Work Stealing for Multi-core HPC Clusters

Kaushik Ravichandran, Sangho Lee, and Santosh Pande

College of Computing, Georgia Institute of Technology, USA
kaushikr@gatech.edu, slee431@gatech.edu, santosh@cc.gatech.edu

Abstract. Today a significant fraction of HPC clusters are built from multi-core machines connected via a high speed interconnect, hence, they have a mix of shared memory and distributed memory. Work stealing algorithms are currently designed for either a shared memory architecture or for a distributed memory architecture and are extended to work on these multi-core clusters by assuming a single underlying architecture. However, as the number of cores in each node increase, the differences between a shared memory architecture and a distributed memory architecture become more acute. Current work stealing approaches are not suitable for multi-core clusters due to the dichotomy of the underlying architecture. We combine the best aspects of both the current approaches in to a new algorithm. Our algorithm allows for more efficient execution of large-scale HPC applications, such as UTS, on clusters which have large multi-cores. As the number of cores per node increase, which is inevitable given today's processor trends, such an approach is crucial.

Keywords: dynamic load balancing, unbalanced tree search, multi-core.

1 Introduction

Today, a large portion of HPC systems are built from multi-core machines connected through high speed interconnects such as InfiniBand. This kind of architecture is seen in systems in the Top 500 list such as Jaguar (Oak Ridge National Laboratory), Hopper (NERSC) and Kraken (National Institute of Computational Sciences). These systems have two distinct kinds of parallelism: intra-node and inter-node. Intra-node parallelism (through shared memory) is due to the existence of multiple cores in a single node, while inter-node parallelism (through distributed memory) is due to the presence of a large number of such nodes. The number of cores per node, and hence intra-node parallelism, is increasing as larger and larger multi-cores become commonplace.

Work stealing algorithms are currently designed for either a shared memory architecture or for a distributed memory architecture and are extended to work on these multi-core clusters in a straight forward manner. The distributed memory implementations extend naturally into a multi-core cluster environment by running separate tasks on different cores. Shared memory implementations are typically extended by using PGAS (Partitioned Global Address Space) languages such as UPC (Berkley Unified Parallel C). PGAS languages present a

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 205–217, 2011.

unified address space over the underlying distributed memory allowing the algorithm to scale across the cluster. Termination detection algorithms are similarly extended.

However, as the number of cores in each node increase (due to larger multi-cores), the differences between a shared memory architecture and a distributed memory architecture become more acute. Current approaches which extend a completely shared memory paradigm or a distributed memory paradigm to a cluster can be improved. Combining the best aspects of both approaches results in a new algorithm which allows for a more efficient execution of large-scale HPC applications, like UTS, which need efficient dynamic load balancing. As the number of cores per node increase, which is inevitable given today's processor trends, such an approach is crucial.

The UTS (Unbalanced Tree Benchmark) [14] is representative of the class of applications which process highly unbalanced workloads. We demonstrate the effectiveness of our approach on the UTS benchmark and report significant speedups on large multi-core clusters over current shared memory implementations and distributed memory implementations.

2 Work Stealing

A fundamental problem in achieving maximum performance from HPC applications is that of dynamic load balancing. Many applications exhibit a large variability in the amount of work they dynamically generate. This variability gives rise to an imbalance in the parallel execution of these applications. Variability could be caused by many reasons. For example, the random nature of input data could cause imbalance between the workloads of different parallel processing elements. The UTS (Unbalanced Tree Search) benchmark [14], is representative of the class of parallel applications which require dynamic load balancing. The benchmark has been carefully designed to be *the* optimal adversary to load balancing strategies. In this paper we shall focus on the UTS benchmark to test our algorithm, both because of its popularity and because of its ability to stress dynamic load balancing algorithms effectively.

Load balancing can be broadly divided into two categories: static and dynamic. Static approaches have been well studied [16,12]. These approaches, however, are not suitable for the kind of applications we are concerned about due to the unpredictability of the problem space and dependence on the input parameters and dataset. Dynamic load balancing algorithms have been proposed to address the types of applications we are looking at.

