Scalable Dynamic Load Balancing Using UPC

Stephen Olivier*, Jan Prins
Department of Computer Science
University of North Carolina at Chapel Hill
{olivier, prins}@cs.unc.edu

Abstract

An asynchronous work-stealing implementation of dynamic load balance is implemented using Unified Parallel C (UPC) and evaluated using the Unbalanced Tree Search (UTS) benchmark [1]. The UTS benchmark presents a synthetic tree-structured search space that is highly imbalanced. Parallel implementation of the search requires continuous dynamic load balancing to keep all processors engaged in the search. Our implementation achieves better scaling and parallel efficiency in both shared memory and distributed memory settings than previous efforts using UPC [1] and MPI [2]. We observe parallel efficiency of 80% using 1024 processors performing over 85,000 total load balancing operations per second continuously. The UPC programming model provides substantial simplifications in the expression of the asynchronous work stealing protocol compared with MPI. However, to obtain performance portability with UPC in both shared memory and distributed memory settings requires the careful use of one-sided reads and writes to minimize the impact of high latency communication. Additional protocol improvements are made to improve dissemination of available work and to decrease the cost of termination detection.

1 Introduction

Combinatorial optimization and enumeration are key techniques in computational science and knowledge discovery. These sorts of problems require exhaustive search of a state space of possibilities. When the state space is very large, as is often the case, a parallel search may be the only hope for a timely answer.

Parallel search of a state space combines a search strategy (e.g. depth-first search, branch-and-bound, iterative deepening) with a load balancing strategy [3, 4]. The state space often has unpredictable and irregular structure that can not be statically partitioned across processors, therefore *dynamic load balancing* techniques are required. Moreover, unlike many applications, there is no natural periodicity for load balancing, since it is impossible to predict when individual procesors will complete their portion of the search. Consequently efficient execution of parallel search requires *asynchronous* load balancing.

Parallel search and asynchronous dynamic load balancing are relatively easily implemented with reasonable performance in a low-latency hardware shared-memory setting. However, the most scalable HPC resources are clusters with memory distributed among nodes. Asynchronous dynamic load balancing is a challenge to implement in a scalable fashion in these settings because communication latencies are much higher and asynchronous parallel programs are difficult to express using MPI.

This paper examines the use of Unified Parallel C (UPC) to implement asynchronous and scalable dynamic load balancing. UPC provides a global address space that is partitioned to provide an explicit notion of locality. UPC programs can be compiled for either a shared memory or a distributed memory model. Our choice of UPC is motivated by the performance advantages of one-sided communication for asynchronous computation and by the simplified expression of asynchronous load balancing in UPC compared to MPI.

We use the Unbalanced Tree Search (UTS) benchmark [1] to measure efficiency and scalability of dynamic load balance. This benchmark requires the complete traversal of a large tree-structured search space. The size and imbalance of the search tree is parameterized. For this benchmark, any load balancing strategy can be used, but *work stealing* [5] is the most promising strategy for scalable load balancing and is the focus of this paper.

While UPC programs can be run in shared memory or in distributed memory settings, the performance implications of algorithmic choices can be dramatically different. The implementation presented in [1] performs well only in shared memory settings. In this paper, we describe a new work-stealing implementation intended to perform well in

*Stephen Olivier is supported by a National Defense Science and Engineering Graduate Fellowship.

distributed memory settings. The contributions are streamlined termination detection, rapid diffusion of work, and an asynchronous request-response protocol for work stealing that minimizes overheads to threads performing useful work. This last contribution was inspired by an MPI implementation of UTS [2], but exploits UPC's one-sided communication operations.

In our scaling experiments we use UTS parameters that yield trees with extreme variability in subtree size at every node in the tree. Using 1024 processors we generate and search a tree with approximately 157 billion nodes. More than 85,000 work stealing operations per second are performed on average throughout the search as processors run out of work and probe other processors for new parts of the search tree to explore. Our UPC implementation obtains the highest performance of UTS to date, achieving a search rate of 1.7 billion nodes per second using 1024 processors, corresponding to a speedup of 819 and an efficiency of 80%.

The paper is organized as follows. Section 2 presents the UTS benchmark and describes the work stealing paradigm for dynamic load balancing. Section 3 describes our UPC implementation of work stealing and contrasts it against previous UPC and MPI implementations. Section 4 provides a detailed performance analysis of implementations of the algorithms on distributed and shared memory architectures. Section 5 discusses related work. Section 6 presents a concluding discussion and plans for future work.

