Dynamic Load Balancing of Unbalanced Computations Using Message Passing

James Dinan[1], Stephen Olivier[2], Gerald Sabin[1], Jan Prins[2],
P. Sadayappan[1], and Chau-Wen Tseng[3]

[1] Dept. of Comp. Sci. and Engineering
The Ohio State University
Columbus, OH 43221
{dinan, sabin, saday}@cse.ohio-state.edu

[2] Dept. of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599
{olivier, prins}@cs.unc.edu

[3] Dept. of Computer Science
Univ. of Maryland at College Park
College Park, MD 20742
tseng@cs.umd.edu

Abstract

This paper examines MPI's ability to support continuous, dynamic load balancing for unbalanced parallel applications. We use an unbalanced tree search benchmark (UTS) to compare two approaches, 1) work sharing using a centralized work queue, and 2) work stealing using explicit polling to handle steal requests. Experiments indicate that in addition to a parameter defining the granularity of load balancing, message-passing paradigms require additional parameters such as polling intervals to manage runtime overhead. Using these additional parameters, we observed an improvement of up to 2X in parallel performance. Overall we found that while work sharing may achieve better peak performance on certain workloads, work stealing achieves comparable if not better performance across a wider range of chunk sizes and workloads.

1 Introduction

The goal of the Unbalanced Tree Search (UTS) parallel benchmark is to characterize the performance that a particular combination of computer system and parallel programming model can attain when solving an unbalanced problem that requires dynamic load balancing [16]. This is accomplished by measuring the performance of executing parallel tree search on a richly parameterized, unbalanced workload. In particular, the workloads explored by UTS exemplify computations where static partitioning schemes cannot yield good parallelism due to unpredictability in the problem space.

Shared memory and Partitioned Global Address Space (PGAS) programming models (e.g. OpenMP, UPC, CAF, or Titanium) provide a natural environment for implementing dynamic load balancing schemes through support for shared global state and one-sided operations on remote memory. On shared memory machines, the ability to offload coherence protocols, synchronization operations, and caching of shared data into the hardware gives these systems a great advantage over distributed memory systems where supporting such global operations often results in high latency and runtime overhead. For this reason, the best performance on distributed memory systems can often only be achieved through direct management of communication operations using explicit message passing [3].

Parallel programming models based on two-sided message passing pose significant challenges to the implementation of parallel applications that exhibit asynchronous communication patterns. Under these models, establishing support for computation-wide services such as dynamic load balancing and termination detection often introduces complexity through the management of additional non-blocking communication operations and the injection of explicit progress directives into an application's critical path. The need to explicitly manage this complexity for high performance exposes the programmer to additional opportunities for race conditions, starvation, and deadlock.

In this paper, we present an implementation of the UTS benchmark using the Message Passing Interface (MPI). We

1-4244-0910-1/07/$20.00

explore approaches to mitigating the complexity of supporting dynamic load balancing under MPI and investigate techniques for explicitly managing load balancing activity to increase performance. Our discussion begins with a presentation of the Unbalanced Tree Search problem followed by a discussion of the implementation of both centralized and distributed load balancing schemes using MPI. We then explore techniques for fine tuning load balancing activity to enhance performance. Finally, we evaluate these schemes on a blade cluster and characterize the performance and scalability attainable in the presence of dynamic load balancing for such systems.

2 Background

2.1 Unbalanced Tree Search Benchmark

The unbalanced tree search (UTS) problem is to count the number of nodes in an implicitly constructed tree that is parameterized in shape, depth, size, and imbalance. Implicit construction means that each node contains all information necessary to construct its children. Thus, starting from the root, the tree can be traversed in parallel in any order as long as each parent is visited before its children. The imbalance of a tree is a measure of the variation in the size of its subtrees. Highly unbalanced trees pose significant challenges for parallel traversal because the work required for different subtrees may vary greatly. Consequently an effective and efficient dynamic load balancing strategy is required to achieve good performance.

