# UTS: An Unbalanced Tree Search Benchmark*

Stephen Olivier[1], Jun Huan[1], Jinze Liu[1], Jan Prins[1], James Dinan[2], P. Sadayappan[2], and Chau-Wen Tseng[3]

[1] Dept. of Computer Science, Univ. of North Carolina at Chapel Hill
{olivier, huan, liuj, prins}@cs.unc.edu
[2] Dept. of Computer Science and Engineering, The Ohio State Univ.
{dinan, saday}@cse.ohio-state.edu
[3] Dept. of Computer Science, Univ. of Maryland at College Park
tseng@cs.umd.edu

**Abstract.** This paper presents an unbalanced tree search (UTS) benchmark designed to evaluate the performance and ease of programming for parallel applications requiring dynamic load balancing. We describe algorithms for building a variety of unbalanced search trees to simulate different forms of load imbalance. We created versions of UTS in two parallel languages, OpenMP and Unified Parallel C (UPC), using work stealing as the mechanism for reducing load imbalance. We benchmarked the performance of UTS on various parallel architectures, including shared-memory systems and PC clusters. We found it simple to implement UTS in both UPC and OpenMP, due to UPC's shared-memory abstractions. Results show that both UPC and OpenMP can support efficient dynamic load balancing on shared-memory architectures. However, UPC cannot alleviate the underlying communication costs of distributed-memory systems. Since dynamic load balancing requires intensive communication, performance portability remains difficult for applications such as UTS and performance degrades on PC clusters. By varying key work stealing parameters, we expose important tradeoffs between the granularity of load balance, the degree of parallelism, and communication costs.

## 1 Introduction

From multicore microprocessors to large clusters of powerful yet inexpensive PCs, parallelism is becoming increasingly available. In turn, exploiting the power of parallel processing is becoming essential to solving many computationally challenging problems. However, the wide variety of parallel programming paradigms (e.g., OpenMP, MPI, UPC) and parallel architectures (e.g., SMPs, PC clusters, IBM BlueGene) make choosing an appropriate parallelization approach difficult. Benchmark suites for high-performance computing (e.g., SPECfp, NAS, SPLASH, SPEComp) are thus important in providing users a way to evaluate how various computations perform using a particular combination of programming paradigm, system software, and hardware architecture.

* This work was supported by the U.S. Department of Defense.

G. Almási, C. Caşcaval, and P. Wu (Eds.): LCPC 2006, LNCS 4382, pp. 235–250, 2007.

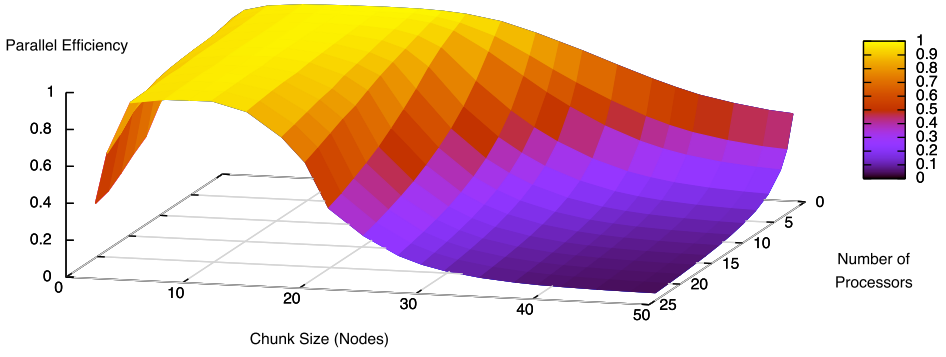


**Fig. 1.** Tree T1 has about 4.1 million nodes, and a depth of 10. The degree of each node varies according to a geometric distribution with mean 4. The parallel efficiency of the UPC implementation using 2 to 24 processors of an SGI Origin 2000 is shown relative to the sequential traversal of the tree. Work is stolen in chunks of $k$ nodes at a time from the depth-first stack of processors actively exploring a portion of the tree. When $k$ is small, the communication and synchronization overheads in work stealing start to dominate performance and lower the efficiency. As $k$ increases, it becomes increasingly difficult to find processors with $k$ nodes on their stack (the expected maximum number of nodes on the stack is 40), hence efficiency suffers from lack of load balance.