Many dynamic load balancing algorithms have been proposed. Two popular algorithms used on both architectures (shared as well as distributed memory) are work sharing and work stealing. Work sharing involves balancing of the workload using a globally shared task queue. Work stealing on the other hand follows a completely distributed approach where idle processing elements take on the onus of finding work by looking around the system and has been found to be more efficient [6]. Its effectiveness lies in the fact that it puts a majority of the overhead on idle processors, minimizing the load on busy processors and

minimizing the need for global information. Work stealing has been proven to be optimal for a large class of problems and has tight space bounds [2], thus, making it the method of choice for large scale distributed clusters.

Termination detection is a critical postlude to work stealing algorithms. This step allows programs to recognize when there is no more work in the system. As we have described before, in the work stealing method, once a processing element becomes idle, it looks around for work that it can steal from other processing elements. Indeed, it is possible that all processing elements have completed their work and are simply looking around for more work endlessly. The process of detecting such a system state and ending the execution is termination detection. Termination detection algorithms, akin to work stealing algorithms have different implementations on shared memory architectures and on distributed memory architectures.

Shared memory architecture. On shared memory architectures, work stealing has been used to effectively parallelize unbalanced workloads. Implementations such as Cilk [9], have popularized work stealing by implementing it in the runtime system. Typically, each processing element maintains a double ended queue in shared memory. When it needs to process a task, it takes a task off the front of its queue and processes it. Any new tasks are added to the front of the queue. When a processing element completes all the tasks in its queue, it becomes idle and begins work stealing. It looks for other processing elements which have excess tasks in their task queues. Once it locates such a queue, it takes tasks from the back of that queue and pushes it onto its own. Of course, the implementation needs to be highly tuned to reduce excess locking overhead. Methods like "THE" [9] eliminate a majority of the situations in which locks are needed. Such methods split the queues into private and public sections, eliminating the need for locking in the private sections while still requiring locking in the public section.

Termination detection in shared memory architectures can be achieved using special kinds of barriers, called cancellable barriers which allow threads to "check-in" and "check out". These barriers are especially suited for shared memory work stealing algorithms.

Attempts have been made to extend such work stealing algorithms to HPC clusters which contain both shared memory and distributed memory [6]. To extend these algorithms to a cluster, an abstraction is needed to apply a shared memory paradigm over the entire cluster. Partitioned Global Address Space (PGAS) languages allow precisely this. These languages allow programmers to write code, assuming a shared memory architecture and the PGAS programming model takes care of the rest. It implicitly converts any cross machine memory accesses into messages using interfaces such as MPI.

While this approach leads to a correct implementation, it is not necessarily efficient, for several reasons. The cross machine memory accesses that are converted into messages cause increased contention, runtime overhead and latency. These overheads can be minimized to some extent by careful tuning of the messages that are sent by using one sided reads and writes and RDMA (Remote

Direct Memory Access). However, using one sided reads and writes is a much more complicated affair and involves a lot more effort. A more serious disadvantage of using RDMA is that often, the underlying system needs to dedicate one core from each SMP to address these accesses transparently and efficiently [6].

Distributed memory architecture. Work stealing algorithms for distributed memory architectures differ from shared memory architectures for several reasons. For one, on shared memory architectures, synchronization primitives, caching and coherence protocols are often taken for granted as the underlying hardware takes care of these issues. Implementing these global operations in a distributed memory architecture often results in high runtime overheads and latencies. Another reason is that some algorithms simply do not scale. For example, it is no longer feasible to allow different tasks to spin on a common memory location, due to the absence of shared memory. Attempting to do so, would introduce a terrible amount of contention at certain nodes due to messaging. Clearly, different algorithms are needed which are suited for distributed memory architectures.

Solutions which use direct management of communication operations using explicit message passing have been shown to be viable [1]. Different algorithms such as Dijikstra's Termination Detection algorithm [4] are more suitable than cancellable barriers for termination detection in a distributed setting.

Practical solutions are typically designed assuming a completely distributed memory, using MPI or a similar message passing interface. A typical multi-core HPC cluster, however, has both shared and distributed memory. It is extremely straight forward to extend the implementation to an entire multi-core cluster by simply running different tasks on different cores, irrespective of whether or not they share any common memory. This is not an optimal solution, since communicating via MPI is certainly slower than through shared memory when it exists. However, this solution is still correct and allows for the execution of these work stealing algorithms across an entire cluster.

