Load Balancing Strategy and Priority of Tasks in Distributed Environments

Salah DOWAJI and Catherine ROUCAIROL
Université de Versailles-St Quentin - Laboratoire PRiSM
45, avenue des Etats-Unis
78000 Versailles FRANCE
e-mail: {sad,roucairo}@prism.uvsq.fr

Abstract

After we have showing that partial cost is not the only important parameter to balance subproblems between queues of best-first B&B distributed algorithms, we introduced an other notion of priority between subproblems that takes into account not only their partial cost but also their capacity to generate other subproblems.
We developed three load balancing strategies using this new notion of priority, with a B&B solution of the Vertex Cover Problem (VCP).
The implementation of the VCP was carried out in a network of heterogeneous Unix workstations used as a single parallel computer through a Parallel Virtual Machine (PVM) software system.
To show the advantages of the new notion of priority, we solved the VCP for graphs of up to 130 nodes with average degree of 50%. Also, we were able to achieve a remarkable load distribution and a good restriction of communication costs over all the different machines in the system.

1 Introduction

Branch and Bound (B&B) algorithms are the most popular techniques used to solve NP-hard combinatorial optimization problems. With this kind of problems, too many tasks/subproblems (represented as nodes in the B&B tree) are created dynamically so that load balancing becomes an important factor to optimize performance. In such cases, subproblems are classified in function of their priorities into several queues, and solutions may be reached more efficiently if priorities given to subproblems by queues are taken into consideration during the parallel execution of the algorithm.

Existing load balancing schemes use partial cost to balance subproblems generated by best-first B&B distributed algorithms different queues.

In this article, we show that using partial cost might lead to a big accumulation of high priority tasks on a few processors, while the other processors continue to work with low priority subproblems. In such a situation, an other notion of priority is needed. It must take into account not only the partial cost of subproblems but also their capacity to generate other subproblems.

Our new notion of priority is applicable to various applications of which those using best-first B&B algorithms and those from the area of Operational Research. We chose to evaluate the strategy with a B&B solution of the Vertex Cover Problem (VCP).

Our aim with using the B&B algorithm is not to study the parallelization of this method, but rather to develop an efficient load strategy that uses priority of tasks.

The implementation of the VCP was carried out in a network of heterogeneous Unix workstations used as a single parallel computer through a Parallel Virtual Machine (PVM) software system. All experiments in this article have been done on a 3 "SUN" workstations (1 SPARCstation ELC and 2 SPARCstations IPC).

We have developed three load balancing strategies: a "one-by-one" and two versions of "partial distribution" strategies.

In this paper, we start by describing the parallelization of B&B and the implementation of the VCP under PVM. Afterward, we discuss the motivations and justifications to find another notion of priority between different subproblems. Finally we present the advantages of our new notion of priority by showing the improvement in performance of the developed load balancing strategies. As a conclusion, we point out that our approach can be applied to a lot of problems based on tree search space exploration.

2 The PVM machine

PVM [1] is a software system that permits a network of heterogeneous Unix computers (workstations in our case) to be used as a single large parallel computer (named the *virtual machine*). Thus larger computational problems can be solved by using the aggregate power of many computers.

PVM supplies functions to automatically start up tasks on the virtual machine and allows tasks to communicate and synchronize with each other. PVM may start up more than one process on any processor in the PVM virtual machine. This affectation is done by calling the routine *pvm_spawn()*. A task is for us a Unix process.

Applications, such as VCP, can be parallelized by using message-passing constructs common to the most distributed-memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel.

Sending a message is composed of three steps in PVM:

First a send buffer must be initialized by call to *pvm_initsed()* or *pvm_mkbuf()*.

Second the message must be "packed" into this buffer using any number and combination of *pvm_pk*()* routines.

Third the completed message is sent to another process by calling the *pvm_send()* routine or multi-cast with the *pvm_mcast()* routine.

A message is received by calling either a blocking or non-blocking receive routine and then "unpacking" each items from the receive buffer. The receive routines can be set to accept ANY message, or any messages from specified source, or any message with a specified message tag, or only messages with a given message tag from a given source. There is also a probe function that returns whether a message has arrived, but does not actually receive it.