One class of applications not well represented in existing high-performance benchmarking suites are ones that require substantial dynamic load balance. Most benchmarking suites either exhibit balanced parallel computations with regular data access (e.g. solvers communicating boundary data such as SWM, TOMCATV, APPSP), or balanced computations with irregular data access and communication patterns (e.g. sparse solvers, FFT, Integer Sort, APPLU). Existing benchmarks that do utilize dynamic load balancing do so only at specific points in the application (e.g. Barnes-Hut n-body simulation, or solvers using adaptive mesh refinement).

We introduce a simple problem – parallel exploration of an unbalanced tree – as a means to study the expression and performance of applications requiring continuous dynamic load balance. Applications that fit in this category include many search and optimization problems that must enumerate a large state space of unknown or unpredictable structure.

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.

In this paper we describe OpenMP and UPC implementations of the UTS problem and evaluate their performance on a number of parallel architectures and for a number of different tree shapes and sizes. The implementations use a work-stealing strategy for dynamic load balancing. Threads traverse the portion of the tree residing on their local stack depth first, while stealing between threads adds a breadth first dimension to the search. Figure 1 illustrates the performance of the UPC implementation on a shared memory machine as a function of number of processors and the granularity of load balancing. We are particularly interested in how well a single UPC implementation performs across shared and distributed memory architectures, since UPC is a locality-aware parallel programming paradigm that is intended to extend to both memory models.

The remainder of the paper is organized as follows. Section 2 describes the class of unbalanced trees, and some of their properties. Section 3 describes the benchmark programs that were developed to implement parallel unbalanced tree search. Section 4 provides a detailed performance analysis of the implementations on a variety of shared and distributed memory architectures, comparing absolute performance, processor and work scaling. Section 5 has the conclusions.

## 2 Generating Unbalanced Trees

We describe a class of synthetic search trees whose shape, size, and imbalance are controlled through a small number of parameters. The class includes trees representing characteristics of various parallel unbalanced search applications.

The trees are generated using a Galton-Watson process [1], in which the number of children of a node is a random variable with a given distribution. To create deterministic results, each node is described by a 20-byte descriptor. The child node descriptor is obtained by application of the SHA-1 cryptographic hash [2] on the pair (parent descriptor, child index). The node descriptor also is the random variable used to determine the number of children of the node. Consequently the work in generating a tree with $n$ nodes is $n$ SHA-1 evaluations.

To count the total number of nodes in a tree requires all nodes to be generated; a shortcut is unlikely as it requires the ability to predict a digest's value from an input without executing the SHA-1 algorithm. Success on this task would call into question the cryptographic utility of SHA-1. Carefully validated implementations of SHA-1 exist which ensure that identical trees are generated from the same parameters on different architectures.

The overall shape of the tree is determined by the *tree type*. Each of these generates the children of its nodes based on a different probability distribution: binomial or geometric. One of the parameters of a tree 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.

We examined a variety of tree shapes and choose to report on two representative shapes due to space limits.

### 2.1 Binomial Trees

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 [3]. The variation of subtree sizes increases dramatically as $qm$ approaches 1. This is the source of the tree's imbalance. A binomial tree is an optimal adversary for load balancing strategies, since there is no advantage to be gained by choosing to move one node over another for load balance: the expected work at all nodes is identical.

The root-specific branching factor $b_0$ can be set sufficiently high to generate an interesting variety of subtree sizes below the root according to the power law. Alternatively, $b_0$ can be set to 1, and a specific value of $r$ chosen to generate a tree of a desired size and imbalance.

### 2.2 Geometric Trees

The nodes in a *geometric tree* have a branching factor that follows a geometric distribution with an expected value that is specified by the parameter $b_0 > 1$. Since the geometric distribution has a long tail, some nodes will have significantly more than $b_0$ children, yielding unbalanced trees. The parameter $d$ specifies the maximum, beyond which the tree is not allowed to grow. Unlike binomial trees, the expected size of the subtree rooted at a node increases with proximity to the root.

Geometric trees have a depth of at most $d$, and have an $O((b_0)^d)$ expected size. The depth-first traversal of a geometric tree resembles a stage of an iterative deepening depth-first search, a common search technique for potentially intractable search spaces.