2 The Unbalanced Tree Search Problem

The UTS problem [1] is to count the nodes in an implicitly defined tree: any subtree in the tree can be generated completely from the description of its parent. The number of children of a node is a function of the node's description; in our current study a node can only have zero or two children. The description of each child is obtained by an evaluation of the SHA-1 cryptographic hash function [6] on the parent description and the child index. In this fashion, the UTS search trees are implicitly generated in the search process but nodes need only be retained while on the depth-first search stack.

Load balancing of UTS is particularly challenging since the distribution of subtree sizes is the same for all nodes in the search space but exhibits extreme variation. The distribution of subtree sizes consists of frequent small subtrees and occasionally enormous subtrees. The expected size of the search starting from any node is the same, so there is no advantage to be gained by stealing one node over another. In this sense the UTS search space is at least as challenging as any actual problem-specific search space, so solutions that work well with UTS are relevant to all search problems.

In a parallel search, each thread performs a depth-first traversal from some given node using its own stack of nodes and counting the nodes it visits. Initially, a single thread holds the root node. Upon completion, the total number of nodes traversed in each thread can be combined to yield the size of the complete tree. A thread that empties its stack becomes idle. Load balancing is performed by moving one or more node(s) from a non-empty stack of a working thread to the empty stack of an idle thread.

Work stealing can be an efficient approach to the load balancing task because it is initiated by idle threads. An idle thread tries to steal one or more nodes from some other thread's nonempty stack. Since steal operations do not require the active participation of the victims, we say that they are "one-sided." The victim, which is actively exploring the search tree, is not interrupted by the operation. This concept is described in [7] as the "work-first" principle.

One key question concerns the number of nodes that are moved between threads at a time. The larger this chunk size k, the lower the overhead to work stealing when amortized over the expected work in the exploration of the k nodes. This argues for a larger value of k. However, the likelihood that a depth first search of one of our trees has k nodes on the stack at any given time is proportional to $\frac{1}{k}$, hence it may be difficult to find large amounts of work to move. Thus, the value of k represents a tradeoff between load imbalance and communication costs.

The performance of UTS at different choices of chunk size is of primary interest to users of the benchmark. The range of chunk sizes for which an implementation achieves peak performance may be narrow or wide, and may shrink as more threads are used. This reflects the sensitivity of the machine to message size. In particular, the performance at low chunk size indicates the efficiency of sending small messages on the machine. Consequently, distributed memory systems that require coarse-grain communication to achieve high performance are particularly challenged by the UTS problem.

3 Work Stealing Implementation in UPC

UPC (Unified Parallel C) is a shared-memory programming model based on a version of C extended with global pointers and data distribution declarations for shared data [8]. The model can be compiled for shared memory or distributed memory execution. For execution on distributed memory, it is the compiler's responsibility to translate memory addresses and insert inter-processor communication. A distinguishing feature of UPC is that global pointers with affinity to one particular thread may be cast into local pointers for efficient local access by that same thread. One-sided communication is also supported implicitly in shared variable references and explicitly in the UPC run-time library via routines such as *upc_memput()* and *upc_memget()*.

In UTS, a collection of local and global state variables

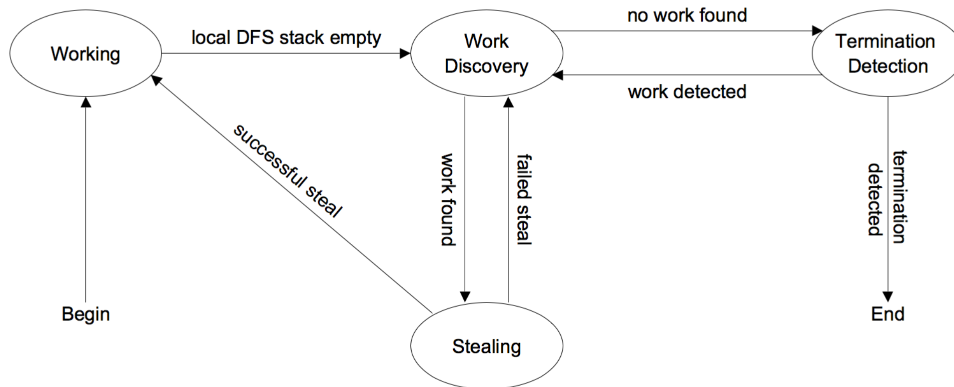

Figure 1. State diagram illustrating the basic operation of parallel threads.