Like that, PVM provides the flexibility to develop parallel applications using its communication facilities. The interested user can refer to [1, 10] for more details about PVM.

3 The Branch & Bound VCP implementation

Branch & Bound [7] is based on exploration of search space to find the optimal solution, but it cuts off parts of this space by using heuristic knowledge about the position of the solution in it. Nevertheless, the size of the search space still grows exponentially with the size of the input (see figure 1). Thus, it is until now an open research problem to study, and the main question which we try to answer with many other people [5, 6, 11, 12], is what can parallelism offer in solving NP-hard combinatorial optimization problems.

The B&B algorithms use a large class of search techniques. Among them, there is the best-first B&B. This technique handles the subproblem with minimal bound. It leads to a very good computational performance but it is very expensive in

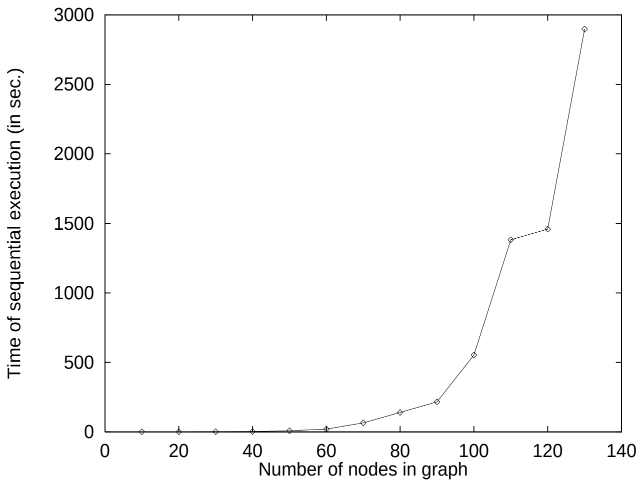

Figure 1: The exponential nature of the VCP for graphs of 50% of average degree.

memory utilization [7, 6, 11, 12].

In this paper, we use a distributed version of best-fist B&B to solve the Vertex Cover Problem (VCP). For an undirected graph $G = (V, E)$ a subset $U \subseteq V$ of nodes is called a vertex cover if every edge is covered by a node from U, i.e. $\{u, v\} \in E$ implies $u \in U$ or $v \in U$. This problem aims to find the vertex cover with minimal cardinality and it is known to be NP-complete [5, 6].

All subproblems are coded as tuples (SC, pc) where SC is a sub-cover (a set of nodes that cover some edges of the graph G) and pc is the number of nodes in SC; we call it from now on, the **partial cost** of the treated subproblem (lower bound on the sub-cover cardinality).

Our B&B algorithm generates two new subproblems (SC_1, pc_1), (SC_2, pc_2) from a given subproblem (SC, pc). The generation of SC_1 and SC_2 by the branch procedure is done by searching for a node v with maximal degree in the rest-graph (part of the original graph that has not yet been covered). $SC_1 = SC \cup \{v\}$, $SC_2 = SC \cup \{w \in E\}$ (for more details, see [5]).

We have implemented the distributed B&B scheme with different strategies of load balancing as we will see later. Our aim in using the B&B algorithm of [5] is not to study the parallelization of this method, but rather to develop an efficient load balancing strategy that uses priority of tasks. Therefore the chosen B&B algorithm is sufficient for our purpose (for more information about B&B algorithms and their parallelizations, refer to [4, 7, 3, 2, 11, 12]).

The chosen B&B algorithm uses a distributed queue organization to temporarily stock the generated subproblems. We chose to execute only one process per processor in the PVM machine. Each processor has a local queue in its local memory and executes the sequential B&B algorithm using this queue.

In our implementation, we have used the best-first technique. Subproblems are classified in many queues in increasing order of their evaluation. The evaluation of a subproblem is given by its partial cost.

With the distributed B&B scheme, the communication plays an essential role in determining the performance especially when we use multi-user machines (like workstations). To improve performance, we must try to reduce the number of exchanged messages during the execution, as we will show in subsequent sections.

In our PVM implementation of the B&B VCP, we used the following kinds of messages to achieve communication between process:

- INIT to initialize the different process started on the different machines in the system,
- END to inform that all work have been done,
- START to indicate that the execution can be started,
- NEW_SOL to indicate that a new solution with a lower bound better than the last known one is found.

