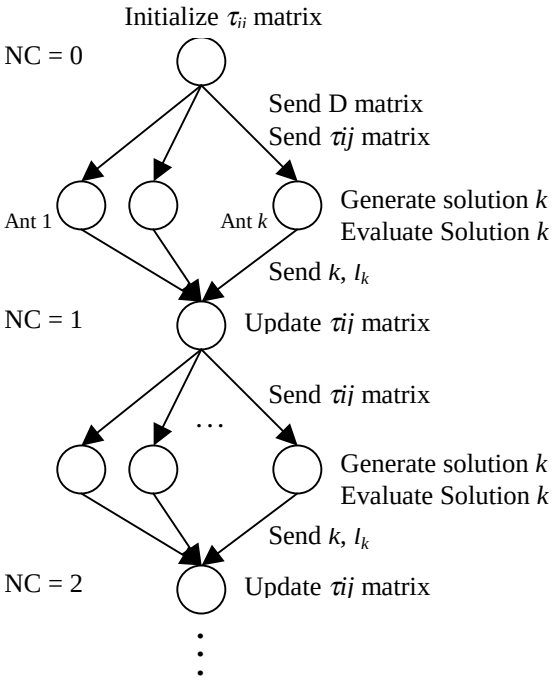
As we can see in Figure 1 the "for loop" is the main part of the algorithm and the source of its complexity (the standard ACO algorithm is $O(n^3)$). In fact, the generation of one solution is of complexity $O(n^2)$ (n is the number of jobs) and since we are in a real scheduling environment, the evaluation has to simulate the industrial process for each solution so it is considerably more time consuming than, for example, the computing of a TSP tour. Besides, these two operations are independant for each ant of a given cycle so they can be easily parallelized.

3.2 Message passing model vs. shared memory model

Figure 2 shows the behavior of an implementation of the ACO based on the parallel synchronous Ant System in a message passing model.

At the beginning of the algorithm, a master process initializes the information, spawns k processes (one for each ant k), and broadcasts the information. At the beginning of a cycle the τij matrix (the pheromone trail) is sent to each process and the computations (generation and evaluation of solutions) are done in parallel. Then, the solutions and their evaluations are sent back to the master, the τij matrix is updated and a new cycle begins by the broadcasting of the updated τij matrix.

Figure 2 **Parallel ACO in a message passing model**



As shown by Bullnheimer *et al.* [1998], if we put aside the communication overhead, this parallelization strategy implies an optimum speedup assuming that there are enough processors available to assign one ant to each processing element. However, this factor cannot be neglected as the high number of communication operations needed (packing and unpacking messages, sending and receiving packets, idle times) are causing a considerable loss in efficiency.

With the recent resurgence of powerful shared memory parallel computers and the availability of OpenMP, which permits the parallelization of the existing sequential C program, it is interesting to experiment a shared memory implementation of the ACO to solve our industrial scheduling problem. In a model where explicit communications are no longer needed it may be possible to obtain good performances with minimal changes to the algorithm.

3.3 Synchronization issue with the updating of the τ_{ij} matrix

At the end of each cycle, the τ_{ij} matrix has to be updated for the ants of the next cycle. Even if in our implementation there is no need of explicit

communications before the updating procedure, the master thread still have to wait for all the processes (ants) of the current cycle to compute and evaluate their job sequences, so this synchronization barrier is independent of the model used and cannot be avoided without altering the original method. In future work we will study the possibility of modifying the algorithm and the frequency of these updates in order to achieve better efficiency without losing searching performance.

3.4 Sharing the load between processors

A first step in the parallelization process could be to naively affect the generation and evaluation of each ant to a different processor (with a *#pragma omp parallel* statement where Number of threads = Number of ants) and by keeping the $\Delta\tau_{ij}$ matrix and the quadtree updatings outside the parallel region since there would be a concurrent update conflict if they were inside. We could also include the updates in the parallel region, in the shared memory, but in a critical zone where only one update at a time could be done. However, in both cases synchronization causes a great loss in efficiency.

Besides, in this situation the chosen number of ants, which is a parameter of the ACO, is limited by the number of processors we have at our disposal, which could be problematic since we could need more ants for bigger problems and we want the application to run on smaller parallel machines. It is then better to share the load between processors in a way to give more than one ant to each processor (with a *#pragma omp parallel for* statement where Number of threads < Number of ants). For this load balancing matter, we can use the static and dynamic solutions provided by OpenMP. This way we get a more efficient implementation and further improvement can be achieved by parallelizing the update of the $\Delta\tau_{ij}$ matrix and of the quadtree, which means having the whole for loop parallelized.

