A Generic Parallel Branch and Bound Environment on a Network of Workstations

Roman Batoukov*
Dept. of Informatics,
University of Bergen,
NORWAY

Tor Sørevik†
Dept. of Informatics,
University of Bergen,
NORWAY

June 7, 1999

By utilizing the idle cycles on networked workstations (NOWs), we can get supercomputing power essential free of cost. But in spite of this obvious economical advantages only a few heroic examples on successful computation on NOWs exists. For parallel program to run efficiently on NOWs they require a coarse grain parallelism and dynamical load balancing.

In this paper we present a generic framework for parallel branch and bound algorithms which has these feature. We discuss and motivate the choice we have made, and present test results from an object oriented implementation.

1 Introduction

Using network of workstations (NOWs) for large scale computation is an appealing idea. The combined power of hundreds or thousands of workstations equals the computational power of a top-of-the-line supercomputer. Moreover, utilizing the idle cycles on networked workstations is an essential free of cost supercomputer, compared to the multi million dollar price of a high end HPC-system.

Nevertheless only a few heroic examples on successful computation on NOWs exists. The most noticeable being, the factorization of RSA 129 [AGLL94] and the record breaking Traveling salesman computation by Bixby et. al. [ABCC98].

There are numerous reasons why NOWs are not extensively used for large scale computations. First of all NOWs typically has lower bandwidth and higher latency than the internal network in todays parallel computers, thus communication intensive parallel applications don't scale well on NOWs. Secondly, one can't assume that each computational node in a NOW has equal power. This might be due to differences in hardware as well a computational load imposed by other usage. Consequently one need a dynamical load balancing scheme which can adapt to available resources of the NOW and the demand of the application. Most existing parallel program do apply static load balancing, and can for that reason not be expected to run well on NOWs.

These two reasons alone exclude most of the problems traditionally solved on parallel computers. There is however still a important subclasses of large scale problems which posses a course grain parallelism and lends itself naturally to a dynamic scheduling approach. It is our claim that a number of combinatorial optimization problems fall into this class.

When few examples of successful computation of combinatorial problems on NOWs are known, we believe the reason to be the programming and administration effort needed. NOWs don't provide a single system image, making it cumbersome to overview the available resource. The programming environment is also much less developed than on a single-image supercomputer. Making programming an even harder

*http://www.ii.uib.no/~roman
†http://www.ii.uib.no/~tors

task than parallel programming already is. Other problems more pronounced on NOWs then elsewhere are reliability and fault tolerance.

In this paper we address some, but not all of the problems listed above. Our target target application field is combinatorial optimization problems. More precisely, our aim is to design a low communication cost, branch and bound algorithm with a corase grain parallelism and an efficient dynamic load balancing scheme. Using object oriented programming techniques we'll try to easen the programming burden by creating a generic framework for a class of problems.

The paper is organized as follows. In section 2 we discuss various aspects of branch and bound algorithms for minimization problem, and point out some important decisions which will effect the overall efficiency of a parallel algorithm for NOWs. Before we in section 3 describe our algorithm. In section 4 we briefly describe our generic programming framework. While we in section 5 provide some numerical experiments, using the knapsack problem as our test case. Finally we wrap up with some concluding remarks in section 6.

2 Some issues in Parallel Branch and Bound algorithms

In this section we assume our optimization problem is formulated as a minmization problem.

A B&B algorithm sets out to traverse the entire search space, by recursively refining the search space into disjoint subspaces. Each subspace is checked for feasible solutions. If none exists the subspace is discarded from the set of potential interesting subspaces. If only one explicit solution exist, its value is computed and if possible, the *current-best solution* is updated. If more than one feasible solution exist a lower bound is computed. If the lower bound is larger than the *current-best solution*, the subspace can be discarded from further investigation. If not the subspace is recursively divided into new subspaces. The new subspaces are kept in a data structure keeping all candidate subspaces.

A key to efficient B&B algorithms is to get good bounding values at an early stage. One don't necessarily need an explicit feasible solution for this. If an upper bound for the objective function exists for a subspace, we can equally well use this for updating the *current-best solution*.

The distinguishing feature of B&B algorithms are

- A branching criterium for division of the search space into subspaces.
- Bounding functions.
- A selection criterium for which subspace to compute next.