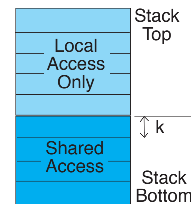

Figure 2. A thread's steal stack as implemented in the shared memory algorithm.

are maintained. Remote accesses of these variables, as well as the locks that guard them, are accomplished through shared variable references. This greatly simplifies coding in comparison to implementations using message passing paradigms such as MPI. Through UPC's shared memory abstraction, a clear correspondence between algorithm and implementation is maintained, unfettered by concerns over matching sends and receives. The UTS search problem is concerned purely with load balance, and hence uses a simple parallel depth-first exploration. For the implementation of more complex state evaluation functions and more sophisticated strategies such as branch-and-bound, UPC offers clear additional advantages.

Algorithms to accomplish load balancing for UTS differ in the details of (1) management of shared access to the local depth-first stack, (2) conditions for transition between states, and (3) termination detection. Each thread spends fruitful time on depth-first exploration of tree nodes from its stack, while stealing allows nodes to be added to it and taken from it. Each thread's execution can be modeled as a state machine. Figure 1 shows a broad representation of such a state machine; details vary based on the particular load balancing algorithm used. Termination detection is required to determine the point at which no work is left in the system and communicate that knowledge to all threads.

We shall first describe in detail an algorithm designed primarily on and for shared memory systems. A UPC implementation of this algorithm was evaluated in [1]. Performance and scaling were excellent on shared memory machines, but disappointing on distributed memory machines. We summarize the design of an existing distributed memory implementation using MPI [2]. We then describe a new algorithm designed for distributed memory using UPC. The two algorithms for UPC share much in common. However, the distributed memory algorithm addresses challenges to the "work-first" principle posed by high-latency interconnects. In particular, locks are avoided and remote accesses that may interfere with the progress of working threads are limited.

3.1 Shared Memory Algorithm

In the shared memory algorithm, the DFS stack is partitioned into two regions: local and shared. A representation of a thread's stack is shown in Figure 2. Each thread may perform push and pop operations at stack top in the local region without requiring locking operations. The shared region is subject to concurrent access, and thus operations must be serialized through a lock. Only remote accesses to a thread's stack incur shared address translation overheads in UPC, and nodes are transferred through one-sided communication. Steal operations are necessary to accomplish load balancing, but they unfortunately incur remote locking and data transfer costs. To amortize the manipulation overheads, nodes can only be moved in chunks of size k between the local and shared regions or between the shared regions of two different threads' stacks.

The basic stages of the algorithm are illustrated in Figure 1. It is useful to consider four major states for each thread:

- **Working:** While there is work on the local stack, threads continue to pop nodes, visit the nodes, push any children onto the stack, and move work between the local and shared regions of the stack. The *release()* operation moves a chunk of k nodes from the local to the shared region, when the local region has built up a comfortable stack depth (at least $2k$ in our implementation). The chunk then becomes eligible to be stolen. If the local stack becomes empty, a *reacquire()* operation is used to move nodes from the shared region back onto the local stack.

- **Work Discovery:** When no more work is available on a thread's own stack, a pseudo-random probe order is used to examine other threads' stacks for available work. Since these probes may introduce significant contention when many threads are looking for work, the count of available work on a stack is examined without locking. Hence a subsequent *steal()* operation may not succeed if in the interim the state has changed. In this case the probe proceeds to the next victim.
- **Work Stealing:** The *steal()* operation locks the stack of a potential victim and checks if a chunk is actually available to be stolen. If so, the chunk is reserved. Then the lock is released. The reserved chunk is transferred outside of the critical region to minimize the time that the stack is locked. Since the transfer is implemented in UPC using one-sided communication, the victim is not required to actively participate and can continue doing work during the transfer.
- **Termination Detection:** When a thread out of work is unable to find any available work in any other stack, it enters a barrier. A thread releasing work sets a global variable that releases idle threads waiting at the barrier, and they resume searching for work. When all threads have reached the barrier, the last thread to enter it sets a termination flag.

As shown in [1], this simple algorithm performs well on shared memory machines. On distributed memory machines, however, communication costs, locking costs, and contention for shared variables conspire to drive down performance. Consider the cost of termination detection, implemented using a cancelable barrier. All threads waiting at the barrier must spin remotely on termination/cancellation flags, requiring an arbitrary number of remote operations. After each $release()$ operation, the cancelable barrier is reset by the thread releasing work. This is a remote operation, and it delays a thread that might otherwise be doing useful work. Furthermore, barrier operations are performed under lock, adding significant remote locking costs. As another example, locking of the shared stack to acquire or release a chunk of work can delay a working thread despite a local stack lock. This is because multiple remote threads attempting to steal work from the working thread can keep the stack locked for a comparatively long time as a result of remote references with high latency.