The overall shape of the tree is determined by the *tree type*. A major difference between tree types is the probability distribution (binomial or geometric) used to generate children for each node. A node in a *binomial tree* has m children with probability q and has no children with probability $1 - q$, where m and q are parameters of the class of binomial trees. When $qm < 1$, this process generates a finite tree with expected size $\frac{1}{1-qm}$. Since all nodes follow the same distribution, the trees generated are self-similar and the distribution of tree sizes and depths follow a power law [14]. The variation of subtree sizes increases dramatically as qm approaches 1. This is the source of the tree's imbalance.

The nodes in a *geometric tree* have a branching factor that follows a geometric distribution with an expected value that depends on the depth of the node. In order to simplify our discussion, we focus here on geometric trees having a fixed branching factor, b. Another tree parameter is the value r of the root node. Multiple instances of a tree type can be generated by varying this parameter, hence providing a check on the validity of an implementation. A more complete description of tree generation is presented elsewhere [16].

2.2 MPI

The Message Passing Interface (MPI) is an industry standard message passing middleware created by the MPI Forum [11]. MPI defines a parallel programming model in which communication is accomplished through explicit two-sided messages. Under this model, data must be explicitly transmitted between processors using *Send()* and *Recv()* primitives.

2.3 Dynamic Load Balancing

The technique of achieving parallelism by redistributing the workload as a computation progresses is referred to as dynamic load balancing. In this work, we examine two different dynamic load balancing schemes: work sharing and work stealing. These two schemes were chosen because they represent nearly opposite points in the design space for dynamic load balancing algorithms. In particular, work stealing is an inherently distributed algorithm which is well suited for clusters whereas work sharing is inherently centralized and is best suited for shared memory systems.

Hierarchical schemes have also been proposed that offer scalable dynamic load balancing for distributed memory and wide-area systems. These schemes offer greater scalability and tolerance for high latency links. However, they are often constructed using work sharing or work stealing algorithms (e.g. Hierarchical Stealing [17]).

2.3.1 Work Sharing

Under the work sharing approach, processors balance the workload using a globally shared work queue. In UTS, this queue contains unexplored tree nodes. Work in the shared queue is grouped into units of transferable work called *chunks* and the *chunk size*, c, parameter defines the number of tree nodes contained within a chunk.

In order to perform depth-first search, each processor also maintains a depth-first stack containing the local collection of unexplored nodes, or *fringe*, of the tree search. When a processor has exhausted the work on its local stack, it gets another chunk of unexplored nodes from the shared queue. If no work is immediately available in the shared queue, the processor waits either for more work to become available or for all other processors to reach a consensus that the computation has ended. When a processor does have local work, it expands its local fringe and pushes the generated children onto its stack. If this stack grows to be larger than two chunks, the processor sends a chunk of its local work to the shared queue, allowing the surplus work to be performed by other processors that have become idle.

MPI-2 has been introduced to provide one-sided put/get semantics, however in the context of this work we specifically target the popular two-sided model of MPI-1.

2.3.2 Work Stealing

While work sharing uses global cooperation to facilitate load balancing, work stealing takes a distributed approach. Under work stealing, idle processors search among the other processors in the computation in order to find surplus work. In contrast to work sharing, this strategy places the burden of finding and moving tasks to idle processors on the idle processors themselves, minimizing the overhead to processors that are making progress. For this reason, work stealing is considered to be *stable* because no messages are sent when all processors are working [2]. In comparison, work sharing is *unstable* because it requires load balancing messages to be sent even when all processors have work.

3 Algorithms

3.1 Work Sharing in MPI

Because MPI provides the programmer with a distributed memory view of the computation, the most natural way to implement work sharing under MPI is with a work manager. This work manager's job is to maintain the shared work queue, to service work releases and work requests, and to detect termination. Because the parallel performance of our work sharing implementation depends greatly on the speed with which the manager is able to service requests, the manager does not participate in the computation.