Good choices for the first two items are problem dependent and will therefor not be discussed here. We only remind the reader that the very point of computing a bound is to compare it with the *current-best solution*. Thus to serve it's purpose, every compute node(PE) have to have available a *current-best solution*. How to globally update the *current-best solution* in a distributed system without hurting the performance with frequent synchronization and communication, becomes an important issue in distributed parallel computation.

The choice made for the selection criterium implicitly impose a data structure on our algorithm. The data structure has to be maintained as the computation progress. The distribution and updating of the data structure turns out to be at the very heart of any parallelization strategy and is closely linked to the selection criterium.

2.1 Subspace selection

A B&B procedure might equally well be described as a tree traversal. The tree is not explicitly created in advance, but are produced by the branching procedure. At any given point each node represent a subtree. Any of these containing feasible solutions within the current bounds, are subject for further investigation, which is done by recursively splitting into subtrees ("Node expansion"). The selection criterium for which node to evaluate next produces the order of the tree traversal. If a feasible solution exists, we eventually find it as a leave node.

Any standard tree traversal like Depth First Search(DFS) Breadth First Search(BFS) and Best First Search(BeFS) are in principal applicable. However, the need for good bounding values suggest the omission of BFS, as this strategy does not reach an explicit solution before most of the tree is traversed. The philosophies behind BeFS is to always search for the best possible solution. If succeeding in this, one soon get a good bounding value and thereby shortening the search. BeFS does however require a more space and time consuming data structure, usually a heap as opposed to DFS which can be implemented with a simple stack.

The evaluation of specific nodes are independent task and thus the process has lots of opportunity for parallelism. Updating the data structure, it being a stack or a heap, is however an inherently sequential operations, and as such a potential sequential bottleneck. Moreover, at a distributed system a centralized updating of one single data structure easily gives us a communication and memory bottleneck. In the next section we propose a new distributed data structure to circumvent these problems.

2.2 Characteristics of NOWs

Comparing NOWs with Single-System-Image parallel computers, we find two thing of importance for algorithm design.

1. The power of the different computational nodes(PE) might be different, and even worse, their computational power might varying during the computation.
2. The communication efficiency is low relative to the computation speed compared with a supercomputer.

The first of these issues is a strong argument for using a dynamic load balancing scheme. The second is a warning against excessive communication. Note that there is a potential conflict here. To make sound decisions on the fly we want all information to be available to the decision maker(s). But since changes in status appears locally as a byproduct of the computation, it needs to be communicated to the decision maker(s).

3 Our Algorithm

3.1 Parallel Data structure and selection strategy

In the previous section we argued that BeFS usually gives fastest convergence. The reason is that by selecting the most promising node we increase our chances to get a good bounding value at an early stage. This advantageous feature we want to keep, but to avoid the sequential bottlenecks inherently involved in choosing the global optimum, we have to construct a distributed data structure. The natural thing to do is instead of insisting on selecting the "best" node, we are willing to work with a "good" one. So instead of ranking all the nodes, we simply sort them into a few classes, say *good, normal* and *bad*, and as long as there is any *good* nodes available, we choose our next node from the set of *good* nodes. If none *good* are available a *normal* is chosen, and only if no other non-empty sets exists, we process a *bad* node.

Besides maintaining the ability to choose nodes for further expansion based on their quality, this simple data structure lends itself naturally to distributed computing. Each PE can keep its own sets, and as long as good nodes are available locally there is no need for communication.

Within this framework we apply DFS at a local level. That means: If the processing of a node results in a subdivision creating one or more *good* nodes, we'll choose one of these as our next candidate.

This selection strategy makes it possible to postpone bound calculation until processing of the subtree (This is not the case for BeFS!). This is of importance in a distributed search, as we increase the chances of elimination of subtrees by postponing the bounding evaluation as long as possible, in hope of receiving a new and improved *current-best* solution. This strategy is often referred to as "The lazy strategy" [Cla97]

3.2 Load balancing and communication

With this distributed data structure each node can work on it's own local search as long as it's local set of *good* nodes is non-empty. When that's no longer the case it have the choice of either continuing with nodes from the *normal* set or looking for *good* nodes somewhere else. Both choice have their drawbacks the first one are forcing us to work on non-optimal candidates, while the second require communication.