In all the tested load balancing strategies, we used, during an operation of load balancing, the following types of messages:

- STOP to stop execution and start a load balancing operation,

- NONE_TO_SOLVE when a process have no work to do, it sends this message asking to start an operation of load balancing,

- SUB_PROB when sending subproblems from process to an other requiring work, we use this message to indicate that the following data contains subproblems to be executed,

- NONE_TO_SEND a process sends this kind of message when it has not sufficient work to participate to a load balancing operation,

- RESTART to restart execution at the end of a load balancing operation.

Execution begins by starting up a process called the *master*. This master in his turn starts up one process at every processor in the used PVM machine (called its *slaves*).

An operation of load balancing is started when one of the *slaves* become inactive (without any work to do) so that it sends a message NONE_TO_SOLVE to the *master*. The *master* in his turn, stops the work of all active *slaves* demanding information about their local loads.

After that *slaves* send information about their local loads to the *master*. The *master* is responsible for determining how many tasks a given *loaded slave* must send to an other one which has not any work to do, as it is indicated in the following two algorithms.

The *master* algorithm:

```
1. Create the root subproblem of the B&B
   tree.
2. Create N slaves (one slave per processor)
   and initialize them by sending INIT
   message.
3. Send the root subproblem to slave ''0''
   and start execution by sending START
   message.
4. While not the end of execution Do
     4.1.  Wait for any message from any
     slave.
     4.2. If message == NONE_TO_SOLVE Then
           - Stop execution by sending
           STOP message.
           - Apply the used load
           balancing strategy.
           - Restart execution by sending
           RESTART message.
         Else
           - Do treatment attached to
           the received message.
     4.3.  Verify the end of the execution.
5. Send END messages to all the slaves.
```

The *slave* algorithm:

```
1. Wait INIT message to do initialization
   before starting execution.
2. If slave number == ''0'' Then
     Wait until receiving the root
     subproblem.
3. Wait START message to start execution
4. end_of_task = FALSE
5. While end_of_task != TRUE Do
     5.1. If local load == 0 and execution
            not yet ended Then
              Send NONE_TO_SOLVE message
              to the master.
          Else
            If exist a STOP message Then
              - Stop execution and do a
              load balancing operation
     5.2. If exist subproblems in local
            queue Then
              Execute the sequential B&B
              algorithm with the first
              subproblem in the queue.
     5.3. If local load == 0 and
            receive END message Then
              end_of_task = TRUE.
```

4 Motivations and justifications of our approach

A sequential execution of the B&B algorithm with graphs of 50% of average degree (figure 2), shows that the difference between the number of handled subproblems before finding the first solution and the total number of all handled subproblems is relatively big. Thus these subproblems could

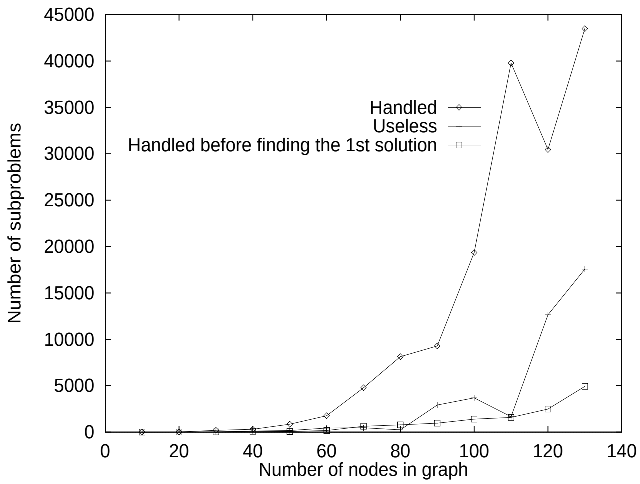

Figure 2: The behavior of the B&B sequential algorithm.