In order to efficiently service requests from all processors in the computation, the work manager posts two non-blocking *MPI_Irecv()* descriptors for each worker in the computation: one for work releases and one for work requests. Work releases are distinguished from work requests by the message's tag. When a worker generates surplus work, it releases a chunk of work to the manager. When a worker requests a chunk of work from the queue manager it sends a work request message to the manager and blocks waiting for a response. If the manager is not able to immediately service the processor's request for work, it adds the processor to the idle queue and services the request once more work becomes available. If the manager detects that all processors have become idle and no work is available in the queue, it concludes that the computation has terminated and it sends all processors a termination message.

3.1.1 Out of Order Message Receipt in MPI

The MPI specification guarantees that between any two threads, the program order of blocking operations is observed [11]. However, in the presence of send buffering and non-blocking receive operations, this guarantee may mislead the incautious programmer into relying on an ordering that can be violated. We encountered this exact problem in our initial implementation which created a very hard-to-find race condition: occasionally the queue manager would lose a chunk of work, resulting in premature termination.

The cause of the race condition was that the manager was receiving messages out of order. A worker would release a chunk of work to the queue manager using a blocking send operation and quickly exhaust its local work, sending out a blocking work request to the queue manager. Both send operations would be buffered at the sender, immediately returning in the sender's context. The larger work release message would then be transmitted by the MPI runtime system using rendezvous protocol whereas the smaller work request message would be transmitted using eager protocol. Because of this, they would arrive at the work manager out of order and when the manager polled its receive descriptors it would see the work request before seeing the work release. If all other processors were in the idle queue at the time the last work request message was received, the queue manager would detect termination early, never having seen the incoming release message.

Rather than solve this problem by using unbuffered sends, we implemented a simple but effective timestamping scheme. Under this scheme, each worker keeps track of the number of chunks it has released to the shared queue and transmits this count along with each work request. The queue manager also maintains a count of the number of chunks it has received from each worker. When the manager attempts to detect termination it compares these counts and if they don't match, the manager knows that there are still outstanding messages in-flight and it continues to poll its receive descriptors.

3.2 Work Stealing in MPI

In general, stealing is a one-sided operation. However, due to MPI's two-sided communication model, processors that have exhausted their local work are unable to directly steal chunks of computation from other processors. Instead, idle processors must rely on cooperation from busy processors in order to obtain work. In order to facilitate this model for work stealing we created an explicit polling *progress engine*. A working processor must periodically invoke the progress engine in order to observe and service any incoming steal requests. The frequency with which a processor enters the progress engine has a significant impact on performance and has been parameterized as the *polling interval*, i.

If a processor has received a steal request at the time it calls into the progress engine, it checks to see if it has surplus work and attempts to satisfy the request. If enough work is available, a chunk of work is sent back to the thief (requesting) processor. Otherwise, the victim responds with a "no work" message and the thief moves on to its next

potential victim. Under this approach, processors with no work constantly search for work to become available until termination is detected. However, because each processor posts only a single receive descriptor for steal requests, the total number of steal requests serviced per polling interval is stable and is bounded by the number of processors in the computation.

3.2.1 Distributed Termination Detection

Our work stealing implementation uses a modified version of Dijkstra's well-known termination detection algorithm [8]. In this algorithm, a colored token is circulated around the processors in a ring in order to reach a consensus. In our implementation the token can be any of three colors: $white$, $black$, or red. Processor 0 owns the token and begins its circulation. Each processor passes the token along to its right as it becomes idle, coloring it $white$ if it has no work and has not given work to a processor to its left or $black$ if it has work or has given work to a processor to its left. In order to address the same out-of-order message receipt race condition encountered in the work sharing implementation, the token carries with it two counters: one counting the total number of chunks sent and another counting the total number of chunks received.

Whenever processor 0 receives the token it checks whether a consensus for termination has been reached. If the token is $white$ and both counters are equal then termination has been reached and processor 0 circulates a red token to inform the other processors. Otherwise, processor 0 colors the token $white$, resets the counters with its local counts and recirculates the token.