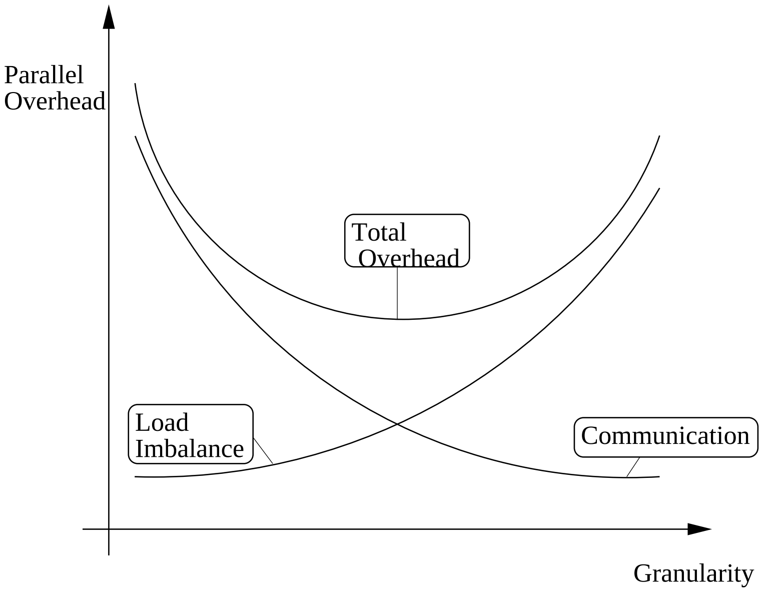

Figure 1: A schematic illustration of how granularity effect two different kind of parallel overhead

As indicated in Figure 1 the relation between the amount of work to be ship and the frequency of shipping it, (usually called the granularity of a parallel problem) and the parallel overhead (extra cost due to parallel processing), works in opposite directions. If we frequently send and process small amount of work, the calculations are fine-grained. Fine-grained calculation pattern normally leads to better load balance, but causes major communication overhead, since it requires very extensive interprocessor communications. The coarse-grained calculation pattern avoids frequent interprocessor data exchange, but will more likely lead to greater load imbalance. In our implementation we have made the extreme choice, as no PE ask for work before all its sets of pending nodes are empty.

Having high communication cost, as is the case on NOWs, we must assure that enough work are transferred in each step. For that reason we introducing a cut-off level for the depth of the subtree. Subtrees beyond this level will not be transferred. This will prevent too small packets to be sent.

Nevertheless, there is always a risk that the shipped subtrees will get pruned shortly after shipment. That could lead to high load imbalance. We try to avoid this by transferring only the most promising nodes, i.e. the nodes with the high probability to contain the optimal solution (nodes of the Good class, or Norm class, if there are no available subproblems of Good class).

Distributed dynamic load balancing methods can be classified according to the acting component of the load transfer (sending process or receiving process), in another words, according to initiating process. There are two types of load balancing algorithms: sender initiated and receiver initiated load balancing [Cla97].

Within our framework, receiver initiated load balancing algorithm seems most suitable. For the PE, having enough local work, it's not reasonable spending time on distributing its status among other PEs and looking for the potential receivers, whereas for a processor, which is idle or weakly loaded, it's a good idea to spend time looking for work. In the receiver initiated work distribution, also known as task attraction, processors having low load compared to the others, send work request to some processors in the network. When the recipient has work to share, it responds with a work packet. Otherwise the requester get notified that the recipient run out of work.

We have implemented two different receiver initiated work distribution schemes:

RR: In the round robin (RR) strategy, idle processors look up a target variable pointing to a potential donor processor. Initially, target is set to the next neighbor processor (p+1) mod P, where p is the rank of the processor, P - the number of the processors involved in computations. After each lookup, target is incremented modulo p, to ensure that all processors are asked with the same probability.

RP: In random polling (RP) , idle processors send work requests to a randomly selected processor. Each donor is chosen with the same probability [P.S94].