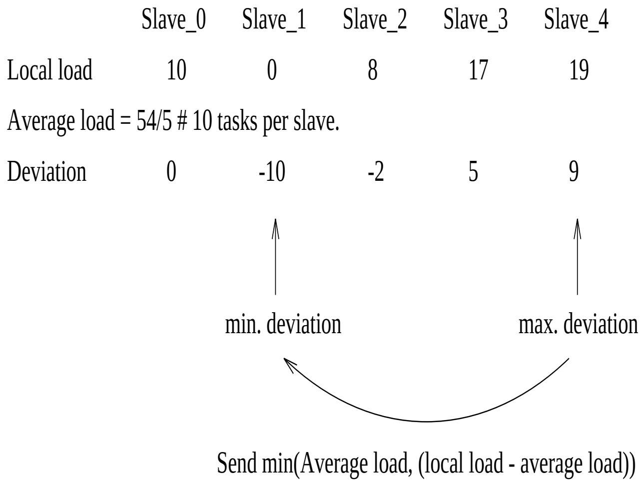

Figure 3: An example of application of partial distribution load balancing strategy, where slave_1 asks the other slaves for work.

be distribute when we have more than one processor.

This B&B VCP implementation provides us a good situation to study the different load balancing strategies.

We have developed three load balancing strategies, a "one_by_one" and two versions of "partial distribution", to understand load distribution between the different slaves. We started by testing strategies with the generated subproblems of each processor classified into a local queue according to their associated **partial cost**.

In the one_by_one load balancing strategy, each slave needing work receives only one subproblem from the beginning of the queue of an other loaded processor.

In the partial distribution strategies, the number of tasks received by a processor requiring work is variable. The number of subproblems sent from the maximum loaded slave to the one requiring work is equal to the minimum of the average load and the local load minus the average load (see the following example given by figure 3).

The first version of partial distribution load balancing strategy performs load balancing by sending subproblems from the beginning of the local queue whereas the second one sends subproblems from the end of the queue.

Figures 4, 5 and 6 show the results of the execution of a 75 nodes graph VCP problem on a PVM machine of three workstations with a "one_by_one", "partial distribution" (Version 1) and "partial distribution" (Version 2) load balancing strategy, respectively.

Notice that we get a very bad distribution of load in the case of the "one_by_one" strategy; at the beginning all slaves have the same quantity of work but after that one of them (the second slave) becomes strongly loaded although this slave is not the one where the root subproblem in the B&B tree has been affected.

We can explain this situation by the fact that some subproblems potentially generate more work than the others in the search space.

We notice also that with version 2 (figure 6) of the partial distribution load balancing strategy, the loads of different slaves are globally more equally distributed comparing to the version 1 (figure 5), and that with version 2 load balancing operations occur less frequently than with version 1. This proves that the **partial cost** used as a criterion to classify subproblems within the

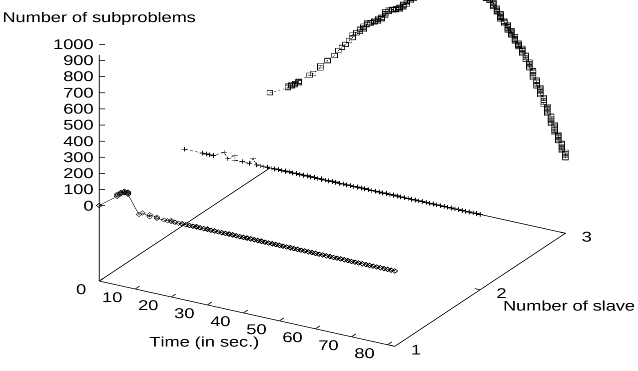

Figure 4: The load distribution by "on_by_one" load balancing strategy.

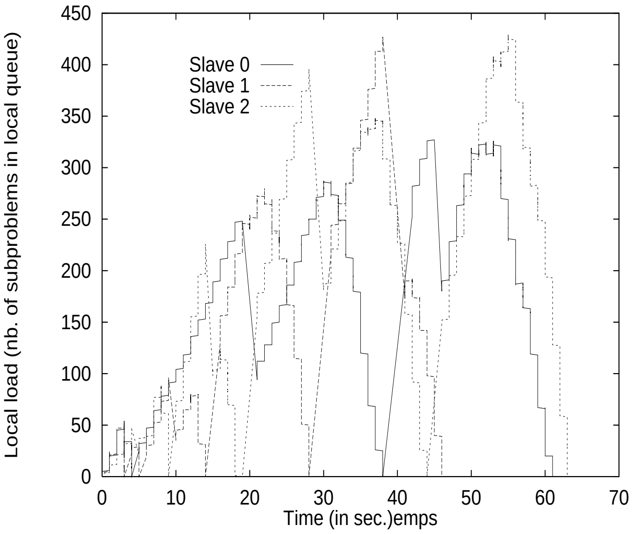

Figure 5: The load distribution by "partial distribution" (Version 1) load balancing strategy.