Our approach. Though extending the shared memory paradigm or the distributed memory paradigm to an entire multi-core cluster maybe correct, it is not necessarily efficient. As multi-core machines become more and more prevalent it is crucial to recognize the differences between shared memory architectures and distributed memory architectures. Our approach uses aspects from both methodologies to come up with a new more efficient algorithm.

Our approach uses two different load balancing strategies. One inside a multi-core node and one across multi-core nodes. For intra-node load balancing we use a popular algorithm which uses lockless task queues in shared memory and cancellable barriers for termination detection, while for inter-node load balancing we switch over to a pure MPI implementation and a termination detection algorithm similar to Dijikstra's. We show that such an approach is more efficient than previous approaches when there are a larger number of cores per node.

The ideas behind our approach can also be used to improve previous implementations. We would like to stress, that our ideas are orthogonal to previous approaches and they too can benefit from our techniques.

3 Design for Our Approach

3.1 Shared Memory Design

Our approach uses a popular algorithm for work stealing in shared memory multi-cores [6]. Split queues are used to alleviate locking overhead and a cancellable barrier is used to detect termination. Each core runs a single thread which is responsible for executing tasks.

The task queues are accessed very frequently and hence operations on them must provide efficient access. Each thread has one local task queue that it uses to maintain its list of tasks. When a thread generates more tasks and needs to add it to its local task queue, it enqueues the tasks at the front of the queue. When a thread needs to remove tasks from the queue, it dequeues them, again, from the front of the queue. When threads become idle, they search other task queues for work. If they find work in some other task queue, they will steal it by dequeue-ing it from the back of the queue.

First and foremost, this task queue should provide efficient access to the local thread, since it is on the critical path of execution. Any delays on task queue operations will directly be reflected in the execution time of the application. Other threads also need access to the task queue to enable work stealing. Concurrent access can be achieved by using a simple locking mechanism on the task queue. This would however add locking overhead for the local thread as well with every enqueue and dequeue operation. To alleviate the locking overhead we can use a single queue, but divide it into two distinct regions: a local region and a global region. The local region would remain lock free for access by the local thread and the global region would be synchronized through a lock. This is called a split queue (Figure 1).

Split queues need additional operations on top of the regular enqueue and dequeue operations to function properly. A thread continuously inserts tasks into the local portion of the task queue. If there is a sufficient amount of work in the local region it can choose to expose the excess work into the global region of the task queue. This operation involves invoking the lock of the global region of the queue. This operation of moving work into the global region of the queue is called the *release* operation. The *release* operations must be performed periodically ensuring enough work for other threads to steal. Correspondingly, when work in the local region of the split queue has been completed, it is then necessary to get some of the work from the global region of the queue (if it exists) back into the local part of the queue. This can be accomplished by simply moving the boundary between the local and global regions of the queue, further towards the global region. This operation must also be locked and is known as the *reacquire* operation. The *reacquire* operation is only performed when the work in the local portion is exhausted. Using the *release* and *reacquire* operations, locking is minimized to the global portions of the queue and the

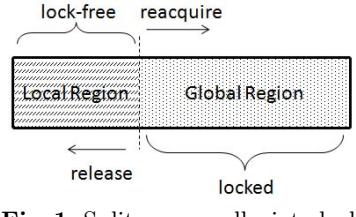


Fig. 1. Split queues alleviate locking overhead

accesses to the local region are lock free, except for the *release* and *reacquire* operations. This contributes minimally to the critical path overhead.

This kind of work stealing follows the depth-first work and breadth-first steal technique [2] that is used by many implementations including Cilk [9]. Our shared memory design is implemented in OpenMP.

Termination detection. Termination detection is achieved using a cancellable barrier. Cancellable barriers allow threads to check-in and also check-out when more work has been released into the system. Once a thread finds that the global regions are empty, it checks into the cancellable barrier. Indeed, this does not mean that there is no work in the system, it simply means that there is no work in the global region of any of the task queues. Other threads could be processing tasks and they could have tasks in their local regions which have not yet been released into the global region. Barriers can be canceled by another thread when it releases work into the system by performing a *release* operation. This kind of cancellable barrier does not scale across nodes in a cluster because of its centralized nature.