3.5 Updating the $\Delta\tau_{ij}$ matrix concurrently

The most important structures that are used by the ACO are the τ_{ij} matrix, the D matrix, the $\Delta\tau_{ij}$ matrix and the quadtree. The τ_{ij} matrix is updated once each cycle and cannot be parallized without changing the behaviour of the algorithm so it stays in the shared memory during the execution. It is accessed in read mode by the generation function and its update is done by the master thread at the end of each cycle, after the end of the parallel region.

The D matrix is constructed at the beginning of the execution and is never updated, so it stays in the shared memory, as well as all the other parameter variables and structures which are accessed in read mode during the execution of the algorithm.

In the original sequential implementation of the ACO, as well as in the first parallel implementation that was made, the $\Delta\tau_{ij}$ matrix is in the shared memory and is updated by one ant (i.e by an OpenMP thread) at a time (in a critical section of the parallel region) with O(n) operations (where n is the number of jobs). With a memory cost, we can improve execution time by creating one matrix for each thread, i.e. for each independant group of ants. The only use of this matrix is to update the τ_{ij} matrix at the end of the cycle, so we can merge all the $\Delta\tau_{ij}$ matrices (the computationnal complexity of this operation is O(n^2)) in parallel after the main parallel region and before the updating of the τ_{ij} matrix without altering the behavior of the algorithm.

3.6 Updating the quadtree concurrently

A similar issue had to be adressed with the updating of the quadtree. Originally there is only one tree structure that is updated sequentially or in a critical construct, but with another memory cost it is possible to create multiples trees (one quadtree for each processor) and merge them outside the parallel region. The merging procedure can be done after the execution of all the cycles and not at each cycle as with the $\Delta\tau_{ij}$ matrix since the tree is a storing structure and not an information structure that is needed by other parts of the algorithm.

However, there may be a drawback in performance due to the fact that the management of the quadtree is done by dynamic memory allocation, i.e. each insertion procedure implies dynamic creation and destruction of nodes. Even if pointers are private for each quadtree, private dynamic memory allocation is not supported by OpenMP at this time and all the quadtrees are part of the same shared memory heap. This issue and its consequences on the performance of the parallel implementation will be adressed in future work.

With the modifications mentioned above being done, we obtain a parallel implementation of the ACO as shown in Figure 3.

4 RESULTS

The implementation strategy mentioned in section 3 has been experimented by starting from an already existing sequential implementation written in C, adding the appropriate OpenMP directives in it and making the necessary changes that have been discussed.

Figure 3 **Parallel implementation of the ACO**

```
NC = 0;
NumThreads = p;
Initialize τij matrix;
Initialize the p quadtrees;
While (NC < NCMax) and (Not Stagnation Behaviour)
  Initialize the p Δτij matrices;
  Parallel for with p threads
  For each ant k do
    Build a job sequence;
    Evaluate solution k on each objective;
    Update Δτij[p] matrix according to the solution k;
    Insert solution k into the quadtree[p];
  Merge the p Δτij matrices in parallel;
  Update τij matrix according to Δτij matrix;
  NC = NC + 1;
Merge the p quadtrees;
```

Tables 1, 2, 3, and 4 shows the performance of the parallel implementation when 1, 2, 4, 8 and 16 processors are used to process order books of size 50 and 80 with 2 sets of parameters. The number of ants (*k*) has been set to 1000. Figure 4 is a graphical representation of execution times of Table 3.

Table 1 **Results of parallel execution with 50 jobs, 100 cycles and 1000 ants**

Number of processors	Execution time (sec)	Speedup	Efficiency
1	572	-	-
2	309	1.85	0.93
4	167	3.43	0.86
8	112	5.11	0.64
16	125	4.58	0.29

Table 2 **Results of parallel execution with 50 jobs, 200 cycles and 1000 ants**

Number of processors	Execution time (sec)	Speedup	Efficiency
1	1136	-	-
2	634	1.79	0.90
4	349	3.25	0.81
8	231	4.91	0.61
16	267	4.25	0.26