3.2.2 Finalizing MPI with Outstanding Messages

During the termination phase of the computation, all processors continue searching for work until they receive the red token. To avoid deadlock, steal requests and their corresponding steal listeners must be non-blocking. Because of this, any processor can have both outstanding *Send()* and *Recv()* operations when it receives the red token.

Many MPI implementations (e.g. MPICH, Cray MPI, LAM, etc...) will allow the user to simply discard these outstanding messages on termination via the collective *MPI_Finalize()*. However, the MPI specification states that a call to *MPI_Finalize()* should not complete in the presence of any such messages. Some MPI implementations, notably SGI's implementation, do honor these semantics. Under these runtime systems, any program that calls *MPI_Finalize()* without first cleaning up its outstanding messages will hang.

MPI does provide a way to cancel outstanding messages by calling *MPI_Cancel()*. However this function is not completely supported on all platforms. Notably, MPICH does not support canceling send operations so any code that relies on *MPI_Cancel()* will have limited portability. In addition to this, the specification states that for any non-blocking operation either *MPI_Cancel()* can succeed or *MPI_Test()* but not both. Therefore trying to cancel a message that has succeeded will result in a runtime error. However simply calling *MPI_Test()* once before calling *MPI_Cancel()* will introduce a race condition. Thus, it would seem that the MPI specification does not provide any safe mechanism for terminating a computation in the presence of outstanding messages!

Our solution to this problem was to introduce another stage to our termination detection algorithm that acts as a message fence. In this new stage we color the token $pink$ before coloring it red. When the pink token is circulated all processors cease to introduce new steal requests and update the token's message counters with counts of the number of steal messages sent and the number received. The pink token then circulates until all control messages have been accounted for (usually 1 or 2 circulations in practice). This is detected by processor 0 by comparing the token's counters to ensure that they are equal. Once they are, processor 0 colors the token red informing all nodes that communication has reached a consistent state and it is now safe to terminate.

3.3 Managing Load Balancing Overhead

We define the *overhead* of a dynamic load balancing scheme to be the amount of time that working processors must spend on operations to support dynamic load balancing. In the following sections, we describe polling-based solutions that allow us to reduce the overhead for each dynamic load balancing scheme by fine tuning the frequency of load balancing operations to better match particular systems and workloads.

3.3.1 Work Stealing

Overhead in our work stealing implementation is naturally isolated to the polling-based progress engine. Working processors must periodically invoke the progress engine to service any incoming steal requests. The frequency with which these calls are made is parameterized as the *polling interval*. If calls are not made frequently enough then steal requests may go unnoticed and the load may become imbalanced. However, if calls are made too frequently then performance will be lost due to the overhead of excess polling.

Algorithm 1 Work stealing polling interval

```
1: if Nodes_Processed % Polling_Interval = 0 then
2:     Progress_Engine()
3: end if
```

In the case of work stealing, we have experimentally observed that the optimal polling interval does not vary with the chunk size or the workload. Instead, the optimal polling interval is a fixed property of the combination of hardware and runtime systems.

3.3.2 Work Sharing

Overhead in the work sharing scheme is incurred when working processors must release a chunk of their work to the work manager. These communication operations are not initiated by a request for work, instead they must occur periodically in order to ensure the load remains balanced. For this reason, work sharing is *unstable*.

In order to fine tune the performance of our work sharing implementation, we have introduced the *release interval*, i, parameter. The release interval defines how frequently a release operation is permitted. Thus, in order for a working processor to release work to the work manager, the processor must now have enough work as well as sufficient elapsed time since its last release.