3.2 Message Passing Algorithm

A work stealing algorithm using message passing is described in [2]. Stealing is performed using a message exchange. To initiate a steal, an idle thread sends a request to a potential victim. Working threads poll for requests at an interval set by a user-supplied parameter. Upon detecting a request, the victim sends a chunk of work (if available) back to the requesting thread. Termination detection is accomplished using Dijkstra's token-passing algorithm [9]. The MPI implementation of this work stealing algorithm is used in Section 4.2 as a point of comparison to the UPC implementations of our shared memory and distributed memory algorithms.

3.3 Distributed Memory Algorithm Using UPC

Our new distributed memory algorithm using UPC addresses key weaknesses in the shared memory algorithm to better adapt it to distributed memory machines. At the heart of the algorithm is careful consideration, not explicitly addressed in the shared memory algorithm, for the latency of the interconnect between the processors. The challenge is to limit remote operations, especially locking, and accelerate the work discovery process. The new algorithm features a streamlined termination detection mechanism, more rapid diffusion of available work, and no locking of the DFS stack. While MPI introduces many complications in the implementation of work stealing, it has a clear advantage in not using any remote locking operations.

3.3.1 Streamlining Termination Detection

The shared memory algorithm's termination detection strategy has the useful property that threads in the termination phase quickly return to the work discovery phase when work becomes available. However, the working threads may be significantly slowed by frequent barrier cancellations. An alternative strategy would be to have idle threads enter the termination phase only when they are nearly certain that no more work remains.

The distributed memory algorithm adopts the latter strategy. A probe of the potential victim's $work_avail$ variable returns distinct values for working threads with no surplus work and threads with no work at all. If after a complete cycle of probes, a thread searching for work finds that all other threads are out of work, only then does it enter the barrier. If it finds even a single thread still working, it continues searching for work and does not enter the barrier.

All threads may appear to be out of work if the locus of remaining work moves out of phase with detecting probes. Probing threads, observing an apparent out-of-work situation, may enter the barrier while there is still work in the system. To accommodate this situation, which in practice seems to occur extremely rarely, threads that have entered the barrier continue to probe for work and leave the barrier if they successfully steal some work. However, each thread that has entered the barrier only inspects one other thread to

avoid overwhelming the remaining working threads, if any. Once all threads are in the barrier, the last thread to enter the barrier launches a tree-based termination announcement.

The advantage of the new algorithm's termination detection over that of the shared memory algorithm is that the expensive barrier operations are performed, almost always, only once. Thus, remote communication costs, locking costs, and contention for the shared variables used to implement the barrier are minimized.

3.3.2 Rapid Diffusion

In our shared memory algorithm, thieves steal one chunk at a time, which may be helpful for workers to accommodate very many thieves. However, we observed that this limitation can be counterproductive given the unbalanced nature of UTS. When left undisturbed for some time and while exploring a large subtree, a thread may generate a large number of unexplored nodes on its DFS stack. Idle threads steal work one chunk at time from these "work source" threads, often completing that chunk's work without generating many new nodes. Thus, they return quickly for another chunk. Each steal in this continued succession of steals requires costly remote operations.

Allowing a thread to steal more than one chunk at time, when available, minimizes the overhead costs incurred in a situation like the one just described. Of course, the transfer sizes per steal are larger, but recall that the one-sided operations of UPC allow the transfer to proceed while the victim continues working. Also, larger transfers may more fully utilize the interconnect bandwidth, mitigating the increase in message size.

In the distributed memory algorithm, the thief thread steals half the available chunks on the victim's steal stack if more than one chunk is available, or one chunk otherwise. In addition to the benefits outlined above, the effect is to rapidly increase the number of "work sources"–threads with work available. Each thread that steals a large number of chunks becomes itself a viable victim to other threads. The addition of more work sources decreases the number of probes required to find a victim and reduces contention for access to the work sources' stacks, hence leads to more rapid diffusion of work.

3.3.3 Lock-less DFS Stack

In an effort to place the burden of load balancing work on the idle threads, our shared memory algorithm had them perform the steals entirely on their own. However, the steal operations still disturb the working threads because they are forced to wait for the shared stack to be unlocked to release or reacquire nodes. The cost of the interfering remote locking operations is typically an order of magnitude greater than the cost of a shared variable reference. In contrast, we observed that a primary advantage of the MPI implementation is its avoidance of locking on the steal stack. In that implementation, working threads poll at intervals to observe and service steal requests by sending the work. Thus, there are no concurrent accesses to the stack. The distributed memory algorithm for UPC uses a similar mechanism for the stealing process.