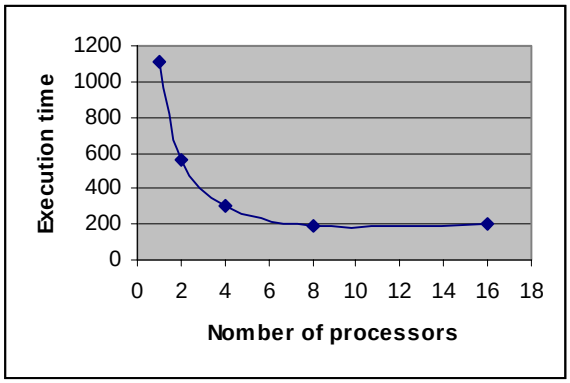
Table 3 **Results of parallel execution with 80 jobs, 100 cycles and 1000 ants**

Number of processors	Execution time (sec)	Speedup	Efficiency
1	1111	-	-
2	564	1.97	0.98
4	305	3.64	0.91
8	187	5.94	0.74
16	204	5.45	0.34

Table 4 **Results of parallel execution with 80 jobs, 200 cycles and 1000 ants**

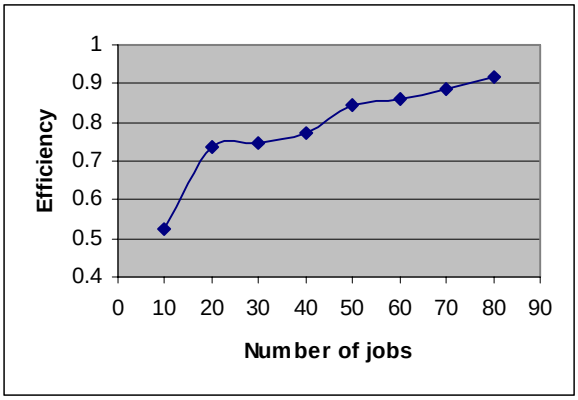
Number of processors	Execution time (sec)	Speedup	Efficiency
1	2181	-	-
2	1152	1.89	0.95
4	613	3.56	0.89
8	381	5.72	0.72
16	429	5.08	0.32

Figure 4 **Execution time with 80 jobs, 100 cycles and 1000 ants**



Regarding problem size, Tables 1 and 3 show that we get better efficiency with 80 jobs than with 50, as we supposed, when the same parameters are applied (Figure 5 shows efficiency with 8 different sizes of order books). We believe that we would need bigger order books to properly see the benefits that can be gained from parallelism, but actual program limitations (mostly the complex industrial evaluation process) hinders the use of bigger order books. A first step in this project was to study feasibility and performance on existing and studied big problems, so in future work we will expand the application to process more than 80 jobs.

Figure 5 **Efficiency with 4 processors, 100 cycles, 1000 ants and order books varying from 10 to 80**



Another issue that we wanted to adress in this work is the penalty in efficiency that is caused by the synchronizations due to the updating of the τ_{ij} matrix. For this matter the application was executed with variations to the number of cycles while keeping the same other parameters. Results show that when NC is increased for the same problem, which implies more updatings of the τ_{ij} matrix and more synchronizations, there is a loss in efficiency.

Overall results shows that our parallel implementation of the ACO for this problem leads to the obtention of significant speedups. However, experiments are not as convincing as we expected, especially with 8 and more processors. The degration of efficiency obtained when increasing the number of processors is faster than we expected and the loss of performance associated with the increase of the number of cycles is smaller. That statement is also true as we increase the number of ants. In fact, when we increase the number of ants to 10 000, which is a high number compared to the usual chosen one, we get better efficiency, but not as high as one could expect when 8 and 16 processors are used (0.99 for 2 processors, 0.92 for 4, 0.76 for 8 and 0.36 for 16).

Further experiments and studies will lead us to understand better the mechanisms of the implementation, the effects of modifying the algorithm parameters and the influence of software and hardware elements that were used for the development. The main issues that we may adress in short term work are :

- The effects of dynamic memory allocation in parallel with OpenMP
- The SGI Origin2000 computer, its CC-NUMA architecture, the data structures used in the application and the way the memory is used
- Better knowledge of the OpenMP environment and of the use of its directives
- The design of the actual application code
- More extensive experimentations with different parameter settings

5 CONCLUSION

In this paper we have presented a shared memory parallel implementation of an Ant Colony Optimization metaheuristic that is applied to an industrial scheduling problem and we have shown the main issues that had to be adressed during the parallelization process. The nature of the ACO and the functionnality offered by OpenMP made the transition from sequential to parallel easier and straightforward while some changes had to be made to the algorithm and to the program to obtain the level of efficiency that we achieved.