Algorithm 2 Work sharing release interval

```
1: if Have_Surplus_Work() and
   Nodes_Processed % Polling_Interval = 0 then
2:     Release_Work()
3: end if
```

The polling optimal interval for our work stealing scheme is a system property that does not vary with respect to chunk size and workload. However, under work sharing, the optimal release interval does vary with respect to these parameters. This is because each of these parameters controls different aspects of the load balancing overhead. Under work stealing the frequency with which working processors must perform load balancing (i.e. overhead) operations depends only on the frequency with which steal requests are generated. The frequency with which these requests are generated is influenced only by the workload and the load balance achieved using the chosen chunk size. Therefore, the polling interval does not directly affect the total volume of load balancing operations. Instead, the polling interval attempts to achieve better performance by trading latency in servicing load balancing requests for reduced overhead of checking for the these requests.

In contrast to this, the work sharing release interval attempts to directly inhibit the frequency with which working processors perform load balancing operations by allowing no more than one release per period. Thus, the overhead of our work sharing scheme is not only related to how frequently a processor generates surplus work, but also to how often it is permitted to release such work.

4 Experimental Evaluation

4.1 Experimental Framework

Our experiments were conducted on the Dell blade cluster at UNC. This system is configured with 3.6 GHz P4 Xeon nodes, each with 4GB of memory; the interconnection network is Infiniband; and the infiniband-optimized MVAPICH MPI environment [15] was used to run our experiments.

Our experimental data was collected for two unbalanced trees, each with approximately 4 million nodes. $T1$ corresponds to a geometric tree with a depth limit of 10 and a fixed branching factor of 4. $T3$ corresponds to a binomial tree with 2000 initial children, a branching factor of 8 and a branching probability of 0.124875. A significant difference between $T1$ and $T3$ is that $T3$ maintains a much larger fringe during the search, allowing it to be balanced using a larger chunk size.

4.2 Impact of Polling Interval on Stealing

Figure 1 shows the performance of our work stealing implementation over a range of polling intervals, for a 32-processor execution. From this figure, we can see that introducing the polling interval parameter allows us to improve performance by 40%-50% on these workloads. However, polling intervals that are too large can result in performance loss by increasing the steal response latency disproportionately to the polling overhead.

We can also see that the optimal polling interval for the stealing progress engine is roughly independent of both the chunk size and the workload. Because of this, on a given system the polling interval can be fixed and only the chunk size must be tuned to achieve optimal performance for a given workload. Based on the data collected here, we have chosen $i = 8$ as the polling interval for our test system.

4.3 Impact of Release Interval on Sharing

Figure 2 shows the performance of our work sharing implementation over a range of release intervals, also for a 32-processor execution. From these two graphs, we can see that tuning the release interval allows us to achieve over 2X performance improvement on $T1$, but very little improvement on $T3$. This is because the performance achievable on $T3$ is most dependent on the choice chunk size.

From this figure, we also observe that the optimal release interval and chunk size both vary with respect to a given workload and that the optimal chunksize also varies with respect to the release interval. While the best performance for $T3$ is achieved with the release interval $i = 32$ and chunk size $c = 50$, $T1$'s best performance is achieved

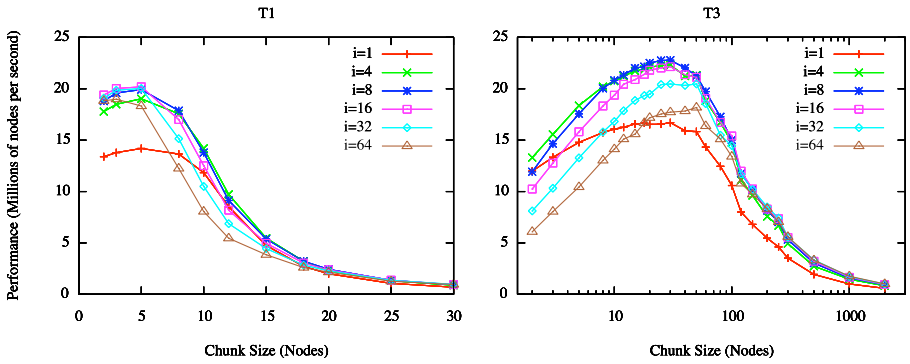


Figure 1. Impact of polling interval on MPI work stealing on Dell Blade cluster using 32 processors