## 3 Implementation

Our implementations perform a depth-first traversal of an implicit tree as described in the previous section. Since there is no need to retain a description of a node once its children have been generated, a depth-first stack can be used. A node is explored by popping it off the stack and pushing its children onto the stack. Parallel traversals can proceed by moving one or more node(s) from a non-empty stack of one processor to the empty stack of an idle processor.

Several strategies have been proposed to dynamically balance load in such a parallel traversal. Of these, work-stealing strategies place the burden of finding and moving tasks to idle processors on the idle processors themselves, minimizing the overhead to processors that are making progress. Work-stealing strategies have been investigated theoretically and in a number of experimental settings, and have been shown to be optimal for a broad class of problems requiring dynamic load balance [4]. Work-sharing strategies place the burden of moving work to idle processors on the busy processors. A thorough analysis of load balancing strategies for parallel depth-first search can be found in [5] and [6].

One key question concerns the number of nodes that are moved between processors at a time. The larger this chunk size $k$, the lower the overhead to both work stealing and work sharing strategies 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 a given time is proportional to $\frac{1}{k}$, hence it may be difficult to find large amounts of work to move. Indeed it was our goal to construct a problem that challenges all load balancing strategies, since such a benchmark can be used to assess some key characteristics of the implementation language, runtime environment, and computing system. For example, distributed-memory systems that require coarse-grain communication to achieve high performance may be fundamentally disadvantaged on this problem.

### 3.1 Work Stealing in UPC and OpenMP

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 [7]. The model can be compiled for shared memory or distributed memory execution. For execution on distributed memory, it is the compilers 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 thread. Explicit one-sided communication is also supported in the UPC run-time library via routines such as *upc_memput()* and *upc_memget()*.

We implemented work stealing in OpenMP for shared memory machines and UPC for shared and distributed memory machines. Instead of trying to steal procedure continuations as would be done in Cilk [8], which requires cooperation from the compiler, in our implementation idle threads steal nodes from another thread's depth-first stack. Initially, the first thread holds the root node. As enough nodes are generated from the root and its descendants, other threads steal chunks of nodes to add to their stacks. Each thread performs a depth-first traversal of some part of the tree using its own stack of nodes. A thread that empties its stack tries to steal one or more nodes from some other thread's nonempty stack. On completion, the total number of nodes traversed in each thread can be combined to yield the size of the complete tree.

### 3.2 Manipulating the Steal Stack

We now consider the design of the stack. In addition to the usual push and pop operations, the stack must also support concurrent stealing operations performed by other threads, which requires the stacks to be allocated in the shared address space and that locks be used to synchronize accesses. We must eliminate overheads to the depth-first traversal performed by a working thread as much as possible. Thus each thread must be able to perform push and pop operations at stack top without incurring shared address translation overheads in UPC or requiring locking operations. Figure 2 shows the stack partitioned into two regions.

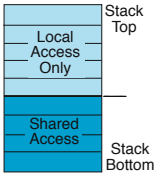


**Fig. 2.** A thread's steal stack

**Table 1.** Sample trees, with parameters and their resulting depths and sizes in millions of nodes

| Tree | Type | $b_0$ | $d$ | $q$ | $m$ | $r$ | Depth | MNodes |
|---|---|---|---|---|---|---|---|---|
| T1 | Geometric | 4 | 10 | – | – | 19 | 10 | 4.130 |
| T2 | Geometric | 1.014 | 508 | – | – | 0 | 508 | 4.120 |
| T3 | Binomial | 2000 | – | 0.124875 | 8 | 42 | 1572 | 4.113 |

The region that includes the stack top can be accessed directly by the thread with affinity to the stack using a local pointer. The remaining area is subject to concurrent access and operations must be serialized through a lock. To amortize the manipulation overheads, nodes can only move in chunks of size $k$ between the local and shared regions or between shared areas in different stacks.

Using this data structure, the local push operation, for example, does not involve any shared address references in UPC or any lock operations. The *release* operation can be used to move 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. A matching *acquire* operation is used to move nodes from the shared region back onto the local stack when the local stack becomes empty.

When there are no more chunks to reacquire locally, the thread must find and steal work from another thread. A pseudo-random probe order is used to examine other 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 performed under lock may not succeed if in the interim the state has changed. In this case the probe proceeds to the next victim. If the chunk is available to be stolen, it is reserved under lock and then transferred outside of the critical region. This is to minimize the time the stack is locked.

