Available online at www.sciencedirect.com
SCIENCE DIRECT®
Parallel Computing 31 (2005) 1034–1047
PARALLEL COMPUTING
www.elsevier.com/locate/parco

A hierarchical distributed-shared memory parallel Branch&Bound application with PVM and OpenMP for multiprocessor clusters

Rocco Aversa, Beniamino Di Martino *, Nicola Mazzocca, Salvatore Venticinque

Dip. Ingegneria dell'Informazione, Seconda Universitá di Napoli, DII, Real Casa dell'Annunziata, via Roma 29, 81031 Aversa(CE), Italy

Received 11 October 2004; received in revised form 15 February 2005; accepted 5 March 2005
Available online 17 October 2005

Abstract

Branch&Bound (B&B) is a technique widely used to solve combinatorial optimization problems in physics and engineering science. In this paper we show how the combined use of PVM and OpenMP libraries can be a promising approach to exploit the intrinsic parallel nature of this class of application and to obtain efficient code for hybrid computational architectures. We described how both the shared memory and the distributed memory programming models can be applied to implement the same algorithm for the inter-nodes and intra-node parallelization. Some experimental tests on a local area network (LAN) of workstations are finally discussed.

Keywords: Hybrid applications; OpenMP; MPI; Branch&Bound

* Corresponding author. Tel.: +39 081 5010282; fax: +39 081 5037042.
E-mail addresses: rocco.aversa@unina2.it (R. Aversa), beniamino.dimartino@unina.it (B. Di Martino), n.mazzocca@unina.it (N. Mazzocca), salvatore.venticinque@unina2.it (S. Venticinque).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2005.03.010

1. Introduction

Branch&Bound (B&B) is a technique widely used to solve combinatorial optimization problems in physics and engineering science. Furthermore, Branch&Bound (B&B) applications represent a typical example of irregularly structured problems whose parallelization using hierarchical computational architectures (e.g. clusters of SMPs) involves several issues, such as the sharing of global computation state and the dynamic workload balancing among nodes. In this paper we show how the combined use of PVM and OpenMP libraries can be a promising approach to exploit the intrinsic parallel nature of this class of application and to obtain efficient code for hybrid computational architectures. Our strategy to yield an effective distributed version of a given Branch&Bound (B&B) application for a distributed multiprocessor architecture was driven by the following rules: no significant variations in the original algorithm structure; reusing a large part of the available code; exploiting the hybrid computational characteristics of the target system. According to this second issue, we chose to combine in our parallel version a coarse grain parallelization technique, using a PVM solution of the B&B application [1] based on the *coordinator/workers* paradigm, together with a finer grain parallelization approach that, using OpenMP primitives, introduces an additional dynamic and efficient workload distribution among the shared memory nodes of the system. The remainder of the paper proceeds as follows: a description of the programming models adopted in this work is provided in Section 2. A conceptual description of B&B optimization technique, together with the relating parallelization issues is presented in Section 3. Section 4 describes the parallel implementation, illustrating the shared memory version and its integration into the original distributed version are presented. In Section 5 we discuss results of tests on a local area network (LAN) of workstations. Finally we give some concluding remarks.

2. Parallel architectures and programming models

The goal of parallelization is to obtain high performance keeping low programming effort and exploiting the hardware resources of the target parallel machine. Many parallel programming environments allow the developer to approach the parallelization by using different programming paradigms, trying to match the intrinsic parallelism of the algorithm with the available hardware. For example, a shared memory implementation of a parallel program is expected to fit well a SMP (Symmetric Multi Processor) machine, while a message passing model will be preferred to take advantage of the characteristics of a cluster of workstations. Whatever kind of target architecture, an efficient parallelization strategy needs: to ensure a balanced distribution of work among processors, to reduce the amount of inter-processor communication, and to keep the overheads of communication, synchronization and parallelism management low [2]. Complying with these requirements, today, for a great deal, is still on the programmer's responsibility.

In the *shared memory approach* the shared address space programming model assumes one or more threads of control, each operating in an address space divided into a region shared between threads, and another one that is private to each thread. In particular the program accesses and updates shared variables simply by using them in expressions and assignment statements. The principal programming environments supporting the shared memory paradigm include: support for data parallel programming (single threaded, global name space, and loosely synchronous parallel computation); open interfaces and interoperability with other languages and other programming paradigms (e.g., message passing using MPI). Another important goal is to achieve code portability across a variety of parallel machines, where portability means not only that parallel programs compile on different target machines, but also that a highly efficient program on one parallel machine is able to achieve reasonably high efficiency on another parallel machine with a comparable number of processors. Other features are automatic support for multithreaded execution, loop tiling, code restructuring and efficient data distribution.