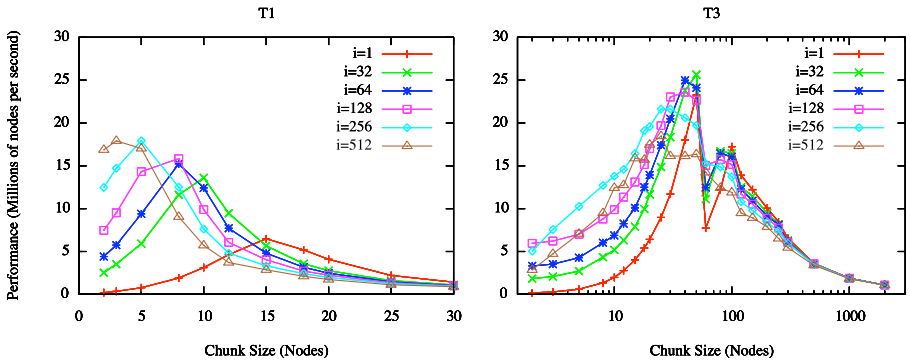


Figure 2. Impact of release interval on MPI work sharing on Dell Blade cluster using 32 processors

for $i = 256$, $c = 5$. However, from the data collected we can see that $i = 128$ is a reasonable compromise for both workloads and in order to draw a comparison between our two load balancing schemes we fix $i = 128$ for our system.

4.4 Performance Comparison

Figures 3 and 4 show the performance in millions of nodes per second for the work sharing and work stealing implementations on trees $T1$ and $T3$. We can immediately see that the overhead of maintaining a shared work queue is a significant impediment to performance in the work sharing implementation and that it leads to poor scaling and significant performance loss with more than 32 processors. In contrast to this, work stealing is more stable with respect to chunk size and is able to scale up to 64 processors.

By fixing the release and polling intervals, we are able to focus on the relationship between chunk size, workload, and performance. This means that under both dynamic load balancing schemes and for a given workload, the frequency of load balancing is inversely proportional to the chunk size. This is because any work in excess of two chunks is considered available for load balancing. Thus, very small chunk sizes lower the cutoff between local and surplus work, creating more opportunities for load balancing to occur. Likewise, very large chunk sizes increase the cutoff between local and shareable/stealable work, reducing the number of chances for performing load balancing. Because of this, performance is lost for small chunk sizes due to high load balancing overhead and performance is lost for very large chunk sizes as the inability to perform load balancing leads to poor work distribution.

This trend is especially apparent under work sharing where smaller chunk sizes increase the frequency of release operations, quickly overwhelming the work manager with load balancing requests. In comparison, under work stealing load balancing operations only occur in response to a processor exhausting its local work. Thus, work stealing is better able to facilitate the fine-grained load balancing required by $T1$ while work sharing struggles as communication with the work manager becomes a bottleneck.

For workloads such as $T3$ which can tolerate more coarse-grained load balancing, work sharing is able to achieve performance rivaling that of work stealing even though one of its processors does no work. This is because processors spend much less time idle as the queue manager is able to satisfy work requests more quickly than can be achieved under work stealing. However, this performance is

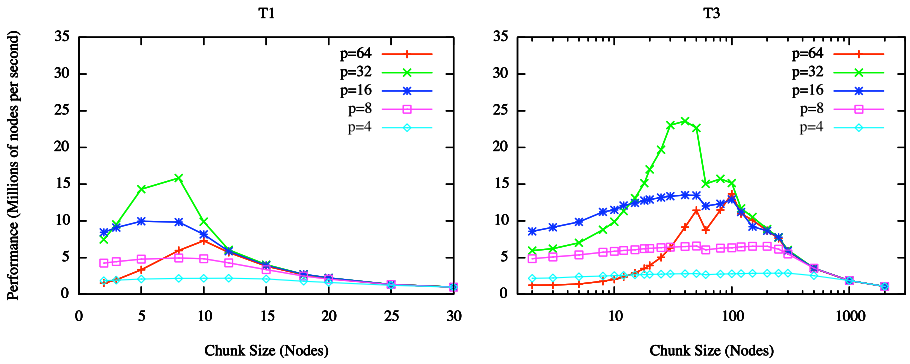


Figure 3. Performance of work sharing vs. chunk size ($i = 128$)