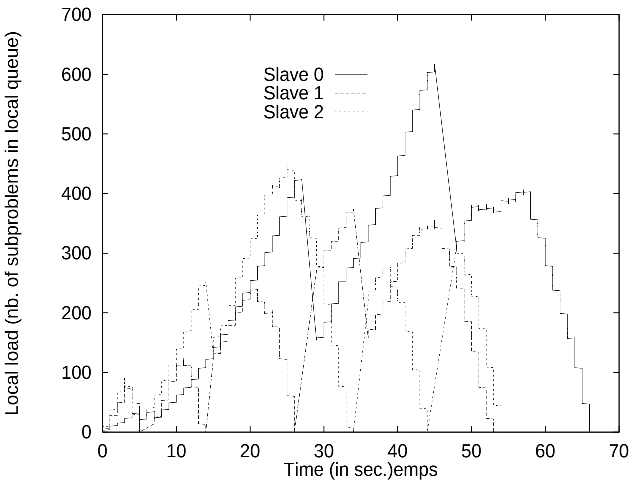

Figure 6: The load distribution by "partial distribution" (Version 2) load balancing strategy.

queues is not the only parameter to apply to get a good load balancing strategy.

The above results indicate that it is important to take into consideration the potential work that subproblems can generate.

Any efficient load balancing strategy should ensure, as much as possible, that the processing of subproblems occurs in the global order of their partial cost and should take into consideration their associated potential work. Consequently, we need to define a priority notion between different subproblems which takes into consideration these two parameters.

5 A new notion of Priority: Potentiality

A subproblem generated during the execution of a B&B algorithm has two associated values: its cost and an indicator representing the potential work it can generate successively to reach an optimal solution (see figure 7). These two values depend on the treated problem. In the VCP problem for example, the first one is equal to the partial cost, and the other value will be taken equal to the number of edges in the rest-graph.

Each generated subproblem is characterized by its position into the indicated region in figure 7. The execution of B&B algorithm starts from **Start** point (where we have the root of B&B tree) and terminates at **End** point (the searched optimal solution).

The potentiality of a subproblem sp is defined as:

Definition .1

$$Potentiality(sp) = (dist(sp, End), \\ potential\_work, \\ ancestors\_nb).$$

Where:

sp is the considered subproblem,

$dist(sp, End)$, the distance between the position of sp and the **End** point,

$potential\_work$, the value of the associated potential work of sp,

$ancestors\_nb$, the number of the ancestors of sp.

In this definition of priority, we introduced a third parameter, the number of ancestors, to select between subproblems that have the same distance from the End point and the same quantity of potential work. We chose to privilege those having the less number of ancestors.

In practice, the value of the optimal solution is not known before the end of the execution, but usually we start execution with an upper bound that maximizes this value.

In the VCP problem, for example, we can use the number of nodes of a given graph as an End point and we compute distances relatively to it.

Generated subproblems are classified into the different queues during the execution of B&B distributed algorithm as a function of their priorities. A subproblem sp_1 has more priority than another subproblem sp_2 if it verifies one of the following rules:

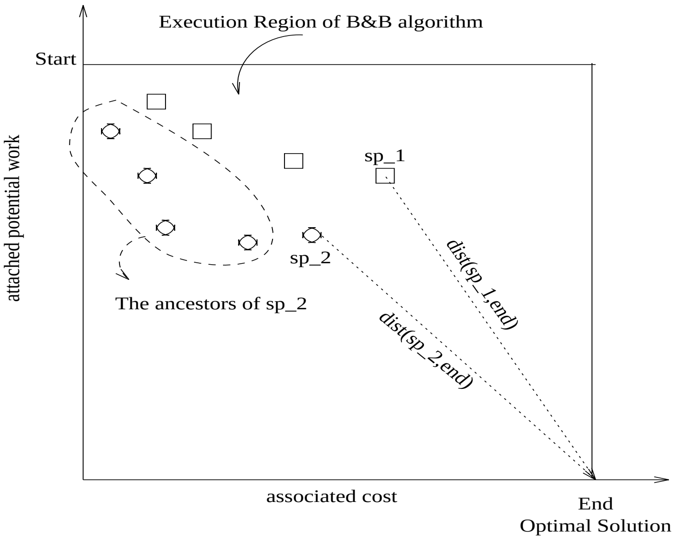

Figure 7: Definition of priority of two nodes sp_1 and sp_2.

$$dist(sp_1, End) < dist(sp_2, End) \tag{1}$$

$$(dist(sp_1, End) == dist(sp_2, End))\&\& \\ (sp_1.potential\_work > sp_2.potential\_work) \tag{2}$$

$$(dist(sp_1, End) == dist(sp_2, End))\&\& \\ (sp_1.potential\_work == sp_2.potential\_work)\&\& \\ (sp_1.ancestors\_nb < sp_2.ancestors\_nb) \tag{3}$$

6 Behavior of tested load balancing strategies with the new notion of priority

This notion has lead to improvements such as a better load distribution among the different processors in the PVM machine, and a reduction of the number of exchanged messages during the execution.

6.1 Influence on load distribution

Figures 8, 9 and 10 show load distribution for an execution of VCP with graph of 75 nodes using

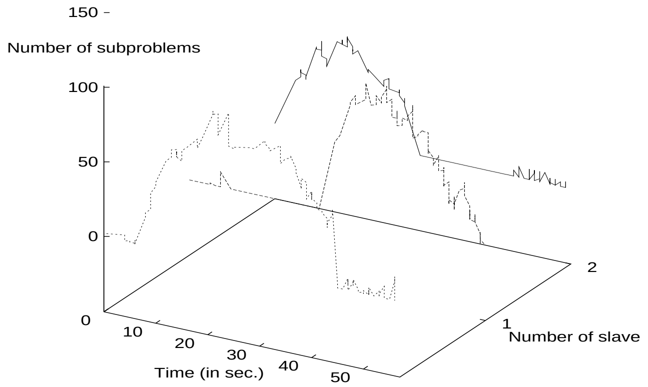

Figure 8: The load distribution by "one_by_one" strategy using the notion of potentiality.

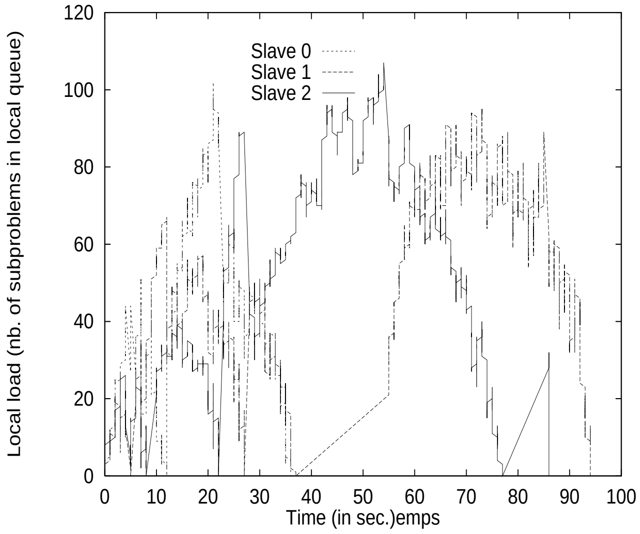

Figure 9: The load distribution by "partial distribution" (version 1) strategy using the notion of potentiality.