In the distributed memory UPC implementation, a thread searching for work, upon detecting work available at a remote thread, attempts to write its thread ID into a lock-protected request variable at the potential victim. If the thief successfully writes its ID, it waits for the victim thread to respond by indicating the amount of work given and the address of the work on the victim's stack. Note that if the request is denied, the indicated amount would be zero, prompting the thief to continue probing other threads. Otherwise, the thief transfers the work onto its stack in a one-sided get operation.

Though the working thread is now responsible for polling for steal requests, the costs are minimal since it only involves a read of a local variable without locking. If a request is pending, two remote writes are required to service it, and a local write resets the request variable.

From the working thread's point of view, it has complete control of its own stack. Since it alone assigns work to be stolen, no locking of the stack is required. That property is key because it eliminates any waiting for a remote thread to finish an operation under lock, a cost incurred in the original implementation by having the thief make the work reservation.

4 Performance Evaluation

To establish a baseline for performance, we shall first present performance results from program execution on a single processor. Performance is measured as the number of tree nodes processed per second of running time. Absolute performance and speedup are reported for program execution on both distributed memory and shared memory machines. During parallel execution, each UPC thread is allocated a dedicated processor.

4.1 Sequential Performance

Experiments were carried out on two Dell blade clusters, Topsail and Kitty Hawk. Each node of the Topsail cluster has two 2.33 GHz quad-core Intel Xeon E5345 processors with 2x4MB L2 cache. Each node of Kitty Hawk has two 2.66 GHz dual-core Intel Xeon E5150 processors with 4MB shared L2 cache. Topsail consists of 520 compute nodes (4160 processors), while Kitty Hawk has 66 compute nodes (264 processors). We used the Intel icc 9.0 compiler with *-O3* for sequential compilation on the Xeon processors.

Label	Explanation	Details
upc-distmem	UPC implementation of the distributed memory algorithm (upc-term-rapdif with lock-less DFS stack)	Sect. 3.3.3
upc-term-rapdif	upc-term with rapid diffusion	Sect. 3.3.2
upc-term	upc-sharedmem with streamlined termination detection	Sect. 3.3.1
upc-sharedmem	UPC implementation of the shared memory algorithm	Sect. 3.1
mpi-ws	MPI work stealing implementation	Sect. 3.2, [2]

Figure 3. Legend of labels used in speedup and performance graphs.

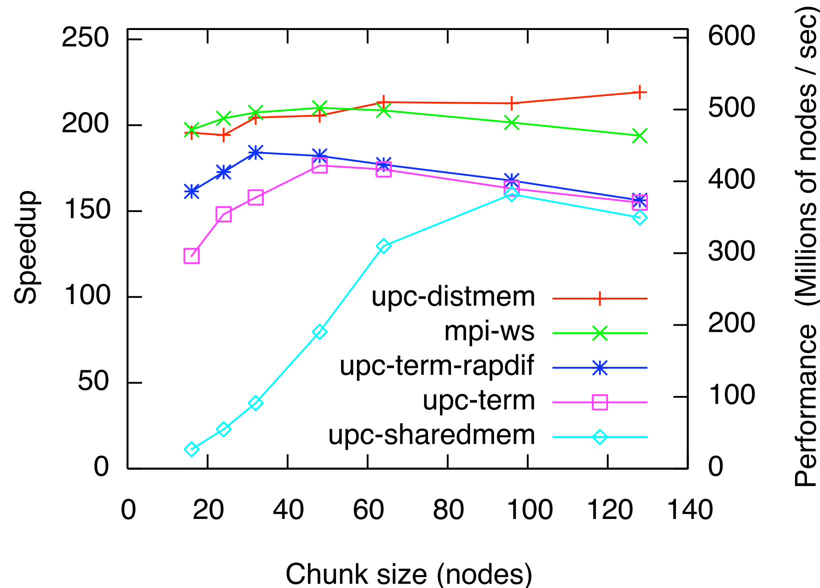

Figure 4. Speedup and absolute performance at different chunk sizes using 256 threads on the Kitty Hawk cluster, illustrating improvements made to the UPC implementation and the importance of work stealing granularity. See Figure 3 above for an explanation of the labels used in this graph.