3.3 Termination detection

In a distributed algorithm, without a central point of global control, it becomes a non-trivial issue to determine when the overall computation is done. We have implemented and tested 2 different termination detection strategies:

The first method (tree-based termination detection [VAAG94]) is based on the introduction of a value, called energy. Initially, energy = 1 for the PE with rank 0 (root PE), and 0 for all other PEs, which initially are idle. When a subproblem is partitioned and a part is sent to some PE, the energy value is halved, and one half is sent together with the subproblem. When a PE is done with its work, it returns the value of energy to the root PE. The termination is detected when the root PE is idle and its value of energy =1. To avoid approximation errors, the value of energy should be represented as a rational fraction. Reduction operation should be applied regularly to this fraction, to prevent overflow. But experiments indicate that it's not always possible to avoid it. Therefore overflow control have to be applied, preventing local work from being partitioned when it could lead to overflow of the energy value.

Another scheme is called the four counter method [P.S96]. Each PE counts its number of sent and received messages of subproblems. When a PE has become idle it contributes its counters to a global sumation. When the global sum over these two counters yields identical results for two consequitive global additions, there cannot be any work left (not even in transition). For simplicity and in order to minimize message complexity, a ring based addition protocol is favored. It's possible to use an almost equally simple and portable asynchronous global reduction operation. A PE only needs to put its counters into a new global sum computation if a new round has been announced and it has become idle. This guarantees that the computation expense for termination detection cannot be higher than the expense for sending requests. Thanks to our modular system design, one termination detection algorithm could easily be replaced by another. We'll consider the modification of the four counter method. For the purpose of generality, we define energy for each PE as the pair (number of received subproblems, number of subproblems sent to other PEs) in case of the four counter method. For the later scheme, as soon as a processor becomes Idle, it resets local message counters and returns energy to dedicated processor (root PE), which accumulates it. As soon as this dedicated processor becomes idle and accumulated number of sent messages equals to the number of received messages, it announces new round of global energy sum computation (global reduction operation).

4 Implementation

It has been our aim to develop a generic programming environment to build parallel distributed B&B programs for the networks of workstations. The idea is to facilitate writing B&B programs for any application and to make it possible to test several parallelization parameters (load balancing strategies, termination detection methods, initialization schemes etc.).

A generalized environment should enable a user to concentrate only on the implementation of the problem dependent parts, without particular attention to the parallelization. To have a general framework encompassing all combinatorial optimization problems, it's necessary to separate the problem dependent parts, problem specification and bounding functions, from the rest of the implementation.

To have modular system where fundamental features like data structure easily can be changed and inherited by different modules object-oriented programming was called for. Our implementation of the system was carried out in the C++ language. For communication we have used MPI [MSHL+96]. An overview of the system component is given in Figure 2.