When a thread out of work is unable to find any available work in any other stack, it enters a barrier. When all threads have reached the barrier, the traversal is complete. A thread releasing work sets a global variable which in turn releases an idle thread waiting at the barrier.

### 3.3 Data Distribution in OpenMP

Unlike UPC, the OpenMP standard does not provide a way to specify the distribution of an array across the memories of processors [9]. Generally this distribution is accomplished by allocating page frames to memories local to the processor which generated the page fault [10]. However this strategy is prone to false sharing when a large number of processors share a relatively small array of per-processor data structures that are being updated. The symptom of such false sharing is poor scaling to large processor counts.

**Table 2.** Sequential performance for all trees (Thousands of nodes per second)

| System | Processor | Compiler | T1 | T2 | T3 |
|---|---|---|---|---|---|
| Cray X1 | Vector (800 MHz) | Cray 5.5.0.4 | 29 | 29 | 31 |
| SGI Origin 2000 | MIPS (300 MHz) | GCC 3.2.2 | 158 | 156 | 170 |
| SGI Origin 2000 | MIPS (300 MHz) | Mipspro 7.3 | 169 | 166 | 183 |
| Sun SunFire 6800 | Sparc9 (750 MHz) | Sun C 5.5 | 260 | 165 | 384 |
| P4 Xeon Cluster (OSC) | P4 Xeon (2.4GHz) | Intel 8.0 | 717 | 940 | 1354 |
| Mac Powerbook | PPC G4 (1.33GHz) | GCC 4.0 | 774 | 672 | 1117 |
| SGI Altix 3800 | Itanium2 (1.6GHz) | GCC 3.4.4 | 951 | 902 | 1171 |
| SGI Altix 3800 | Itanium2 (1.6GHz) | Intel 8.1 | 1160 | 1106 | 1477 |
| Dell Blade Cluster (UNC) | P4 Xeon (3.6GHz) | Intel 8.0 | 1273 | 1165 | 1866 |

The Irix-specific *distribute_reshape* directive provided compiler-mediated distribution of the array across processor memories, improving performance on the SGI Origin. However we found that on the Intel compiler on the SGI Altix, this directive was not operative.

As a result, we replaced the array of per-thread data with an array of pointers to dynamically allocated structures. During parallel execution, each thread dynamically allocates memory for its own structure with affinity to the thread performing the allocation. This adjustment eliminated false sharing, but was unnecessary for UPC. The facilities for explicit distribution of data in OpenMP are weaker than those of UPC.

## 4 Performance Evaluation

The UTS program reports detailed statistics about load balance, performance, and the tree that is generated. We have examined data on several different parallel machines and discovered some meaningful trends, which are presented in this section.

Three sample trees were used for most of the experiments. Two of these are geometric trees and one is a binomial tree. All trees are approximately the same size (4.1 million nodes, ±1%). However, they vary greatly in depth. The parameters, depth, and size of each tree are given in Table 1.

### 4.1 Sequential Performance

Before examining parallel execution, we ran the benchmark sequentially on a variety of systems. The performance results are given in Fig. 2. Unless otherwise noted, all performance measures are the average of the best eight out of ten executions. Note that some systems were tested with both their native compilers and GCC. This establishes a helpful baseline for the parallel performance results presented later. When compiling UTS with OpenMP, the native compilers were used. The Intrepid UPC compiler, based on GCC, was used to compile UTS with UPC on the shared memory systems. The *-O3* optimization flag was used when

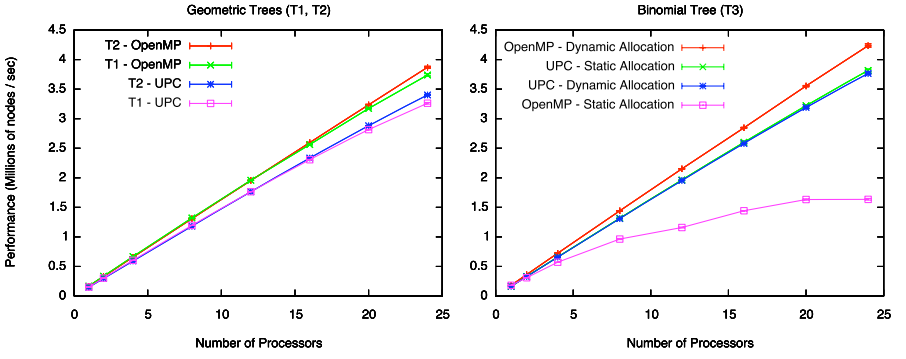


**Fig. 3.** Parallel performance on the Origin 2000 using OpenMP and UPC. On the left, results for geometric trees T1 and T2. On the right, results for T3 with static versus dynamic data allocation for the steals stacks. The chunk size used in each run is chosen to optimize performance for the tree type.