3.2 Distributed Memory Design

So far we have described the design of a shared memory implementation of work stealing. To expand this across the cluster we need a way to steal across nodes. We could have used a programming model like PGAS to convert any shared memory steals across nodes into messages sent through the underlying interconnect. However, due to remote locking latencies and overheads of the PGAS language and runtime, we found that co-ordinating cross node steals through MPI was much more efficient.

Hence, MPI is used to achieve load balancing across nodes. MPI provides for efficient message passing which is under our control rather than a PGAS language [14,7]. One important consideration which guided our design was that stealing from a thread in the same node is many times faster than stealing from another node in the cluster (in our experiments, about 50 times faster). Hence, we always perform work stealing inside a node before we cross the node boundary. This important guiding principle can be applied to any work stealing algorithm for improved performance.

One approach to enable work stealing across nodes would be to allow individual threads to send MPI steal requests to different nodes when they detect that there is no more work in the local node. This design requires a multi-threaded implementation of MPI and introduces locking overheads in the MPI runtime. To avoid any locking overhead and to maximize portability of our implementation we needed to use a single threaded implementation. This led us to choose a design in which we designate one thread per node to be in charge of cross node MPI steal requests. This designated thread will send cross node steal requests only when all the work on the local node has completed since stealing intra-node is much more quick than stealing across nodes. The designated thread (hereafter referred to as *thread0*) is also in charge of responding to steal requests from other nodes as well as termination detection.

thread0 will begin sending out cross node steals when it has exhausted all the tasks on its task queue and when all the other threads have checked into the cancellable barrier. We provide a special check-in mechanism for *thread0* so that it can "peek" into the current state of the barrier and wait till all the other threads have arrived at the barrier. By using this "peek" check-in,*thread0*, can determine when all the other threads have finished. If some thread generates work or if *thread0* finds work, the barrier gets canceled and all the other threads resume work stealing.

thread0 of a given node will choose a victim and send out a steal request to that victim. The *thread0* of every node processes tasks from its task queue just like every other thread, but in addition, it also periodically checks for any new steal messages it might need to service. If it has work in its local task queue, it will dequeue several nodes (controlled by parameter $chunksize$) it and send it out the thief. If it has no work, it will respond with the fact that it has no work. If *thread0* of a node, sends out a steal request and it gets back work, it will enqueue it onto its local task queue and cancel the barrier to wake all other thread up. If on the other hand, it gets a message saying that there is no work at the victim node, it will choose another victim and proceed. Using this approach, the maximum number of outstanding messages in the system will be bounded by the number of nodes since each node sends out steal requests one at a time. Different methodologies can be adopted for choosing the order in which victims are selected, however, we employ random work stealing which has been proven to be optimal [2].

If we had used a shared memory paradigm (a PGAS language like UPC) the steal operations will disturb the working threads in other nodes because these threads will be forced to wait for the global regions of these task queues to be unlocked to perform any *release* or *reacquire* operations. The cost of these interfering remote locking operations is typically an order of magnitude greater than the cost of a shared variable reference [15]. However, our implementation using MPI similar to [7], enables the thread to perform operations on their local task queues without waiting for locks from threads external to the node (which have the highest latency). They simply service steal requests at regular intervals removing the necessity of locks from external threads. Our distributed memory design is implemented in MPI.

Termination detection. The described techniques enable efficient work stealing across nodes in a HPC cluster, but this is not enough. We also need to detect termination across all the nodes in the cluster. If after attempting to steal from other nodes, *thread0* does not find any work in other nodes it will begin the global (across cluster nodes) termination detection process.

For termination detection in a single node we employed a cancellable barrier. This approach is simply not scalable to an entire cluster for reasons explained previously. We need a different algorithm to detect termination across the cluster. Many algorithms have been proposed in literature. We chose to use a modified version of the well known Dijikstra's termination detection algorithm [5] similar to [7] which is a token based termination detection algorithm. We refrain from describing the algorithm here due to a lack of space. Details can be found in [5].

Recall, that for *thread0* to have participated in the global termination detection process, it must have finished sending out all its cross node steal requests. And to have sent out cross node steal requests, it must have been the case that the other threads in the node are still waiting on the cancellable barrier. Once, *thread0* determines that global termination has been reached, it performs a complete check-in to the cancellable barrier on the node (as against a "peek" check-in that it normally performs). Once, *thread0* checks in, threads waiting at the cancellable barrier are notified that global termination has been reached and all threads terminate execution.

