Increasing NP-Complete Branch & Bound Parallel Algorithms' Performance in MPI 1.2 with Randomized Work Stealing

Stéfano Mór, Nicolas Maillard
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Grupo de Processamento Paralelo e Distribuído
Av. Bento Goncalves, 9500
{sdkmor, nicolas}@inf.ufrgs.br

Abstract

This paper studies the application of the Randomized Work Stealing Algorithm into the parallel message-passing work balancing of a branch & bound NP-Complete solver algorithm. Randomized Work Stealing is a classical and efficient algorithm for multithreaded computations schedulers' management.

Brief theoretical analysis and practical results are taken. For this, we rely on MPI-1.2 and an implementation of the Parallel Tree Search method for the parallelization of algorithms (based on backtrack or branch & bound) applied for the well-known Knapsack problem.

Our main contribution is the proposal of a computing model that preserves some of the main Randomized Work Stealing known properties at a strict MIMD parallel load balancing context rather than the original multithreaded scheduler one, thus improving the former's performance. At the end, we show that nearly 80 percent of performance can be gained for this specific problem without loss of linear memory usage and almost linear speedup, even at an homogeneous processor environment.

1. Introduction

Since 1996, MPI (*Message-Passing Interface*) is the *de facto* standard on message-passing parallel computation [3]. One of the main factors behind MPI's success is its great flexibility, having few constraints about how a program should exchange its data and how well this data should be distributed over multiple processors. Since parallel (specially MIMD) large-scale computing has assumed a big role on execution of $\mathcal{NP}$-Complete problems, efficient on-line workload management on MPI became central in this type of execution. This work centers on achieving this efficient workload scheduling by adaptating the *Randomized Work Stealing Algorithm* (RWSA), designed originally for distributed thread scheduling, to a MPI-1.2 context.

RWSA was first seen in [1]. It is a high-level algorithm for scheduling multithreaded computations at an homogeneous processor environment. It is basically a distributed algorithm, where each ready, threadless processor prompts (steals) another randomly-chosen processor for work (a thread). It achieves a provable optimal performance. While most work at on-line workload scheduling is normally highlighted on heterogeneous processors, here we are interested at the gain of performance that the RWSA provides on homogeneous processors environments, like most clusters; even though less workload disbalance occurs, a great amount of heavy ($\mathcal{NP}$-Complete) problems may overcharge each occurrence. In this article, we show that an adapted RWSA provides a good and light solution to this kind of disbalance.

Parallel Tree Search (PTS) is a model –structured in MPI prototypes– for parallelizing tree-like progression algorithms, *i.e.*, algorithms based on branch & bound/backtrack. It was proposed on the end of [5] and is designed for dynamically correcting the natural disbalance introduced by the *master-slave* (MS) classical approach, where some processors quickly branch short subtrees and stop computation even if the other processors are still branching the longest ones. This problem, as one may notice, occurs even on the presence of an homogeneous set o processors. PTS fills perfectly for our pourpose, since it may handle huge disbalance of workload at runtime even on an homogeneous processor environment.

Our target application is a branch & bound solution for the Unlimited Knapsack Problem [4], which definition is as follows:

Definition 1.1 (Unlimited Knapsack Problem)**.** Let n be the total number of item types ($n \in \mathbb{N}$). Each $x_i \in X = \{x_1, \dots, x_n\}$ ($X \subset \mathbb{N}$) means the number of items of type i, where each type has associated a value $v_i \in V = \{v_1, \dots, v_n\}$ ($V \subset \mathbb{R}_+$) and weight $w_i \in W = \{w_1, \dots, w_n\}$ ($W \subset \mathbb{R}_+$). Maximun Knapsack supported weight is $C \in \mathbb{R}_+$. The Unlimited Knapsack Problem is, then,

$$\text{to maximize} \quad \sum_{i=1}^{n} (x_i \times v_i)$$
$$\text{subject to} \quad \left(\sum_{i=1}^{n} (x_i \times w_i) \right) \leq C$$

Next section shows an branch & bound algorithm for solving knapsack and how to parallelize it with PTS.

2. The PTS B&B Knapsack Solution

The worst-case complexity of a Knapsack Problem branch & bound solver algorithm (henceforward referred as KBB) is

$$C_p[\text{KBB}](n) \in \mathcal{O}\left(\left(\frac{C}{\text{MIN}\,(W)} \right)^n \right)$$

We are trying to reduce this exponential execution time by introducing paralelism. PST applies when we name a job (or work) a root node of a subtree given by an index on X as on Definition 1.1. It is important to notice that for our B&B Knapsack solution one must guarantee that for the 3-uple input (V, W, C), if $i \leq j$ then $v_i/w_i \leq v_j/w_j$ (*i.e.*, it is sorted by density); this work is not taken into account here.