compiling on the shared memory machines. On the Pentium 4 and Dell blade clusters, the Berkeley UPC compiler was used. The Berkeley compiler translates UPC to C for final compilation on the native compiler. An experimental set of optimizations offered by Berkeley UPC was not used in the results reported here, but will be used shortly.

Despite the similar sizes of the trees, most systems showed significant and systematic differences in performance due to differences in cache locality. Ideally, a node will remain in cache between the time it is generated and the time it is visited. The long tail of the geometric distribution results in some nodes generating a large number of children, flushing the L1 cache. In the binomial tree T3, nodes generate only 8 children, if any, keeping L1 cache locality intact for faster execution. The impact of this difference in sequential exploration rate will be seen in the parallel results given in the next section, where we will show that the parallel performance also varies by tree type.

### 4.2 Parallel Performance on Shared Memory Machines

Performance using UPC and OpenMP on the SGI Origin 2000 is given in the left graph of Fig. 3. The OpenMP results for T1 and T2 are from the modified version of the benchmark discussed in Section 3.3. The results for T3 in the right graph of Fig. 3 quantify the performance change between OpenMP implementations using static and dynamic allocation of per-thread data structures. OpenMP performance is dramatically improved using dynamic allocation. UPC performance is slightly decreased using dynamic allocation version, as the extra shared pointer dereference adds to overhead costs. (The two are nearly indistinguishable in the graph.)

The UPC version runs more slowly than the OpenMP version due to the increased overheads of UPC. The variations in runtimes by tree type seen in

Section 4.1 are also present in the parallel runs. Absolute performance on T3 is higher, though all scale well.

### 4.3 Parallel Performance on Distributed Memory

Using the same UPC code, we compiled and ran UTS on a Dell blade cluster at UNC. The cluster's interconnect is Infiniband, and we configured the Berkeley UPC runtime system to run over VAPI drivers, as opposed to UDP or MPI. Still, performance was markedly poor. Figure 4 shows the overall performance of the sample trees and a comparison of parallel efficiency between the SGI Origin and the Dell blade cluster for trees T1 and T3. Note each plotted value represents the best of 100 runs per chunk size on the cluster at that processor count. We have found that the performance varies as much as one order of magnitude even between runs using identical settings. While the efficiency on the Origin is consistently above 0.9, the program's efficiency on the cluster falls off sharply as more processors are used. Poor scaling was also seen on the P4 Xeon Cluster at Ohio Supercomputer Center (OSC).

Poor performance on distributed memory is consistent with previous UPC evaluation [11]. Program designs that assume efficient access of shared variables do not scale well in systems with higher latency.

Key factors contributing to the poor peak performance and high variability are the work-stealing mechanism and termination detection, neither of which create performance bottlenecks on shared memory machines. Each attempted steal involves several communication steps: check whether work is available at the victim, reserve the nodes, and then actually transfer them. Failed steal attempts add additional communication. The termination detection uses a simple cancelable barrier consisting of three shared variables: a cancellation flag, a count of nodes at the barrier, and a completion flag. It is a classic shared variable solution that uses local spinning when implemented in a shared memory machines. We speculate that the distributed memory runtime has a different coherence protocol that is a poor match to algorithms that rely on local spinning.

### 4.4 Visualization for Detailed Analysis

When tracing is enabled, each thread keeps records of when it is working, searching for work, or idle. It also records the victim, the thief, and the time of each steal. Since some systems, in particular distributed memory machines, do not have synchronized clocks, the records kept by the threads are adjusted by an offset. This offset is found by recording the time following the completion of a OpenMP or UPC barrier. This gives millisecond precision on the clusters we tested, but more sophisticated methods could likely do better.

The PARAVER (PARallel Visualization and Events Representation) tool[12] is used to visualize the data. PARAVER displays a series of horizontal bars, one for each thread. There is a time axis below, and the color of each bar at each time value corresponds to that thread's state. Yellow vertical lines drawn between the bars represent the thief and victim threads at the time of each steal.