3.3 Combined Approach

The combination of the above schemes provides for a very efficient work stealing approach for HPC clusters. To summarize, we prioritize intra-node steals over inter-node steals and use a different algorithm for intra-node steals (asynchronous steals and cancellable barriers) and for inter-node steals (polling for steal requests and Dijikstra's termination detection). Note that while better victim selection in the previous approaches would improve their performance it is insufficient and the use of two different algorithms is vital. Figure 2 depicts all the interactions in the form of a state diagram.

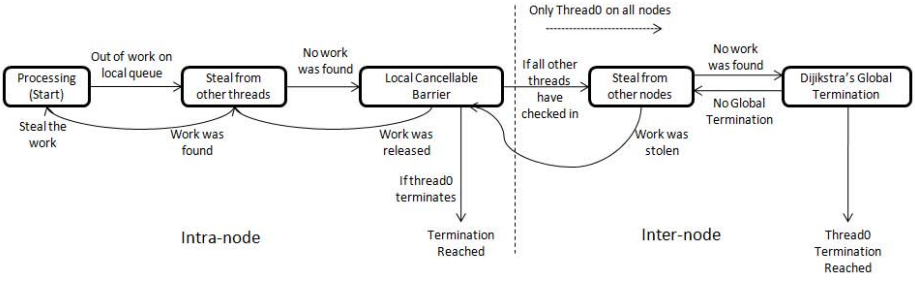


Fig. 2. Interactions of the algorithm

4 Evaluation

4.1 UTS

The Unbalanced Tree Search benchmark has been designed to be representative of a class of HPC applications which require substantial dynamic load balance [14]. Applications that fit this category include many search and optimization problems that must enumerate a large state space of unknown or unpredictable structure. UTS has become *the* benchmark with which load balancing algorithms are benchmarked. The benchmark is itself based on a simple problem - the parallel exploration of an unbalanced tree. The problem is to count the total number of nodes that this tree generates. The tree is generated through an implicit construction that is parametrized in shape, depth, size and imbalance. The tree is generated by traversing down the tree from the root node. When we process the root, its children (nodes) are generated. Now, each of these nodes

are recursively processed to generate the entire tree. While the processing time of each node is fairly constant, there is a high amount of variance in the sizes of each of the sub-trees leading to imbalance in workloads. [14] can be referred for a more detailed explanation. Figure 3 depicts an unbalanced tree showing a weblog file [10]. There are two types of trees that are used as part of the benchmark: Geometric Trees and Binomial Trees. Details can be found in [14].

4.2 Results

In this section we compare our implementation with two highly tuned state of the art implementations: one which extends a shared memory paradigm to a multi-core cluster using UPC [15] and another which extends a distributed memory paradigm to a multi-core cluster using purely MPI [7].

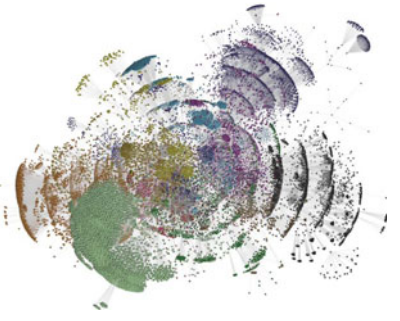


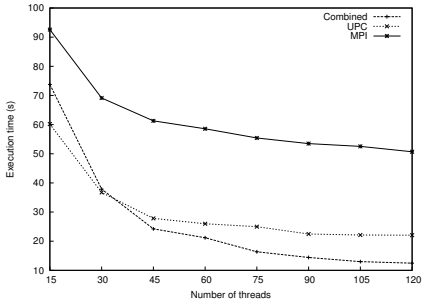
Fig. 3. Unbalanced tree showing a weblog

The UPC implementation extends a shared memory paradigm to the entire multi-core cluster. Accesses to remote memory are converted under the hood into cross node MPI messages. It has been found that simply using UPC and extended a shared memory algorithm is simply not scalable [15,14,7] and the authors of this implementation had to use several techniques such as polling instead of asynchronous stealing to improve performance to an acceptable level. In our experiments we have found this tuned UPC implementation to be faster than the MPI version, consistent with previous findings [15]. Our approach is significantly faster than the current implementations at higher thread/node counts. In this section we will refer to the three implementations as: the Combined approach (our method), the UPC approach and the MPI approach.