Examples of available compilers which support shared memory programming are HPF [3] and OpenMP compliant platforms [4]. HPF is an high performance Fortran Compiler that extends Fortran 90 language providing access to high-performance architecture features while maintaining portability across platforms. The programmer is often provided with an Application Programming Interface (API) that can be used to explicitly direct multi-threaded, shared memory parallelism. OpenMP is an Open specifications for Multi Processing via collaborative work with interested parties from the hardware and software industry, government and academia. It is composed of three primary API components: Compiler Directives, Runtime Library Routines and Environment Variables. The API is specified for C/C++ and Fortran. Multiple platforms have been implemented including most Unix and Windows platforms. An example of JAVA implementation is JOMP (JAVA OpenMP) [5], it is composed of a open JAVA pre-compiler and JAVA runtime API.

Message passing models assume a collection of processes, each operating in a private address space and each able to name the other processes. The normal uniprocessor operations are provided on the private address space, in program order. The additional operations, send and receive, operate on the local address space and the global process space: send transfers data from the local address space to a process; receive accepts data into the local address space from a process. Each send/receive pair is a specific point-to-point synchronization operation. By means of these simple mechanisms many message passing languages and runtime systems offer primitives for collective synchronization and communication such as barriers or broadcast and multicast. Some examples are MPI [6], PVM [7]. Furthermore a lot of platforms are available today targeted to systems which are distributed on wide area networks (GRID). Some examples exploit web computing [8,9], mobile code based technology [10], or reuse classical and standard protocols and languages [11,12]. We adopt in this paper both the shared memory and Message passing programming paradigms to parallelize an irregular application targeted to cluster of SMP workstations. Our hybrid implementation exploits the Message passing model

for inter-node parallelization and the shared memory model within the single workstation for intra-node parallelization.

3. The Branch&Bound parallel application

A *discrete optimization problem* consists in searching the optimal value (maximum or minimum) of a function $f : \vec{x} \in \mathscr{Z}^n \rightarrow \mathscr{R}$, and the solution $\vec{x} = \{x_1, \ldots, x_n\}$ in which the function's value is optimal. $f(\vec{x})$ is said *cost function*, and its domain is generally defined by means of a set of m constraints on the points of the definition space. Constraints are generally expressed by a set of inequalities:

$$\sum_{i=1}^{n} a_{i,j} x_i \leqslant b_j \quad \forall j \in \{1, \ldots, m\} \tag{1}$$

and they define the set of feasible values for the x_i variables (the *solutions space* of the problem). Branch&Bound is a class of methods solving such problems according to a *divide&conquer* strategy. The initial solution space is recursively divided in subspaces, until attaining to the individual solutions; such a recursive division can be represented by a (abstract) tree: the nodes of this tree represent the solution subspaces obtained by dividing the parent subspace, the leaf nodes represent the solutions of the problem, and the tree traversal represents the recursive operation of dividing and conquering the problem.

The method is enumerative, but it aims to a non-exhaustive scanning of the solutions space. This goal is achieved by estimating the best feasible solution for each subproblem, without expanding the tree node, or trying to prove that there are no feasible solutions for a subproblem, whose value is better than the *current* best value. (It is assumed that a best feasible solution *estimation function* has been devised, to be computed for each subproblem.) This latter situation corresponds to the so called *pruning* of a search subtree. B&B algorithms can be parallelized at a *fine* or *coarse grain* level. The fine-grain parallelization involves the computations related to each subproblem, such as the computation of the estimation function, or the verification of constraints defining feasible solutions. The coarse grain parallelization involves the overall tree traversal: there are several computation processes concurrently traversing a different branch of the search tree. In this case the effects of the parallelism are not limited to a speed up of the algorithmic steps. Indeed, the search tree explored is generally different from the one traversed by the sequential algorithm. As a result the number of explored nodes can be greater than in the sequential case and the resolution time could increase in a not predictable manner. As it has been demonstrated in [13], this approach exhibits *anomalies*, so that the parallelization does not guarantee an improvement in the performance. For most practical problems, however, the bigger the problem size, the larger are the benefits of the parallel traversal of the search tree. Parallel B&B algorithms can be categorized on the basis of four features [14]:

(1) how information on the global state of the computation is shared among processors (we refer to such information as the *knowledge* generated by the algorithm);
(2) how the knowledge is utilized by each process;
(3) how the workload is divided among the processes;
(4) how the processes communicate and synchronize among them.