The primary sample problem used for our performance evaluation is a highly unbalanced tree of about 10.6 billion nodes. The root node has 2000 children; all other nodes have either two children or none at all.[1] Over 99.9% of the work is contained just one of the 2000 subtrees below the root. Such extreme variation in subtree size occurs throughout the tree, rendering methods such as work splitting ineffective. For experiments using a larger number of processors, we use a similar tree of size 157 billion nodes[2].

One core of the E5345 in Topsail achieves an exploration rate of 2.10 million nodes/sec on the sample problem. One core of the E5150 in Kitty Hawk achieves a rate of 2.39 million nodes/sec on this problem. The sequential rate of depth-first search primarily reflects the speed at which the processor can calculate SHA-1 hash evaluations.

[1] For reproducibility, we give the exact UTS parameters here: The tree is generated using the binomial distribution with a random seed $r = 0$. Nodes below the root have $m = 2$ children with probability $q = \frac{1}{2}(1 - 10^{-8})$, and no children with probability $1 - q$. As stated above, the root has $b = 2000$ children.

[2] Parameter settings: binomial distribution, $r = 559, m = 2, b = 2000, q = \frac{1}{2}(1 - 10^{-6})$

4.2 Parallel Performance on Distributed Memory

Within both Topsail and Kitty Hawk, compute nodes are connected using the Infiniband high-speed interconnect. Each node has a distinct memory, so the shared memory abstraction in UPC is supported by translating remote memory references into remote reference API calls. On both Kitty Hawk and Topsail, the Berkeley UPC run time has been built directly upon Infiniband network driver APIs, respectively VAPI and OFED for the two machines.

On both Topsail and Kitty Hawk, performance comparisons are made with the existing MPI implementation of

UTS using work stealing [2]. Optimal parameters for communication tuning (e.g. polling intervals) were used in the results presented here. For MPI experiments on Topsail, MVAPICH is used; on Kitty Hawk, MVAPICH2 is used.

Figure 4 compares the speedup and performance of the UPC implementations of the shared memory and distributed memory algorithms with MPI work stealing implementation. Figure 3 gives an explanation of the labels used in this graph and the graphs that follow, also pointing to the sections of the paper or references providing details on each algorithm or implementation. Since the UPC run time is built directly over the native VAPI drivers, one-sided communications for work transfers in the UPC implementation are fully and efficiently supported. The distributed memory UPC implementation performs slightly better than the MPI work stealing implementation, while the performance of the shared memory UPC implementation lags far behind.

The performance of the UPC distributed memory implementation is particularly encouraging given that MPI library implementation is highly tuned compared to the UPC run time. Note that each of the refinements presented in Sections 3.3.1-3.3.3 shows an improvement in these results; the total improvement is about 37%.

4.2.1 Work Stealing Granularity

The granularity of work stealing often has a significant impact on the performance of the program. When chunk size is very small, many load balancing operations are performed, resulting in large overhead to working threads. When chunk size is very large, too few load balancing operations are performed, resulting in large idle times for some threads despite the availability of surplus work. Between these two extremes, there is a "sweet spot," a range of chunk sizes that results in the best performance. This sweet spot resembles a plateau from which performance falls off rapidly on both sides. As more processors are used, performance is more sensitive to chunk size.

Figure 4 illustrates the effect of chunk size on speedup and performance for the various implementations. Note that the shared memory UPC version suffers extreme performance degradation at low chunk sizes. The implementation of the shared memory UPC algorithm behaves particularly poorly at low chunk sizes because many more cancelable barrier operations, involving shared variable references and lock operations, are done to support the resulting high number of releases and steals.

4.2.2 Further Scaling

On the larger cluster, Topsail, the final implementation shows solid performance on at least 1024 processors. As shown in Figure 5, the best UPC implementation processes

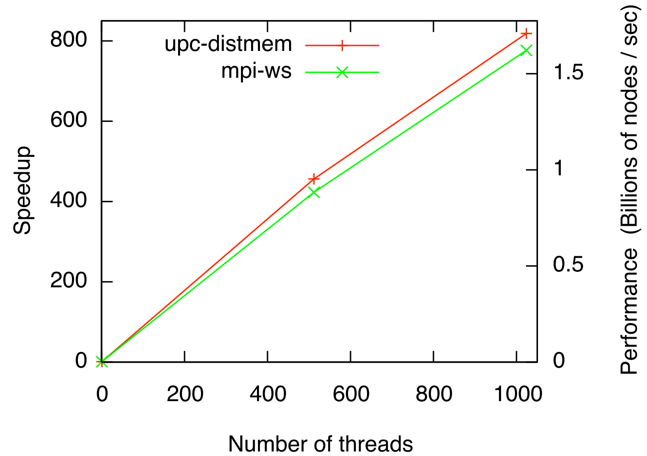

Figure 5. Speedup and absolute performance for a 157 billion node tree on Topsail.