On PST, all processors have a work list (some container –*e.g.*, a *queue*– with a list of independent data to be processed, potentially out-of-order) and will act as follows:

1. One chosen processor receives the root of the tree and generate a number of δ tree nodes, possibly using *width-first search*. It then scatters the node above all other processors and each other processor does a *depth-first search* on it.
2. Local depth-first search continues until all processor's subtree is exhausted or a limit is reached.
3. Whenever a job is completed, it services any requests it has received for work either with a split from its queue or a message of no remaining work.
4. Whenever its queue is empty, a processor requests for more work from another processor, receiving more work or a message of no remaining work.

The computation ends when all processors have its work list empty (so prompting another processor will not work) and, in some way, synchronize themselves on the final output.

Actually, we have performed a slight alteration on PTS; it originally references MPI *processes* not processors. However, MPI process notion is not accurate (some implementations are over an UNIX process, some are over a set of Windows processes) and since our initial approach is theoretical, we may use an unbounded number of processors and thus allocate one processor per MPI process.

Some major blanks are placed on PTS definitions, making it more generic; *e.g.*, there is no clear indication of which other processor p_j a processor p_i must ask for work. Synchronization for ending computation and the type of container are also very abstract and, thus, does not allow much consideration on the impact of the adopted solution on some algorithm's complexity. Work granularity is also undiscussed.

We propose to fulfill this uncleared aspects with concepts and structures extracted from the multithreaded computation scheduler RWSA; mapping it to PTS context and proving some properties of it must provide a robust paradigm in both theoretical and practical levels. Our test-platform will be an MPI implantation of a PTS version KBB integrated with RWSA. Next section will handle RWSA.

2.1. Mapping Randomized Work Stealing Algorithm into PTS

RWSA acts on the context of a multithreaded computation; having one program that unfolds upon many threads (more threads than processors), this threads are allocated on-line by RWSA in order to provide optimal processor use. Results are only available after execution, since it operates at runtime.

Each processor has a *deque* and, whenever this deque is empty, it steals thereads for another randomly chosen processor. Please, see [2] for further details.

It has been proven that RWSA has an expected execution time of $T_1/P + \mathcal{O}(T_\infty)$ on a fully strict computation, where T_1 is execution time with one processor, P is the number of processors and T_∞ (*critical path lenght*) is the minimum execution time with an unbounded number of processors. Similarly, the space required for computation is $\mathcal{O}(S_1 P)$ where S_1 is the minimum serial space requirement and any commnication latency is proportional just to the number of messages and not to any singular snapshot constraint.

We expect to maintain some of this properties by mapping RWSA in PTS, obtaining an optimal PTS version as follows. That is done in two ways: structural mapping (where PTS data structures are mapped into RWSA DAG) and algorithm-step mapping (where algorithm-specific steps of RWSA are mapped into the PTS ones).

Structural mapping is made this way:

PTS	RWSA	Method
container	deque	As main container of PTS we use RWSA's deque (*double ended queue*) concept.
work/job	thread	We push jobs to the deque.
job/work unfold step	task	Each step of the node unfold on PTS is a task at RWSA, since it is sequential and dependent from the previous one.
subtree space	activation frame	total space of a job is equal to the memory spent on holding root node's subtree.

Algorithm-step mapping is quite more complex:

1. Thread spawning on RWSA is only made at the beginning, being root's initial input decomposing on PTS

KBB. In this case, there are no dependency edges, since no job depends on its predecessor. It is clearly fully strict, by lack of counter-example.

2. Thread stalling on RWSA does not have a mapping on PTS, since no subtree unfolding blocks.

3. Thread death on RWSA is a complete branch unfold on PTS. Like RWSA, next job is taken from bottom-most position of the tree. Whether there are no jobs on the ready deque, it steals the topmost job from a deque of a randomly chosen process.

4. Thread enabling on RWSA does not have a mapping on PTS since a stall does not have it too.

Supposing that all that was left remains like on original RWSA (*v.g.*, synchronized message attending), we may propose the following corollary, without presenting the proof:

Corollary 2.1 (PTS/RWSA Kept Properties)**.** *By the above RWSA $\rightarrow$ PTS mapping, we maintain execution space at $\mathcal{O}(S_1P)$ and the expected delay on attending proportional to the total number of messages M.*

Although proof is omitted, it is easily traced by making the fundamental observations that no proposed changes to RWSA's core to adapt it for PTS violates

1. the proof of space complexity S_1P, which not depends on stalling and enabling threads; or

2. the proof of delay time for receiving work proportional only to the number M for exchanged messages, since probability remains $1/P$ for each x_{ir} on *Balls & Bins* game.