The knowledge is *global* if there is a single common repository for the state of the computation at any moment, accessed by all processes, otherwise it is *local*. In this latter case, processes have their own knowledge bases, which must be kept consistent to a certain degree to speed up the tree traversal and to balance the workload. With respect to the knowledge use, the algorithms are characterized by: (1) the *reaction strategy* of processes to the knowledge update (it can range from an instantaneous reaction, to ignore it until the next decision is to be taken); (2) the *dominance rule* among workers (a worker *dominates* another if its solutions best value is better than the lower bound on the solutions of the other): it can be *partial* or *global*, if a node can be eliminated only by a dominant node belonging to the same subtree traversed by a process, or by any other dominant node; (3) the *search strategy*, that can be *breadth*, *depth* or *best* first. With regard to the workload division, if all generated subproblems are stored in a common knowledge base, to each process that becomes idle the most promising subproblem is assigned (on the basis of an heuristic evaluation). If the state of the computation is distributed among processes (local knowledge), then a *workload balancing* strategy has to be established, consisting of a relocation of subproblems not yet traversed.

The parallel algorithm we present solves the (0–1) *knapsack problem*, which can be stated as follows [15]:

$$\text{maximize} \quad \sum_{i=1}^{n} c_i x_i \tag{2}$$

$$\text{with} \quad x_i \in \{0,1\} \quad \forall i \in \{1,\ldots,n\}$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_{i,j} x_i \leqslant b_j \quad \forall j \in \{1,\ldots,m\} \tag{3}$$

where $a_{i,j}$ and b_j are positive integers.

4. The hybrid implementation

In the *coordinator/workers* concurrent programming model, a coordinator process spawns a set of worker processes, which perform the actual computation; the coordinator also manages the sharing of the global knowledge among the workers. The structure of the application is depicted in Fig. 1. The algorithm we chose is exploited both at distributed level by a PVM implementation and on the single SMP node by the OpenMP support.

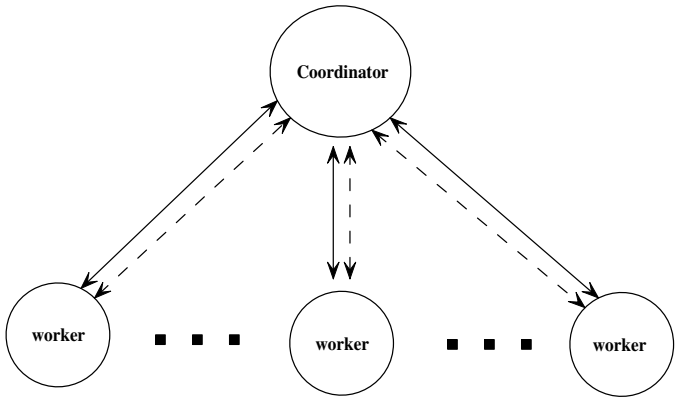

Fig. 1. Structure of the coordinator worker algorithm.

At distributed memory the workers are PVM processes spawned on the available nodes, while at shared memory we have many workers threads inside a process, scheduled on different processors.

The parallelization process follows the following phases:

(1) *The decomposition* of the problem consist of the initial division of the original solution space. The solution space is splitted by the coordinator many times till the number of subspaces is equals to the number of workers.
(2) *Assignment.* The coordinator assigns a single task to each worker. Subsequently the task could be splitted again in order to perform new assignments whether some workers will be idle.
(3) *The orchestration* is very simple. Each worker explores the solution space of the problem by splitting and solving the assigned one. When a better solution is found it is communicated to the other workers (which reside both on the same node and on the remote ones). When a worker is idle it asks for new tasks to the local workers before and to the remote ones later.
(4) *The mapping* is performed by the coordinator that spawns the PVM processes across the nodes. The master thread inside each process generates a pool of threads, one for each processor available on a SMP node.