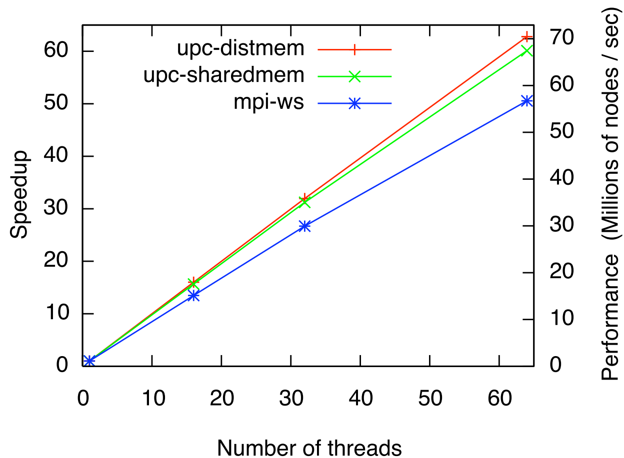

Figure 6. Speedup and absolute performance on the SGI Altix 3700.

a 157 billion node tree at a rate of over 1.7 billion nodes/sec, a speedup of 819.

4.3 Parallel Performance on Shared Memory

To evaluate performance portability on shared memory, we tested the UPC implementations of the shared memory and distributed memory algorithms on an SGI Altix 3700. Each processor of the Altix is a 1.6 GHz Intel Itanium2 with 256kB of L2 cache and 6MB of L3 cache. We used the GNU Intrepid UPC compiler 4.0.3 with *-O3* for UPC. The sequential version of UTS, compiled with gcc 4.0.3 *-O3*, processes 1.12M nodes/sec. Results are close for both

UPC implementations: near-linear speedup on up to at least 64 processors. The machine's low latency hypercube interconnect efficiently supports UPC shared variable accesses. However, the performance of the MPI implementation lags slightly behind the UPC implementations on this platform due to poor cache behavior and MPI overheads.

5 Related Work

Seminal work in load balancing techniques for parallel search includes [3] and [4]. Much of the other literature in parallel search is problem-dependent or search-strategy dependent, identifying particular aspects of the problem or search that can be used to ameliorate load balancing problems.

Analysis of work stealing with random probes is found in [7] and [10]. Many implementations of work stealing, such as the CILK language run time [11], consider shared-memory settings that are more forgiving of solutions with poor latency tolerance. Intel's Thread Building Blocks (TBB) use work stealing for efficient load balancing between threads on multi-core processors [12]. ATLAS [13] and Satin [14] use hierarchical work stealing for clusters and grids.

A work stealing method incorporating both load balancing and load distribution for data locality is considered in [15]. To evaluate their algorithms, the authors present a tool to generate task graphs with parameterized degree of parallelism, but not parameterized imbalance as in UTS. Randomized load balancing using work pushing for tree structured computation is explored in [16].

Many experimental evaluations of these load balancing schemes have been limited to small processor counts or older machines, and on problems of varying imbalance. Our load balancing strategies in UTS have been shown to scale well even on distributed memory machines on extremely unbalanced workloads.

6 Discussion

In this section we reflect on our work in a broader context. First we discuss our experience using UPC as the language for implementation. Then we consider the progress of our approach toward our stated goal–to demonstrate scalable dynamic load balancing.

6.1 UPC Programmability and Efficiency

Compared with MPI, the UPC model of compiler-generated communication with an explicit model of locality presents substantial simplifications for portable parallel programming.

In our UPC implementations, the algorithms are expressed clearly in the code, thanks to UPC's shared memory abstraction. They could be easily augmented to use more complex search methods such as branch-and-bound and backtracking as needed in different applications.

Early experience with UPC model found that it encouraged shared-memory programming techniques that carried extremely high cost in distributed memory settings [17]. However, UPC implementations have improved as a result of better one-sided communication support from processor interconnects. When combined with a more accurate model of communication cost, it is possible to write UPC programs that have high performance in both shared and distributed memory settings. The implementation of our asynchronous work stealing protocol is an example.

Nevertheless we still find that UPC performance can have anomalous dependence on the underlying hardware and runtime system. For example, using the Berkeley UPC compiler we need in some places to insert calls on the communication progress engine via *bupc_poll()*. This engine is needed when the underlying communication uses active messages rather than hardware one-sided communication, and this varies with details of the drivers and hardware. In order to be more widely adopted, UPC must further move beyond portability to performance portability.