Next section will avail the benefits of both properties holding, presenting some valuable practical results.

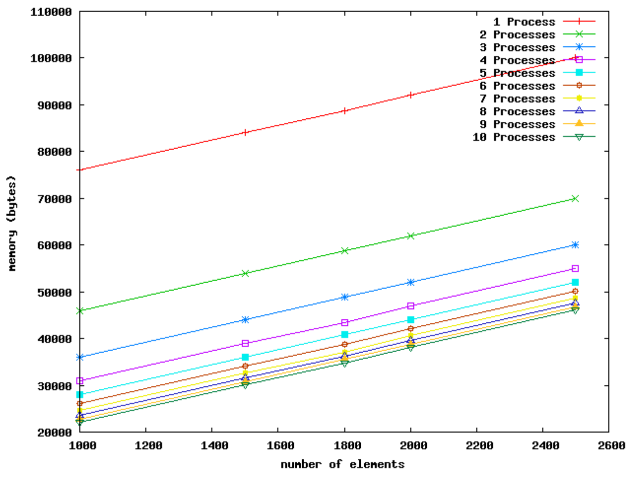
2.2. Results

We have runned PST KBB with MPI-1.2 for showing practical results based on our previous observations. To our next considerations, graphical data will be plotted. Our experiments have been run on GPPD's *labtec* cluster, with LAM MPI and the following configurations:

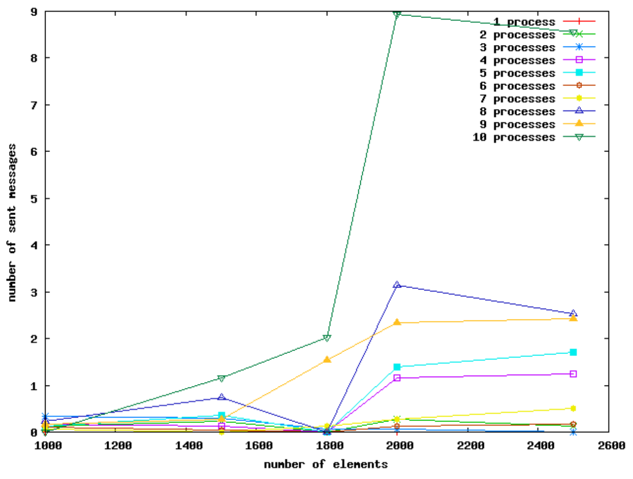
- each configuration run for 30 times, with arithmetical average taken only when standard deviation were less than 0.01, what always happened before all the executions.
- 1000, 1500, 1800, 2000 and 2500 item types.
- $1 \leq P \leq 10$.

First, Figure 1(a) confirms our assumption of linear memory grown variating with n, showing physical evidence of Corollary 2.1's first part.

Corollary 2.1's second part implies that the expected message delay is proportional to M and does not vary with n. Indeed, Figure 1(b) shows that message exchange is pretty chaotical, relying more on tree shape, bound operations and the difference in performance from one processor to another (even on an homogeneous processor environment like *labtec*, difference in performances occur, since memory hierarchy does not have linear behavior).



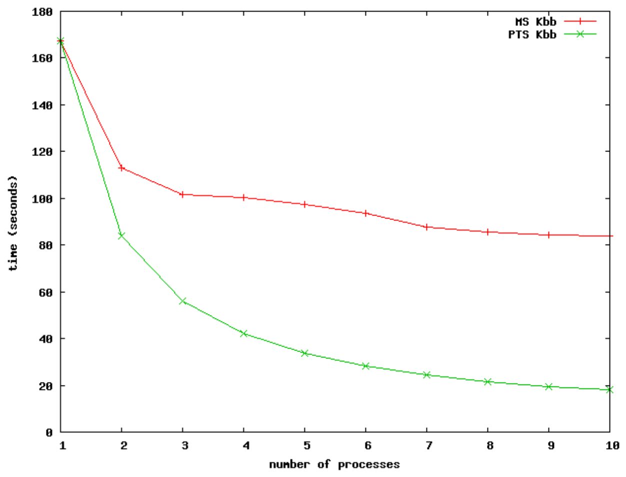
(a) Memory growing linearly with n.