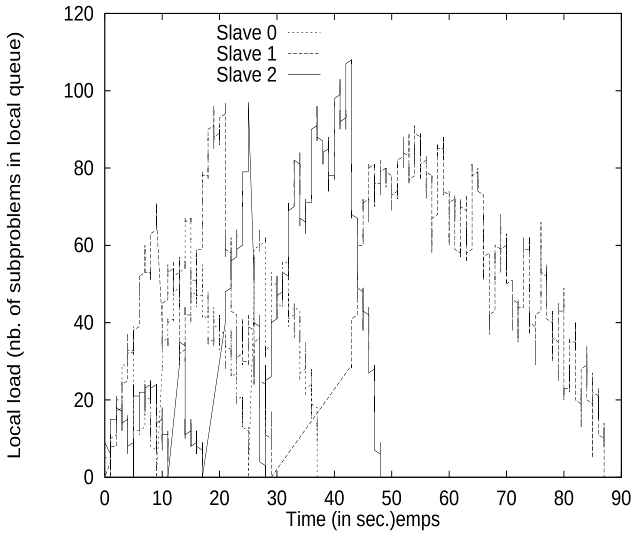

Figure 10: The load distribution by "partial distribution" (version 2) strategy using the notion of potentiality.

our notion of priority between subproblems with the three load balancing strategies developed in the previous section.

For the "one_by_one" load balancing strategy, we remark that at the beginning of the execution (figure 8), the assigned subproblems have more priority than the others so that the number of load balancing operations is lower than its number without using our priority notion (figure 4). At the end, we have a lot of load balancing operations when the priority of subproblems begins to decrease.

The load is equally distributed with partial distribution load balancing strategies (version 1 and 2, figures 9 and 10).

We can see also that the load of the most loaded slave does not exceed the 110 subproblems while it was around 450 subproblems (with version 1) and 600 subproblems (with version 2), without using our potentiality notion (figures 5 and 6).

6.2 Influence on communication costs

To show the importance of the communication factor, we have chosen to study the exchanged

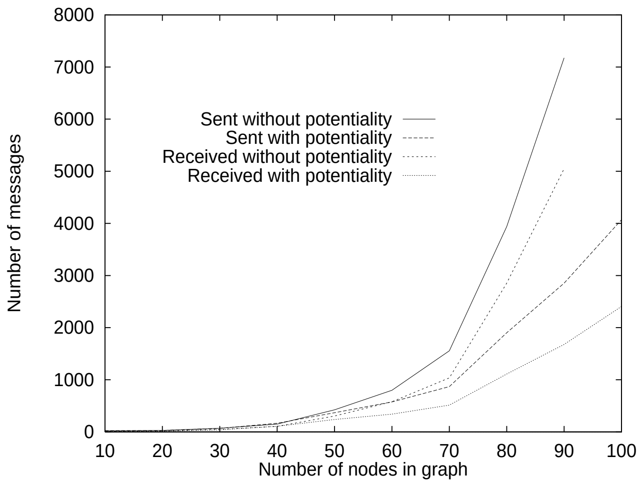

Figure 11: The sent and received messages using "one_by_one" strategy with and without the notion of potentiality.

messages between the master and all of its slaves.

In figure 11, we remark that there is a clear difference in the number of sent and received messages with and without using the potentiality notion. This number is lesser in the first case.

For the two versions of partial distribution load balancing strategy (figures 12 and 13), we remark that the number of sent and received messages takes an asymptotic value. In version 1, the number of exchanged messages remains approximately the same, while in version 2 the utilization of our priority notion lightly increases the number of exchanged messages. In fact, with this version, when an unloaded slave takes subproblems from the end of the queue of a loaded slave, this unloaded slave executes the work and rapidly becomes unloaded. This light augmentation of the number of load balancing operations shows that subproblems are really classified in queues as a function of their priority of execution (see figure 10).

7 Conclusions & perspectives

We analyzed the behavior of three load balancing strategies and proved the necessity to have a new priority order between the different executed

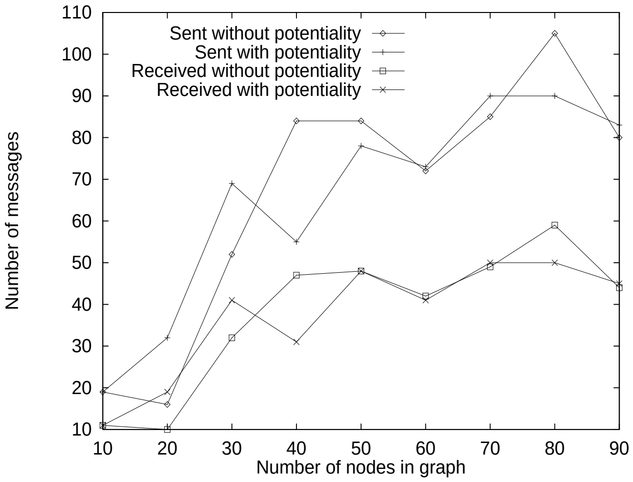

Figure 12: The sent and received messages using "partial distribution" (version 1) strategy with and without the notion of potentiality.