4.1. The PVM coarse grain parallelization

The implementation consists of the following phases:

- a *coordinator* process produces P instances of a *worker* process, decomposes the assigned problem in P disjoint subproblems and assigns them to the workers;
- each worker explores its own subtree with a *depth-first* strategy, and updates its local current best value, sending it to the coordinator, when it generates a feasible solution;

- when the coordinator receives a local current best value, it compares it with the global current best value and eventually updates and broadcasts it to the workers.

The load balancing among the workers is accomplished with the following strategy:

- when a worker completes the exploration of its subtree, it sends a message to the coordinator, and waits for a new load share from it;
- the coordinator manages a list of idle workers: when it is not empty, it polls active workers for a share of the load, until it receives a positive answer, then it assigns the share to the first idle process in the list.

Thus, the coordinator is in charge of managing the updates and the broadcast of the current best value, and of balancing the load among workers. The presence of a coordinator also allows the detection of the *termination condition*. This is verified when all workers are idle at the same time, and there are no messages carrying work units, which have been sent but not received yet. Since the coordinator holds the global state of the computation (lists of idle workers and of workload messages), it is able to detect this situation. The characteristics of the algorithm at distributed level, with respect to the categorization of the B&B algorithms presented in the previous section, are the following: (1) each worker has its own knowledge base (local knowledge); (2) processes react instantaneously to knowledge updates (the *signal* primitives of the PVM environment are used for this purpose); (3) a global dominance rule is adopted, since the current optimal value is broadcasted to all processes as soon as it is updated; (4) the search strategy is *depth-first*, as it is more suited in the case of local knowledge; (5) with respect to workload sharing between processes, load balancing is provided, activated in presence of an idle process, since there is a local knowledge sharing; (6) with regard to synchronization, the algorithms are asynchronous, since there is no synchronous exchange of information and workload between executors, but this exchange is performed on the basis of executor's local events, and so asynchronously with respect to the other executors.

4.2. The OpenMP fine-grain parallelization

The shared memory extension of the algorithm allows many threads to solve in parallel the problem assigned to a pvm worker process applying recursively the divide and conquer strategy. Each thread begins the traversal of the subtree assigned to the pvm process from a different node at a deeper level. The solution space is an array of boolean values specified till an early index *lsc*; a feasible solution is the same array specified till the full dimension *N*. The not fully specified array represents also a branch of the tree. The solution space is splitted by incrementing the index *lsc* and obtaining two new arrays with the the value of index lsc-1 equals to 1 and 0. As the PVM workers, the threads divide the subspaces assigned to them, but put the new subproblems in the same shared bag. Every time the solution space is divided into two new problems the thread pushes the first into the common bag

and continues to explore the second one. When a branch of the tree is pruned a new task is taken from the common bag. When a new optimum is found the global optimum is updated. The global optimum is shared among the threads in order to optimize the tree branching. As the bag is shared among all the workers we do not need a inner workload balancing strategy. When the bag is empty all the threads becomes IDLE and a new problem is asked by the master thread to the PVM coordinator process. The computation of the feasible solution is performed in a OpenMP parallel section where the constraints and the function loads are shared, while the computations are executed on private data. The different threads concur to write the same shared data when:

- a new local optimum is found an the global one needs to be updated;
- a new subproblem is pushed into the bag;
- a new subproblem is taken from the bag.

In these three situations we need to define an OpenMP critical sections in order to make atomic each of these actions. In order to explain as the OpenMP support was exploited we describe here how the sequential version has been extended. As it is shown Fig. 2, this kind of parallelization is straightforward and very simple to implement.

However in this case we have three pure parallel computing sections (without any concurrence on shared values):

(1) the computation of the new optimum;
(2) the verification of the feasibilities;
(3) the branching of the tree.

We have three critical section which introduce a loss of performance when the different threads are executing the same part of the code:

(1) the updating of the global optimum;
(2) the pushing of a new task in the shared bag;
(3) the extraction of a task from the shared bag.