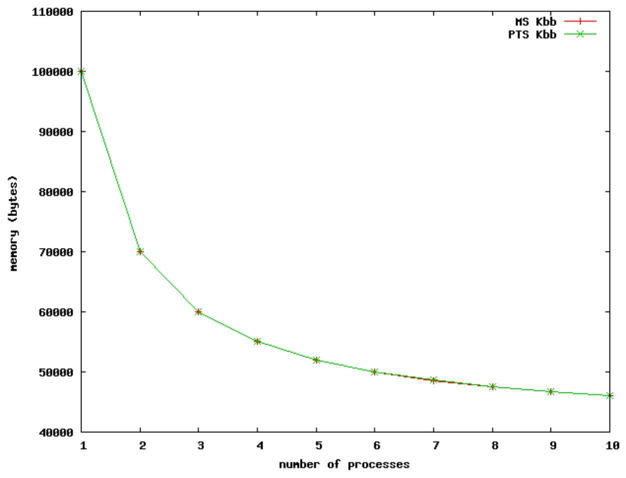
(b) Message exchange unpaired with n

Figure 1. Linear memory grownth and chaotic communication behavior with n, as in Corollary 2.1; PST on KBB scales well also because there is no space complexity adding.

With Corollary 2.1 holding, we must ensure that PST KBB still grants great performance gain. In Figure 2(a) we show that our PST implementations can be until 80 percent faster than an MS version for the same input; this master-slave approach will have its resources' loads unbalanced, since all work is previously distributed and cannot be recupered. We do not show PST RWSA against PST *ad hoc* because we have not found any *ad hoc* configuration for PST that could clearly be more efficient than the former. Also,



(a) Execution time of PST KBB (—×—) against an MS KBB (—+—).



(b) Memory allocation for a single process of PST KBB (—×—) against an MS KBB (—+—).

Figure 2. PST KBB *vs.* MS KBB : execution time and memory allocation.

Figures 2(b) and 3 shows that memory consumption of PST KBB is the same of MS KBB while it achieves performance very close to linear speedup.

2.3. Conclusion & Future Work

Our approach for filling the blanks on original PTS proved to achieve good results so far. By mapping concepts of original RWSA we have also mapped some of its properties and showed that it really holds on a practical context. However, this work lacks of theoretical basis for foundamenting performace gain, since our mapping of RWSA is not yet proven to maintain execution time at $T_1 + \mathcal{O}(T_\infty)$ and, so, we cannot rely on this single result to generalize it. Our next steps are clearly directed to reach this proof, by proposing and testing new ways of mapping RWSA thread context into a cluster context.

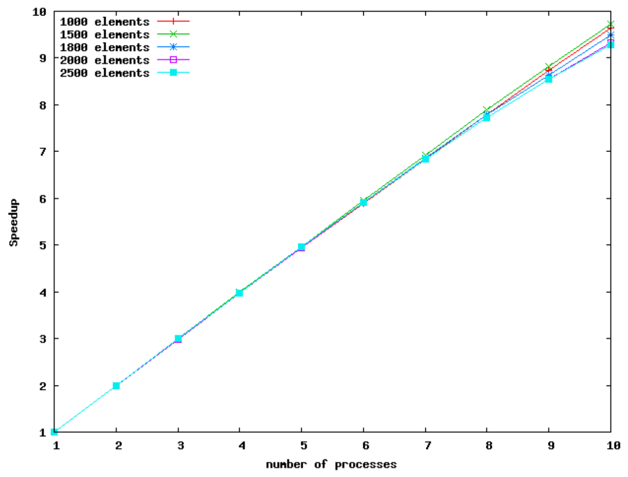


Figure 3. Very close to linear speedup achieved with PST KBB.

Our approach can be extended to an interesting subset of the $\mathcal{NP}$ class. Since any problem of $\mathcal{NP}$ could be automatically converted to a Knapsack problem in polynomial time, those problems with linear conversion time might prove efficiently resolved by our solution with very good performance. Our work goes in that direction too.

References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. MIT Laboratory for Computer Science, 1994.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of ACM*, 46(5):720–748, September 1999.

[3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - Portable Parallel Programming with Message-Passing Interface*. Scientific and Engineering Computation Series. The MIT Press, Massachusetts Institue of Technology - Cambridge, Massachusetts 02142, 2 edition, 1999.

[4] H. Keller, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2005.

[5] P. S. Pacheco. *Paralell Programming With MPI*. Morgan Kaufmann Publishers, 1 edition, 1997.