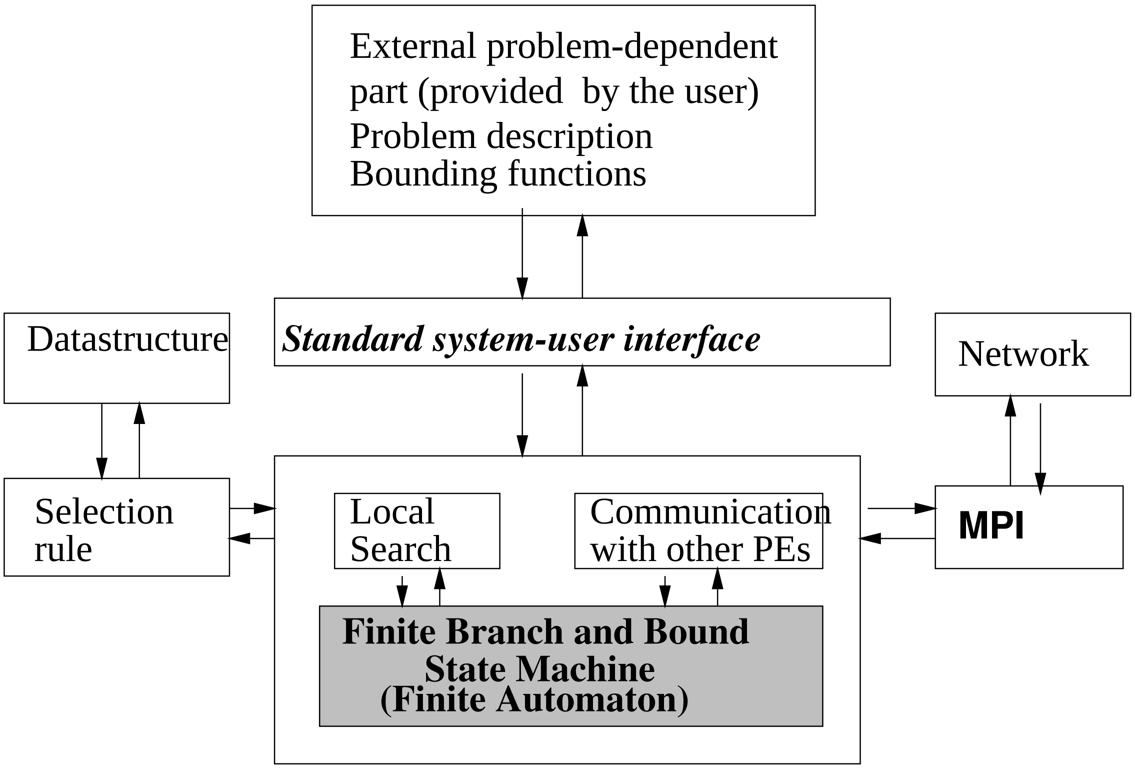

Figure 2: System overview

The framework defines an Finite Automata as it posses:

- A finite number of states
- A well defined input alphabet (incoming messages or internal computational results)
- A set of rules for changing of states
- A finite number of action accompanying each state.

This has been used and our algorithm is implemented as a Finite Automaton.

5 Numerical experiments

As our test case we have chosen the 0-1 Knapsack problem. This can formally be stated as:

$$\max_{\forall x} \quad \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$\sum_{i=1}^{n} w_i x_i \quad \leq M \tag{2}$$

$$x_i \in \{0,1\} \quad \forall i \tag{3}$$

n	Parallel execution time		
	average	Min	Max
300	221.3	216.9	227.8
500	120.1	95.6	135.0
700	45.5	42.7	54.2
900	281.1	258.9	290.5

Table 1: Timing for 5 independent runs of 4 different problems

where $v_i, w_i, i = 1, ..., n$ and M are problem dependent parameters while $x_i, i = 1, ..., n$ are the free variables. An upper bound on this problem is easily found when relaxing the integer constrain. The "Value/weight ratio" (v_i/w_i) can be used to sort the variables in decreasing order of importance. A lower bound can now be computed by taking the first k elements of the sorted list such that

$$LB = \sum_{i=1}^{k} w_{p(i)} x_{p(i)} \leq M < LB + w_{p(k+1)} x_{p(k+1)} \tag{4}$$

In a "best-first-spirit", the sorted list is also used when selecting variable to branch on.

In [P.S96] P. Sanders shows that if v_i and w_i are chosen from a uniform random distribution the heuristics are so efficient that the number of node expansion is close to the theoretical minimum, n. He therefore suggest to correlate the choice of v_i and w_i by first selecting w_i at random and then let $v_i \in [w_i + v_{min}, w_i + v_{max}]$. The gap between v_{max} and v_{min} can be used to tune the difficulty of the problem. In our tests we have chosen $w_i \in [0.01, 1.01]$, $v_i \in [w_i + 0.1, w_i + 0.125]$ and $M = \frac{1}{2} \sum_{i=1}^{n} w_i$.

Our testbed was networked 300 Mhz, Sun UltraSPARC 10 connected by 100 Mbits Ethernet. The different workstations was not doing other significant work during our experiments, thus the assumption of equal power for each node is satisfied. The network, however, was a part of a larger Ethernet segment, and we had no control on whether or not other traffic took place during our tests.

Due to the 'noise' in the network, it is impossible to predict which workstation is first done with its task and will ask for more work to do. The dynamic feature of the algorithm introduce nondeterminism in how the search tree is traversed, implying that different trees might be created for different runs of identical problems. This nondeterminism actually do account for unpredictable differences in execution time. This is illustrated in Table 1 where we report minimum, maximum and average running time for 5 runs of 4 different problems.

Note also that there might be a huge difference in computational difficult due to random effects when generating the data. The table also indicate that the computation time is linear in times of n as well as number of nodes generated.