The critical sections deal with three different and not overlapped domains. In fact the bag of tasks is a buffer accessed according to a first-in-first-out strategy. It means that two threads could concur to write the same data just when they are executing in the section identified by the same label `opt`, `push` or `pop`. First of all we should consider that for each iteration we are sure to enter at least one critical section. In fact when a new optimum is found in the space of solutions two critical section are crossed:

- we enter the first critical section in order to update the new optimum;
- and we divide the problem and fill the bag in order to identify the final solution or a better value.

```
main() {
 #pragma parallel omp shared(opt, bag)
 {
while(!bagEmpty)
     {
        x = select_next_node(N);
        verify_feasibility(x);
        if "x isfeasible solution"
        #pragma omp critical (opt)
          {
            update_supposed_best
            send(new_supposed_best,coor);
           }
        else branch
      if (!branch)
        #pragma omp critical (push)
       {
           fill_the_bag      //divide: search new feasible solutions
       }
      else
       #pragma omp critical   (pop)
       {
        if "other nodes to explore" //conquere feasible solutions
            take_from_the_bag
       }
     }//end while
}// end omp parallel

I AM IDLE !!!!
 }
```

Fig. 2. The OpenMP implementation of the parallel algorithm with a shared bag of tasks.

Instead when the new solution is not feasible or it is worse than the current optimum only one critical section is executed in order to take a new problem from the bag. About the scalability of the problem, when the problem size increases we see above all an exponential growth of the solution space, whose main effect is a bigger number of iterations. It means that the execution time of the critical section increases too much. Finally it easy to foresee that the degree of concurrence increases with the number of threads. In order to overcome this kind of troubles we tried to optimize the implementation of the algorithm described above. It is possible to reduce the concurrence among the threads by defining a shared, but distributed bag. If each thread is owner of a private repository of tasks we have at the most one execution of critical code. We mean that the critical code is executed only when a better optimum and a feasible solution are found. The update of the global value is needful in order to perform a more consistent pruning of the tree. When the problem size or the number of threads increase the number of updates does not change in the same way.

The number of executions of the critical code depends just by the input data and the way according to which the solution space is divided among the threads. The pseudo code, showed in Fig. 3, describes the implementation of the optimized approach.

As it can be seen we implemented the distributed bag as an array of pockets. Each thread picks out of its own pocket. The index of each pocket is associated to the identifier of the owner OpenMP thread. In order to perform a load balancing among

```
main() {
 #pragma parallel omp shared(opt, bag[])
 {
while(!bagEmpty)
     {
        x = select_next_node(N);
        verify_feasibility(x);
        if "x isfeasible solution"
        #pragma omp critical
          {
            update_supposed_best
            send(new_supposed_best,coor);
           }
        else branch
      if (!branch){
           fill_the_bag[myid]      //divide: search new feasible solutions

      else
        if "other nodes to explore" //conquere feasible solutions
        {
            omp_unset_lock(locks[myid]);
            take_from_the_bag[myid]
            omp_unset_lock(locks[myid])
        }
        else
        {
            otherid=(myd+1)%n_threads
            while((otherid!=myd)|(got_new_problem))
             {
               omp_set_lock(locks[otherid]);
               take_from_the_bag[otherid]
               omp_unset_lock(locks[otherid]);
               otherid=(otherid+1)\%n_threads
             }
          }

     }//end while
}// end omp parallel

I AM IDLE !!!!
 }
```

Fig. 3. The optimized implementation of OpenMP based algorithm.

the threads, when the owned pocket is empty, the worker tries to pull a task from the other pockets. The concurrent access to the same pocket takes place very seldom. Anyway we must assure that the access to the same pocket, in order to get a new problem, is exclusive among the threads.

The labelled critical section used in the previous implementation is not suited to solve this kind problem. In fact the label is a static attribute for the section while the pocket accessed by a thread when it enter the critical section is known only at run-time. The collision caused by the concurrent access to the particular pocket is avoided by an array of locks. A thread reserves and releases the lock i every time it needs to to take a task from the pocket i. If the bag is empty the thread becomes idle and wait for the others workers at the end of the parallel section. Hence the process become idle and asks for new load to the remote coordinator.

The characteristics of the algorithm at this level, with respect to the classification of the B&B algorithms presented in the previous section, are the following: (1) each worker has its own knowledge base (private knowledge, and private space of solution) and a shared one; (2) processes react instantaneously to knowledge updates (all the relevant information is stored in shared data); (3) a global dominance rule is adopted, since the current optimal value is assigned to shared global one; (4) the search strategy is *depth-first*, as it is more suited in the case of local knowledge; (5) with respect to workload sharing between processes, load balancing is provided by a coordinated access to shared bag of tasks; (6) with regard to synchronization, the algorithms are asynchronous, since there is no synchronous exchange of information and workload between executors. This exchange is performed on the basis of local events that allow the mutual exclusion in upgrading shared information.