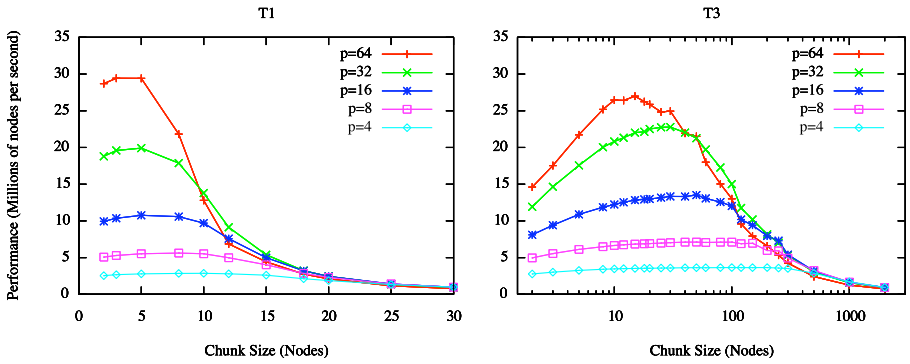


Figure 4. Performance of work stealing vs. chunk size ($i = 8$)

only available over a small range of chunk sizes due to the delicate balance between contention to communicate with the work manager and achieving an even work distribution.

On tree $T3$ we can also see that the work sharing implementation is very sensitive to the message latency. This is visible at chunk size 50 in Figure 3 where the larger chunk size has caused the MPI runtime to switch from eager to rendezvous protocol for work transfers. On this tree, we can also see that even though it is better suited for work sharing, we are unable to achieve scalability past 32 processors as the work manager's latency grows proportionately in the number of processors.

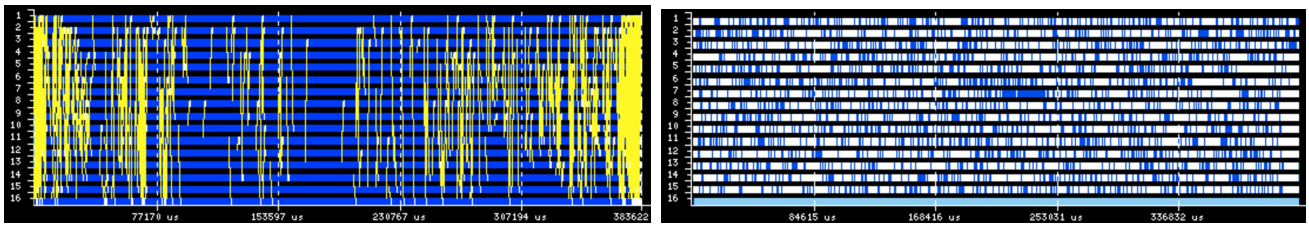
4.5 Load Balancing Visualization

Figure 5 shows Paraver [10] traces for 16 threads running UTS on tree $T1$ on the Dell Blade cluster. The dark blue segments of the trace represent time when a thread is working and the white segments represent time when a thread is searching for work. Under work stealing, A yellow line connects two threads together via a steal operation. Steal lines have been omitted from the work sharing trace in order to improve readability. Under work sharing, all load balancing operations happen with respect to the manager (processor 16) who performs no work. Therefore, any processor's transition from the idle state (white) to the working state (blue) must be the result of a work transfer from the manager to the idle node.

In figure 5(a), we can see that under work stealing execution is divided into three stages: initialization, steady-state, and termination. Steal activity is high as the fringe expands and collapses in the initialization and termination stages. However, for over 60% of the runtime the workload remains well distributed, leading to relatively low steal activity. Figure 5(b) shows a similar trace for work sharing on tree $T1$. From this trace, we can see that a great deal of time has been wasted on idle processors. This is because as each processor releases work to the manager, it releases its nodes closest to the root. Because the geometric tree is depth limited, this causes each processor to frequently give away the nodes that lead to the good load balance achieved under work stealing.