We tested two of the load balancing techniques discussed in section 2. The Asynchronous Round Robin(ARR) and the Random Polling(RP). For the same problem as above RP proved to be consistently best in terms of generating the least number of nodes as well as measured by running time.

We have also implemented two different termination detection scheme. Tree-based termination detection and the four counter method. The later being far better than the first. The reason is that tree-based termination detection easily can provoke numerical overflow. To avoid this an overflow check has to be inserted. If potential overflow is encountered, the processor in question simply refused to partition the work, but insists doing it itself. Consequently we frequently observed a huge load imbalance when tree-base determination detection was used.

To quantify the different obstacles to optimal speedup we have defined 2 measures:

Parallel extra cost The total number of nodes generated by the sequential algorithm, N_{seq}, is usually less than the total number of nodes, N_{par}, generated in the parallel case. The difference can be regarded as the parallel extra cost. Assuming the work of the N_{par} nodes are equally distributed

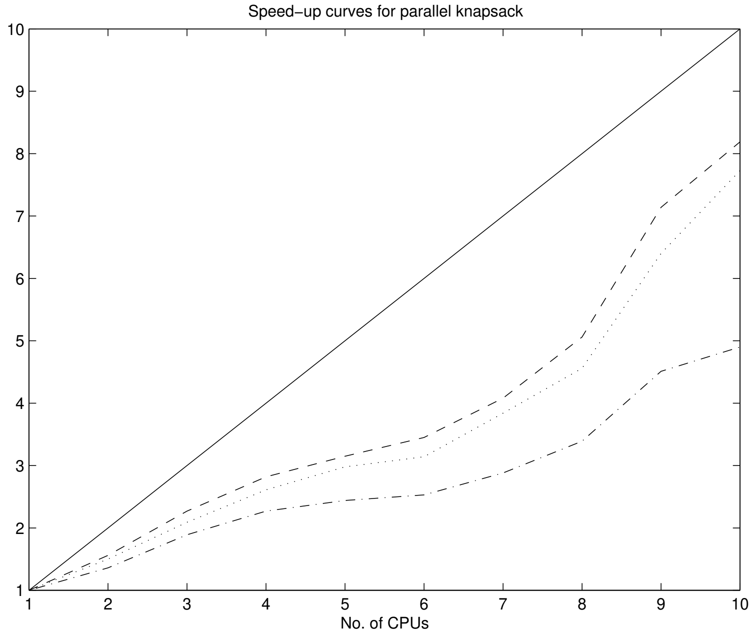

Figure 3: **Experiments on NOWs** The dashed line is upper bound for Speed up corrected for extra work. The dotted line is upper bound for Speed up corrected for extra work and load imbalance, while the dash-dotted line is the actual speed up achieved.

to the P processors we get a bound on the speed up

$$\bar{S_p} = \frac{P \cdot N_{seq}}{N_par} \tag{5}$$

The Load imbalance measure is the maximal positive deviation from the average number of nodes computed on any single processor. It is formally defined as:

$$\Delta L = \frac{\max_{i=1,\ldots,P}(np_i) - \frac{N_{par}}{P}}{N_{par}} \tag{6}$$

where np_i is the number of nodes computed on PE i while $N_{par} = \sum_{i=1}^{P} np_i$.

Putting these two together we can compute an upper bound for the speed-up, assuming that the computational work is proportional to the number of nodes computed, and that each PE has the same computational power. The bound is

$$\bar{\bar{S_p}} = \bar{S_p}\frac{1}{p\Delta L + 1} = \frac{P \cdot N_{seq}}{N_{par}}\frac{1}{P\Delta L + 1} \tag{7}$$

The first quotient representing an upper bound due to increased total work, the second an additional slow down due to load imbalance.

In Figure 3 we display the obtained speed-up on NOWs compare it with the bound due to parallel overhead (6) and the theoretical speed-up bound given in (7). As before we have repeated the test 5 times and recorded the averages. From the figure it's clear that our implementation has substantial parallel overhead. The reason for this is probably the very cautious work-attraction strategy we have implemented, where a PE may be working on *bad* nodes even if *good* nodes are available elsewhere.