6.2 Performance and Scalability

In the end, we were able to achieve excellent performance and scalability using a single UPC program that is portable across multiple machines and scales to high processor counts. As UPC compilers, run-time systems, and hardware support improve, we expect the development of applications using efficient low-level protocols to become simpler and more predictable, yielding better performance portability across platforms.

We optimized our application using the "work-first" principle that minimizes interference to threads making active progress. We observe 93% efficiency of threads *in the working state* compared to a single thread running optimized sequential UTS. The other 7% reflects not only time spent servicing steal requests, but also cold cache misses after stealing and the native C compiler's failure to fully optimize translated UPC source code.

Outside the working state, overhead time is spent searching for work, stealing work, or in termination detection. We could further improve the performance of our work stealing algorithm by decreasing this time. One way we may decrease the latency of probing for work and stealing in large clusters of shared memory multiprocessor nodes is to first try to steal work within a cluster node before probing off-node. Such an implementation could use the *bupc_thread_distance()* function in Berkeley UPC to dis-

cover which threads are located on the same node.

In conclusion, our efforts demonstrate that UPC is viable and in some cases clearly superior for complex parallel programming problems. For the important class of exhaustive search problems, we have achieved performance portability through careful consideration of communication costs in distributed memory architectures. Our implementation achieves strong performance and scalability while leveraging the UPC shared memory abstraction.

7 Acknowledgment

The authors thank the Renaissance Computing Institute for the use of the Kitty Hawk cluster and the University of North Carolina for the use of the Topsail cluster and the SGI Altix.

References

[1] Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An unbalanced tree search benchmark. In Almási, G., Cascaval, C., Wu, P., eds.: Proc. LCPC 2006. Volume 4382 of LNCS., Springer (2007) 235–250

[2] Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., Tseng, C.W.: Dynamic load balancing of unbalanced computations using message passing. In: Proc. 6th Intl. Workshop Perf. Mod., Eval., Opt. Par. Dist. Sys. (PMEO-PDS'07) / IPDPS'07., IEEE (2007)

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

[4] Grama, A., Kumar, V.: State of the art in parallel search techniques for discrete optimization problems. IEEE Trans. Knowledge Data Engr. **11**(1) (1999) 28–35

[5] Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments. In: Measurement and Modeling of Computer Systems. (1998) 266–267

[6] Eastlake, D., Jones, P.: US secure hash algorithm 1 (SHA-1). RFC 3174, Internet Engineering Task Force (2001)

[7] Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proc. 35th Ann. Symp. Found. Comp. Sci. (1994) 356–368

[8] UPC Consortium: UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab (2005)

[9] Dijkstra, E.W., W.H.J.Feijen, van Gasteren, A.: Derivation of a termination detection algorithm for distributed computations. Inf. Proc. Letters **16** (1983) 217–219

[10] Sanders, P.: A detailed analysis of random polling dynamic load balancing. In: Intl. Symp. Par. Arch., Algs., Nets.(ISPAN '94). (1994)

[11] Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proc. 1998 SIGPLAN Conf. Prog. Lang. Design Impl. (PLDI '98). (1998) 212–223

[12] Kukanov, A., Voss, M.: The foundations for scalable multi-core software in intel threading building blocks. Intel Technology Journal **11**(4) (2007)

[13] Baldeschwieler, J.E., Blumofe, R.D., Brewer, E.A.: Atlas: an infrastructure for global computing. In: EW 7: Proc. 7th ACM SIGOPS European workshop, NY, NY, ACM (1996) 165–172

[14] van Nieuwpoort, R., Kielmann, T., Bal, H.E.: Satin: Efficient parallel divide-and-conquer in java. In: Euro-Par '00: Proc. 6th Intl. Euro-Par Conf. Par. Proc., London, UK, Springer-Verlag (2000) 690–699

[15] Berger, E., Browne, J.: Scalable load distribution and load balancing for dynamic parallel programs. In: WCBC 99. (1999)

[16] Chakrabarti, S., Yelick, K.: Randomized load-balancing for tree-structured computation. In: IEEE Scalable High Performance Computing Conf. (1994) 666–673

[17] Berlin, K., Huan, J., Jacob, M., Kochhar, G., Prins, J., Pugh, W., Sadayappan, P., Spacco, J., Tseng, C.W.: Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In Rauchwerger, L., ed.: Proc. LCPC 2003. Volume 2958 of LNCS. (2003) 194–208