5. Experimental results

The system used as hardware platform for our experimental executions is the Cygnus cluster of the Parsec Laboratory, at the Second University of Naples. This is a Linux cluster with four SMP nodes equipped with two Pentium Xeon 1 GHz, 512 MB RAM and a 40 GB HD, and a dual-Pentium Xeon 3.2 GHz front end. The nodes are interconnected by 100 Mbit Ethernet switch. For the experiments we report in this Section, we have installed, on our system, the ROCKS LINUX distribution supported by NPACI, the Intel C++ Compiler version 8.0 and the PVM environment version 3.4.4. We executed at first the sequential version of our application and the different parallel implementations.

The results are showed in Fig. 4 and reported in Table 1.

About Table 1, on the first two columns we have the pure OpenMP implementation using a shared bag and the pure OpenMP implementation using the distributed bag, both executed on a single node. On the last columns we have the pure PVM and hybrid implementations executing on four nodes. According to the irregular behaviour of the application, due to the dependence of the algorithm on the input data, we get very different results, even with the same dimension of the original problem ($N = 40$ unknowns). The collected results refer to three different input data set.

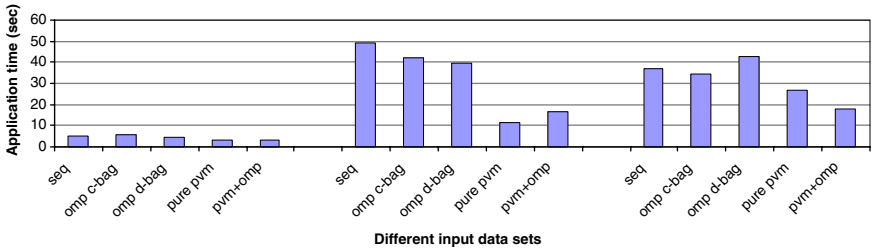

Fig. 4. The execution time for different set of input data.

Table 1
Performance results for different implementations

Input data	*seq*	*omp_lp*	*omp*	*pure pvm*	*pvm + omp*
1	4779	5681	4403	3315	3363
	1853657	2930622	2812890	4831090	7422951
2	49427	42833	39226	11719	16323
	19041082	21515089	25490114	17169639	38584062
3	36887	34204	42661	26816	17492
	14656792	18239698	28736101	40089176	43818122

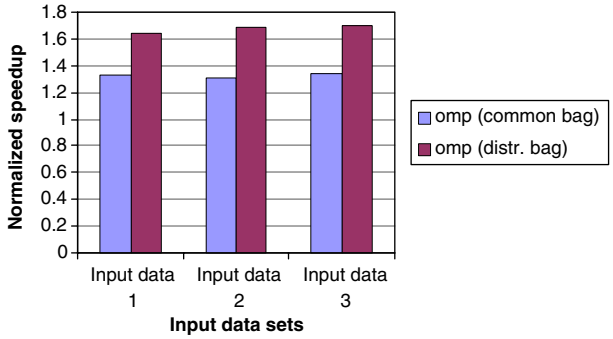

Fig. 5. Normalized speedup for the shared bag and distributed bag OpenMP implementations.

On the first row, for each input data set, it is reported the application time required by the execution for the different implementations. On the second row, it is reported the total number of subproblems which were explored. It is reasonable that the execution time is strictly related to the number of problems which are effectively solved. The fastest execution, with the first data set, is due to the effect of a relevant pruning of the tree. Whatever a relevant pruning does not occur in the first phase of the computation, the number of solutions explored can increase a lot, as it is happens for the second and the third data set.

About the improvement or the loss in performance that we can appreciate when a greater number of processes is involved, they depend by how the solution space is divided among the processes.

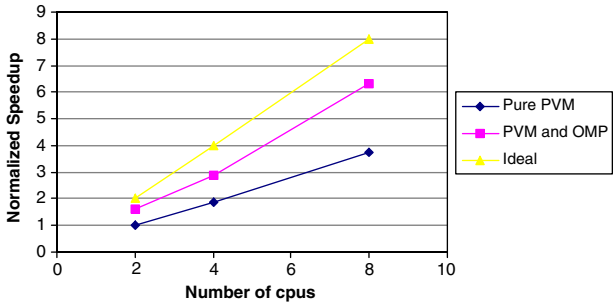

Fig. 6. Normalized speedup for the pure PVM and hybrid parallel implementations.