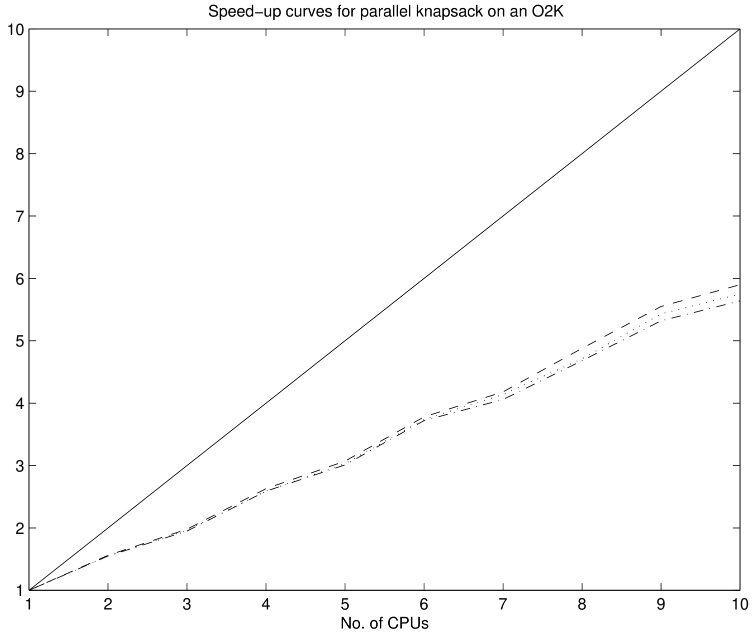

Figure 4: **Experiments on Origin 2000** The dashed line is upper bound for Speed up corrected for extra work. The dotted line is upper bound for Speed up corrected for extra work and load imbalance, while the dash-dotted line is the actual speed up achieved.

The Load imbalance is however low, while there is a still a substantial gap between the Theoretical speed-up bound as computed by (7). A parallel cost not included in (7) is the communication cost. To confirm whether or not communication cost can explain the discrepancy between measured speed-up and the bound given in (7)we run the same experiment on an origin 2000. The results are displayed in Figure 4. This maneuver almost closed the gap between the theoretical bound and the measured speed-up. Thus we feel confident that on NOWs the gap is due to slow communication.

6 Conclusions and future work

The key element of our parallel Branch and bound algorithm is the data structure which facilitate a parallel implementation and dynamic load balancing. The algorithm contains a number of user tunable parameters which allows the user to experiment with the level of granularity. By targeting a Network of Workstation we have deliberately chosen a coarse granularity. Even so we find the load balancing to be pretty good while we had to pay substantially for the communication. The main reason for this is that we have used the knapsack problem as our test case. The evaluation of a node for this problem is very lean in arithmetic. Which makes the computation/communication ratio much worse than almost any other problem. Testing our scheme on more compute intensive problem is therefor first on our agenda.

An other important feature is the object oriented programming, which facilitate easy adaption to other combinatorial optimization problems. To prove that this claim hold in practice we will try our system on more sophisticated problem.

References

[ABCC98] D. Applegate, R. E. Bixby, V. Chvatal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, ICM(III):645–656, 1998.

[AGLL94] Derek Atkins, Michael Graff, Arjen K. Lenstra, and Paul C. Leyland. The magic words are squeamish ossifrage. In Josef Pieprzyk and Reihanah Safavi-Naini, editors, *Advances in Cryptology – ASIACRYPT '94*. Springer, 1994.

[Cla97] J. Clausen. *Parallel branch and bound - principles and personal experiments*, pages 239–267. Klewer academic publishers, 1997.

[MSHL+96] M.Shir, S.Otto, S. Huss-Lederman, D. Walker, and D. Dongarra. *MPI The complete reference*. MIT press, 1996.

[P.S94] P.Sanders. A detail analysis of random polling load balancing. In *International Symposium on Parallel Architectures, Algorithms and Networks*, pages 382–389, Kanazava, Japan, 1994.

[P.S96] P.Sanders. A scalable parallel tree search library. In *2nd Workshop on Solving Irregular Problems on Distributed Memory Machines*, Honolulu, Hawaii, 1996.

[VAAG94] V.Kumar, A.Gramma, A.Gupta, and G.Karypis. *Introduction to parallel computing. Design and analysis of algorithms.* Benjamin Cummings, 1994.