Our aim was to increase the execution time of the algorithm. The resulting implementation has shown that is was possible to design an efficient parallel Ant Colony Optimization metaheuristic in a shared memory model with OpenMP. It also has shown some limitations as we increased the number of processors used in the application and the causes of those drawbacks should be analysed in near future.

In an another part of this project we plan to exploit parallelism potential of the ACO in a way that will improve its solution searching capabilities without increasing its execution time. It is likely that this goal will imply a model of co-evolution of many ant colonies, which means a higher level of parallelization, the possible use of a message passing model and a resulting implementation that mixes MPI and OpenMP.

ACKNOWLEDGEMENT

This work was partly supported by the Centre Lorrain de Calcul à Hautes Performances (Centre Charles Hermite : CCH) and by the Centre Informatique National de l'Enseignement Supérieur (CINES).

REFERENCES

Bullnheimer B., Kotsis G., Strauss C. [1998], *Parallelization Strategies for the Ant System*. In R. De Leone, A. Murli, P. Pardalos, and G. Toraldo, editors, High Performance Algorithms and Software in Nonlinear Optimization, volume 24 of Applied Optimization, pages 87-100. Kluwer : Dordrecht.

Colorni A., Dorigo M., Maniezzo V. [1991], *Distributed optimization by ant-colonies*, in Proceedings of the European Conference on Artificial Life (ECAL'91), edited by F. Varela and P. Bourgine, 134-142, Cambridge, Mass, USA, MIT Press.

Colorni A., Dorigo M., Maniezzo V., Trubian M. [1994], *Ant system for job-shop scheduling*, Belgian Journal of Operations Research (JORBEL), Statistics and Computer Science, 34, 1, 39-53.

Deneubourg J.L., Pasteels J.M., Verhaeghe J.C. [1983], *Probabilistic behaviour in ants: A strategy of errors?*, Journal of Therical Biology, 105, 259-271.

Deneubourg J.L., Goss S. [1989], *Collective patterns and decision-making*, Ethology & Evolution, 1, 295-311.

Dorigo M. [1992], Optimization, learning and natural algorithms, Ph.D. Thesis, Politecnico di Milano, Italy.

Dorigo M., Di Caro G. [1999], *The Ant Colony Optimization Meta-Heuristic*', In: D. Corne, M. Dorigo and F. Glover Editors, New Ideas in Optimization, McGraw-Hill.

Dorigo M., Gambardella L.M. [1997], *Ant colonies for the traveling salesman problem*, BioSystems, 43, 73-81.

Dorigo M., Maniezzo V., Colorni A. [1991], Positive feedback as a search strategy, Technical Report No 91-016, Politecnico di Milano, Italy, 20 pages.

Finkel, R. A., Bentley, J. L. [1974], *Quad trees, a data structure for retrieval on composite keys*, Acta Informatica 4, 1-9.

Goss S., Beckers R., Deneubourg J.L., Aron S., Pasteels J.M. [1990], *How trail laying and trail following can solve foraging problems for ant colonies*, In: Behavioural Mechanisms of Food Selection, R.N. Hughes ed., NATO-ASI Series, vol. G20, Berlin: Springler-Verlag.

Gravel M., Price W., Gagné C. [2001], *Scheduling continuous casting of aluminum using a multiple-objective ant colony optimization metaheuristic*, Document de Travail no. 2001-004, Faculté des Sciences de l'Administration, Université Laval, Québec, Canada. (submitted for publication)

Habbas Z., Krajecki M., Singer D. [2000], *Domain Decomposition for Parallel Resolution of Constraint Satisfaction Problems with OpenMP*, In: Proceedings of The Second European Workshop on OpenMP, Edinburgh, Scotland, 1-8.

OPENMP ARCHITECTURE REVIEW BOARD [1998], OpenMP C and C++ Application Program Interface Version 1.0, http://www.openmp.org.

Talbi E-G., Roux O., Fonlupt C., Robillard D. [1999], *Parallel ant colonies for combinatorial optimization problems*, BioSP3 Workshop on Biologically Inspired Solutions to Parallel Processing Systems, in IEEE IPPS/SPDP'99 (Int. Parallel Processing Symposium / Symposium on Parallel and Distributed Processing, edited by Jose Rolim, Lecture Notes in Computer Science Vol., Springer-Verlag, San Juan, Puerto Rico, USA.