As the classical speedup measure is not meaningful since the different executions work with a different number of solutions, we will provide a normalized measure of the speedup in order to remove the dependence on the irregularity of the problem.

In Fig. 5 we compare the speedup of the pure OpenMP version executed on a single node. The optimized implementation perform always better already with two treads.

In Fig. 6 it is showed the normalized speedup for the execution of the *pure PVM* implementation and the *hybrid* one. It scales always well.

6. Conclusion

We presented a strategy for the parallelization of Branch&Bound algorithms for distributed memory of SMP architectures. It is based on a coordinator/worker paradigm at distributed level supported with a dynamic workload balancing facility. In order to take advantage of the SMP architecture of the computational nodes, according to a similar approach, we recursively defined a master thread and a pool of workers which share a bag of tasks. Finally, we described a PVM-OpenMP implementation and presented some experimental results discussing the irregular behaviour of the Branch&Bound application. Future works will explore the extension of the proposed parallelization approach to a target platform consisting of a wider number of nodes, also geographically distributed, supported by a GRID infrastructure. We were already able to evaluate the overhead introduced when the application is launched by the globus v. 2.0 gatekeeper. We estimated a growth of the user time equals to 0.2 s about.

References

[1] B. Di Martino, N. Mazzocca, S. Russo, Paradigms for the parallelization of Branch&Bound algorithms, PARA, 1995.

[2] D.E. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: a Hardware/software Approach, Morgan-Kaufman Publisher, 1998. August.

[3] P. Mehrotra, J. Van Rosendale, H. Zima, High performance Fortran: history status and future, Parallel Computing (1997), Special Issue on Languages and Compilers for Parallel Computers.
[4] L. Dagum, R. Menon, OpenMP: An industry-standard API for shared-memory programming, Computational Science & Engineering 5 (1) (1998).
[5] J.M. Bull, M.E. Kambites, JOMP—an OpenMP-like interface for Java, in: Proceedings of the ACM 2000 Java Grande Conference, June 2000, pp. 44–53.
[6] G. William, E. Lusk, N. Doss, A. Skjellum, A highperformance, portable implementation of the MPI message passing interface standard, Parallel Computing 22 (6) (1996) 789–828.
[7] The PVM Concurrent Computing System: Evolution, Experiences and Trends V. Sunderam, J. Dongarra, A. Geist, R. Manchek, Parallel Computing 20 (4) (1994) 531–547.
[8] A. Baratloo, M. Karaul, Z. Kedem, P. Wycko, Charlotte: Metacomputing on the Web. in: Proceedings of the 9th International Conference on Parallel and Distributed Computing systems, Dijon, France, September 1996.
[9] B.O. Christiansen, P. Cappello, M.F. Ionescu, M.O. Neary, K.E. Schauser, D. Wu, Javelin: internet-based parallel computing using java, Concurrency: Practice and Experience 9 (11) (1997) 1139–1160.
[10] P. Evripidou, C. Panayiotou, G. Samaras, E. Pitoura, The PaCMAn metacomputer: parallel computing with java mobile agents, Future Generation Computer Systems Journal 18 (2) (2001) 265–280, Special Issue on Java in High Performance Computing.
[11] Globus: A Metacomputing Infrastructure Toolkit I. Foster, C. Kesselman, International Journal of Supercomputer Applications 11 (2) (1997) 115–128, Provides an overview of the Globus project and toolkit.
[12] MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface N. Karonis, B. Toonen, I. Foster, Journal of Parallel and Distributed Computing (2003).
[13] H.T. Lai, S. Sahni, Anomalies in parallel Branch&Bound algorithms, Communications of the ACM 27 (6) (1984) 594–602.
[14] H.W.J. Trienekens, Parallel Branch&Bound Algorithms, Ph.D. Thesis at Erasmus Universiteit-Rotterdam, November 1990.
[15] C. Ribeiro, Parallel computer models and combinatorial algorithms, Annals of Discrete Mathematics, North-Holland, 1987, pp. 325–364.