5 Related Work

Many different schemes have been proposed to dynamically balance the load in parallel tree traversals. A thorough analysis of load balancing strategies for parallel depth-



(a) Work Stealing: $p = 16, i = 8, c = 5$

(b) Work Sharing: $p = 16, i = 128, c = 5$

Figure 5. Paraver traces for tree T1

first search can be found in [12] and [13]. Work stealing strategies in particular have been investigated theoretically as well as in a number of experimental settings and have been shown to be optimal for a broad class of problems requiring dynamic load balance [5].

Dynamic Load Balancing under message passing has been explored in the context of different problems [1] [17] as well as in a more general context [4] [18]. Many of the challenges we encountered in the implementation of work stealing on top of MPI are similar to those encountered by the implementers of active messages over MPI [7], one-sided communication over MPI [9], and shared memory programming over the MPI conduit [6].

6 Conclusions

We have enhanced the UTS benchmark with MPI implementations that allow us to extend our characterization of the performance of unbalanced computations to commodity clusters and distributed memory systems. Taking our UTS implementation as an example, we have identified several challenges faced by implementers of irregular parallel algorithms under MPI. In particular, we explored support for dynamic load balancing and distributed termination detection using an explicit polling progress engine. We also explored tolerating out-of-order message receipt and provided a portable solution to finalizing the MPI runtime system in the presence of asynchronous messages.

By introducing additional parameters to explicitly manage communication overhead, we observed a speedup of up to 2X in parallel performance. We also observed that, while work sharing may achieve better performance on certain workloads, work stealing achieves comparable, if not better, performance and scalability across a wider array of chunk sizes and workloads.

References

[1] R. Batoukov and T. Srevik. A generic parallel branch and bound environment on a network of workstations, 1999.

[2] P. Berenbrink, T. Friedetzky, and L. Goldberg. The natural work-stealing algorithm is stable. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 178–187, 2001.

[3] K. Berlin, J. Huan, M. Jacob, G. Kochhar, J. Prins, W. Pugh, P. Sadayappan, J. Spacco, and C.-W. Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In L. Rauchwerger, editor, *LCPC*, volume 2958 of *Lecture Notes Comp. Sci.*, pages 194–208, 2003.

[4] M. Bhandarkar, L. Kale, E. de Sturler, and J. Hoeflinger. Adaptive load balancing for MPI programs. In *ICCS '01*.

[5] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th Ann. Symp. Found. Comp. Sci.*, pages 356–368, Nov. 1994.

[6] D. Bonachea and J. C. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets. In *SHPSEC '03*.

[7] Dan Bonachea. AMMPI: Active Messages over MPI, 2006.

[8] E. W. Dijkstra and C.S.Scholten. Termination detection for diffusing computations. *Inf. Proc. Letters*, 11(1):1–4, 1980.

[9] J. Dobbelaere and N. Chrisochoides. One-sided communication over MPI-1.

[10] European Center for Parallelism. PARAVER, 2006.

[11] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[12] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *J. Par. Dist. Comp.*, 22(1):60–79, 1994.

[13] V. Kumar and V. N. Rao. Parallel depth first search. part ii. analysis. *Int'l J. Par. Prog.*, 16(6):501–519, 1987.

[14] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD '05*.

[15] J. Liu, J. Wu, S. Kini, P. Wyckoff, and D. Panda. High performance RDMA-Based MPI implementation over InfiniBand, 2003.

[16] S. Olivier, J. Huan, J. L. J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. In *LCPC*, Nov. 2006.

[17] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP'01*.

[18] G. Weerasinghe. Asynchronous Communication in MPI. Master's thesis, University of Connecticut, Connecticut, USA, 1997.