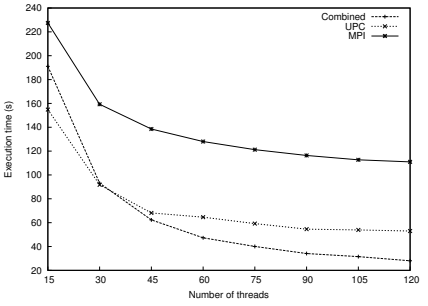
For our experiments we used a 15-node, 120-core IBM BladeCenter H Linux cluster with 2 socket x Core2 quad processors. Each node supported the parallel execution of up to 8 threads. This allowed us to observe the behavior of the three implementations as we increased the number of threads on each core from 1 to 8. The nodes in the cluster were connected using Ethernet and we used MPI for message passing across nodes. The UPC implementation was compiled with Pthreads support which enabled intra-node communication to happen through shared memory.

Tests were performed using 3 trees generated by the UTS benchmark. We used two geometric trees (GEO1 and GEO2) and one binomial tree (BIN1)[1]. We increased the number of threads running per core from 1 to 8 hence scaling

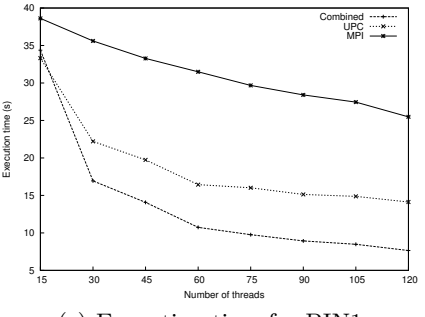
[1] For reproducibility we provide the exact parameters used to generate the tree.
GEO1 (1635119272 nodes): Geometric (Fixed), $d = 15$, $b_0 = 4$, $rootseed = 29$.
GEO2 (4230646601 nodes): Geometric (Fixed), $d = 15$, $b_0 = 4$, $rootseed = 19$.
BIN1 (1060980001 nodes): Binomial, $b_0 = 2000$, $q = 0.024999999975$, $m = 40$, $rootseed = 316$.



(a) Execution time for GEO1



(b) Execution time for GEO2



(c) Execution time for BIN1

Fig. 4. Execution Time

Table 1	Inter-node	Intra-node
Combined	5233	132919
UPC	13128	1501
MPI	800501	N/A

Table 2	Inter-node	Intra-node
No. steals	5233	132919
Total time	0.25923	0.13852
Time/steal (s)	0.000049539	0.000001042

(a) Table 1: Number of steals for different approaches. Table 2: Time taken for each steal in the Combined Approach in sec.

Fig. 5. Steals Breakdown

the implementation from 15 threads (one on each node) to 120 threads (8 on each node). Figure 4 shows the results we obtained.

Discussion. Our implementation which combined the use of two different algorithms consistently performed better than the two other implementations at higher core counts on GEO1, GEO2 and BIN1. The MPI implementation is slower than both the combined approach and the UPC approach in all the tests. Let us consider the case when only 1 thread runs on each node of the cluster. We call this the *baseline case*. Each additional test increases the number of threads on the node by 1. In the *baseline case*, the Combined approach is slower than

the UPC approach by as much as up to 23%. With 2 threads running per node, the Combined approach is slower than the UPC approach by approximately 2%. However, as we increase the number of threads per node, the Combined approach consistently performs better, with almost a 20% improvement when using 4 threads/node which only increases as we increase the number of threads/node.

The fact that the Combined approach is slower than the UPC implementation when using a smaller number of threads points to two facts. That at lower thread counts, the overheads of using two separate termination detection and stealing algorithms slows down the overall execution and also that the base UPC implementation is highly tuned and performs very well on machines with low threads/node. However, we observe that at higher threads/node counts the Combined approach provides significant speedups over the UPC approach and the MPI approach.