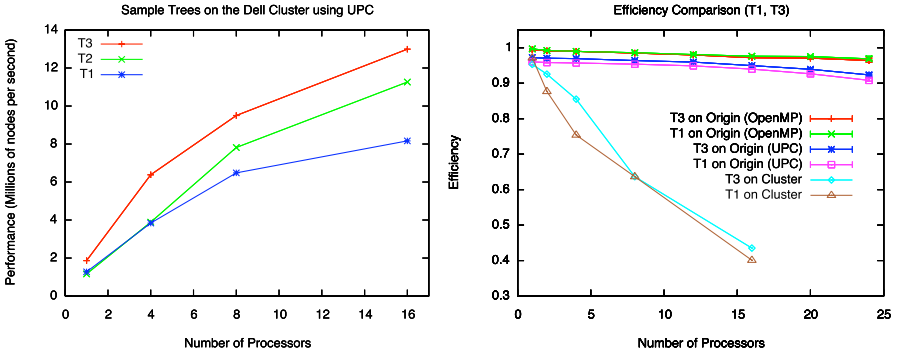


**Fig. 4.** Performance on the Dell blade cluster (left); Parallel efficiency on the Dell cluster versus the Origin 2000 (right)

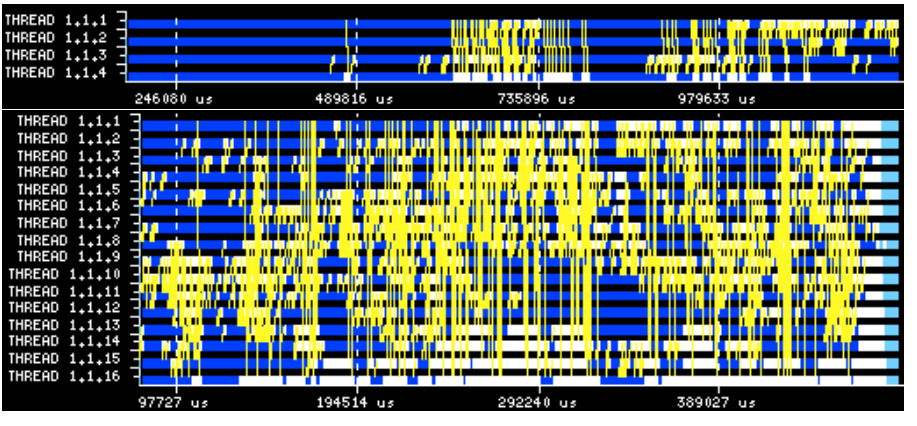


**Fig. 5.** PARAVER time charts for T3 on 4 vs. 16 processors of OSC's P4 cluster

Figure 5 shows PARAVER time charts for two runs on OSC's P4 Xeon cluster at chunk size 64. The bars on the top trace (4 threads) are mostly dark blue, indicating that the threads are at work. The second trace (16 threads) shows a considerable amount of white, showing that much of the time threads are searching for work. There is also noticeable idle time (shown in light blue) in the termination phase. The behavior of the Origin with 16 threads, shown in Fig. 6, is much better, with all threads working almost all of the time.

### 4.5 Work Stealing Granularity

The most important parameter in performance is chunk size, which sets the granularity of work stealing. A thread initiates a steal from another thread only when at least two chunks have accumulated in that thread's steal stack. If the