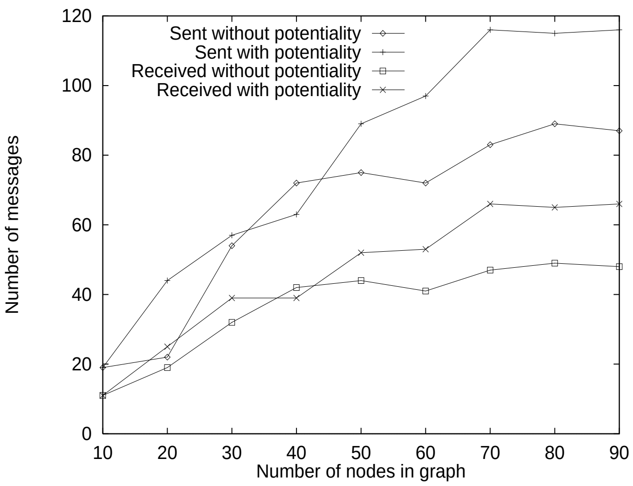

Figure 13: The sent and received messages using "partial distribution" (version 2) strategy with and without the notion of potentiality.

subproblems in a B&B application.

We introduced the notion of potentiality based on an estimation of the capacity of a subproblem to generate work, and on an evaluation of the distance between the subproblem and the optimal solution. We believe that this kind of priority can be applied to any B&B application where subproblems are characterized by the previous parameters.

Furthermore, we studied the impact of this notion on the behavior of load balancing strategies for distributed best-first B&B applications. These strategies show that after taking into consideration the potentiality of subproblems, their selection to participate to an operation of load balancing is not the same at the beginning and at the end of the execution.

All these load balancing strategies are developed in a centralized manner (there is one master processor that controls the other ones in the PVM machine). Future works will consist of investigating the effect of the use of our potentiality notion when several masters instead of only one (like the strategies in [9, 8]) are used.

References

[1] Al Geist, Adam Beguelin, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge, Tennessee 37831, May 1993.

[2] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. Technical Report 913, Centre de recherche sur les transports, Montréal (Canada), May 1993.

[3] Ten-Hwang Lai and Rartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *Communication of the ACM*, 27(6):594–602, June 1984.

[4] E. L. Lawler and D. E. Wood. Branch and bound methods: A survey. *Operations Research*, (14):670–719, 1966.

[5] R. Lüling and B. Monien. Two strategies for solving the vertex cover problem on a transputer network. In *proc. of the 3rd Int. Workshop on Distributed Algorithms*, pages 160–171. number 392 in Lecture Notes of Computer Sciences, 1989.

[6] R. Lüling and B. Monien. Load balancing for distributed branch and bound algorithms. In *Proc. of Int. Parallel Processing Symposium*, pages 543–549. IPPS, 1992.

[7] C. Roucairol. Parallel computing and combinatorial optimization. In *EURO XIII, Operational Research Designing Practical Solutions*, July 1994.

[8] Vikram A. Saletore and Mannan A. Mohammed. Hierarchical load balancing schemes for branch-and-bound computations on distributed memory machines. In *Hawaii International Conference on System Software*, January 1993.

[9] A. B. Sinha and L. V. Kale. A load balancing strategy for prioritized execution of tasks. In *Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems (in co-operation with ECOOP '92)*, June 1992.

[10] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–546, March 1994.

[11] Benjamin W. Wah and Chee Fen Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE Transactions on software engineering*, SE-11(9):922–934, September 1985.

[12] Myung K. Yang and Chita R. Das. Evaluation of a parallel branch-and-bound algorithm on a class of multiprocessors. *IEEE Transactions on parallel and distributed Systems*, 5(1):74–86, January 1994.