Table 1 in 5(a) shows the breakdown of the number of inter-node and intra-node steals performed by the various approaches. These numbers were obtained by running GEO2 with a total of 90 threads on 15 nodes. The Combined approach prioritizes intra-node steals over inter-node steals. The UPC implementation however, performs a much higher number of inter-node steals while compared to intra-node steals (however, roughly the same proportion considering that there are 15 nodes). The MPI implementation performs only inter-node steals since we assume a completely distributed memory. Table 2 in 5(a) points to the fact that inter-node steals are almost 50 times slower than intra-node steals!

The underlying reason behind the speedups is the fact that we use two different algorithms for inter-node steals and intra-node steals. The speedup cannot be attributed to the use of shared memory alone, since the UPC implementation was compiled with Pthreads support which enables use of shared memory for threads which are collocated on the same node. Similarly the speedup cannot be attributed to improved victim selection (based on locality). In our experiments we found that tuning victim selection in the UPC and MPI approach improved performance by only a modest 2% (at 8 threads/node). We conclude that the most important factor in the speedup is the use of a different algorithm for inter-node steals and intra-node steals. Retrofitting either a shared memory paradigm or a distributed memory paradigm on top of a multi-core cluster does not perform as well at higher thread/node count.

5 Related Work

A large amount of effort has been invested in studying different kinds of load balancing algorithms. Load balancing methods have been broadly classified into static methods and dynamic methods. Static methods such as [16,12] are suitable in situations where work can be divided fairly before execution. Task graph scheduling [13] finds a schedule given a set of tasks which are organized as a graph. Dynamic approaches so far have focused on using a distributed memory approach using MPI [7] and techniques like RDMA [17] or using a shared memory approach [14] using OpenMP. PGAS languages extend the shared memory

paradigm across an entire cluster (containing nodes with distributed memory) hence allowing load balancing algorithms to work across an entire cluster. Certain languages like X10 [3], Cilk [9] employ dynamic load balancing techniques. Hierarchical techniques have been proposed in ATLAS [11]. Work stealing techniques, in general, have been well studied and have been shown to be applicable to various applications on distributed memory machines such as [17,8].

6 Conclusion and Future Work

In this paper we propose an algorithm which uses different algorithms for inter-node and intra-node steals and demonstrate marked improvements on the UTS benchmark over current implementations. Future work involves optimizing our implementation using various techniques such as work pushing and using smarter victim identification schemes. We strongly believe that the concepts that are presented in this paper can be applied to current implementations and can be used in designing work stealing algorithms in the future. We would like to thank the anonymous reviewers for their comments. We also gratefully acknowledge the support of NSF grants CCF-1018544 and CCF-0916962.

References

1. Berlin, K., Huan, J.: Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 194–208. Springer, Heidelberg (2004)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46, 720–748 (1999)
3. Charles, P., Grothoff, C., Saraswat, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40, 519–538 (2005)
4. Scholten, C.S., Dijikstra, E.W.: Termination detection for diffusing computations (1980)
5. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters 11(1), 1–4 (1980)
6. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 53:1–53:11. ACM, New York (2009)
7. Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., Tseng, C.-W.: Dynamic load balancing of unbalanced computations using message passing. In: IPDPS 2007, IEEE International, pp. 1–8 (2007)
8. Dowaji, S., Roucairol, C.: Load balancing strategy and priority of tasks in distributed environments (1994)
9. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. SIGPLAN Not. 33, 212–223 (1998)
10. Isenberg, P.: Phyllotactic patterns for tree layout, `http://pages.cpsc.ucalgary.ca/~pneumann/wiki/pmwiki.php?n=MyUniversity.PhylloTrees`
11. Eric Baldeschwieler, J., Blumofe, R.D., Brewer, E.A.: Atlas: An infrastructure for global computing (1996)

12. Kim, C., Kameda, H.: An algorithm for optimal static load balancing in distributed computer systems. IEEE Trans. Comput. 41, 381–384 (1992)
13. Kwok, Y.-K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. 31, 406–471 (1999)
14. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.-W.: Uts: an unbalanced tree search benchmark. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
15. Olivier, S., Prins, J.: Scalable dynamic load balancing using upc. In: ICPP 2008, pp. 123–131. IEEE Computer Society, Washington, DC, USA (2008)
16. Tantawi, A.N., Towsley, D.: Optimal static load balancing in distributed computer systems. J. ACM 32, 445–465 (1985)
17. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. SIGPLAN Not. 36, 34–43 (2001)