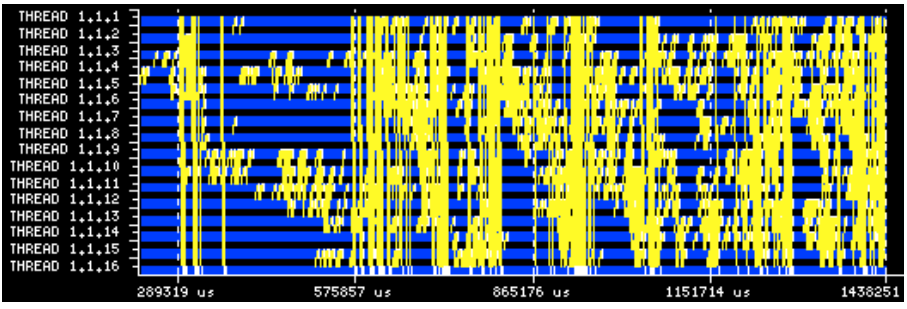


**Fig. 6.** PARAVER time chart for T3 exploration on 16 processors of the Origin 2000

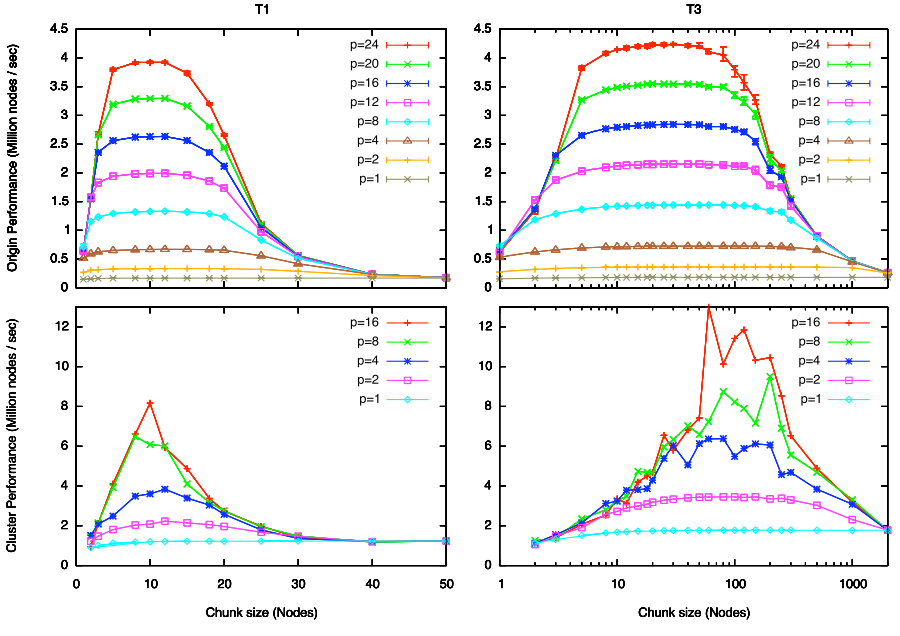


**Fig. 7.** Performance of UTS at various chunk sizes on the SGI Origin 2000 (top) and Dell blade cluster (bottom). For the cluster, the best performance out of 100 executions at each setting is shown.

chunk size is set too small, the threads will be too busy stealing (or trying to steal) to get any work done. If the chunk size is too large, nodes will remain on the stacks where they were generated and too few steals will occur.

**Optimal Ranges for Chunk Size.** Figure 7 shows the performance of UTS at various chunk sizes for trees T1 and T3 on the Origin 2000 using OpenMP and on the Dell blade cluster using UPC. Results for UPC on the Origin were nearly identical to those for OpenMP, and results on the Altix were similar. Note the

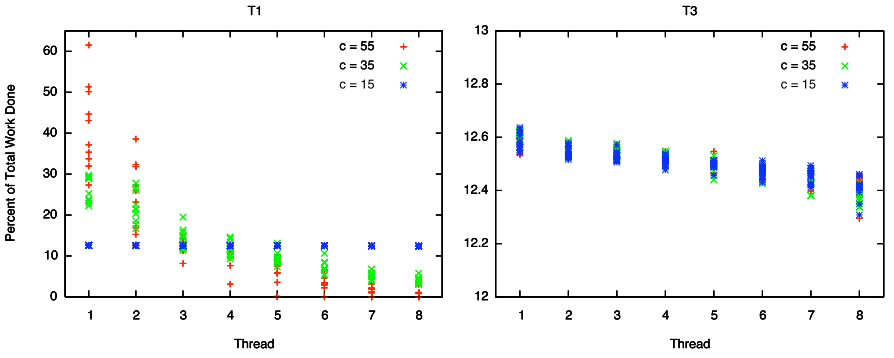


**Fig. 8.** Percent of work done exploring T1 (left) and T3 (right) by each thread in an 8-thread run, plotted for 10 runs per chunk size on the SGI Origin 2000

vast difference in the sizes of the optimal chunk size ranges for the two trees on the Origin. For example, T1 performs reasonably well on 24 processors at a chunk size of 5 to 15 nodes, while T3 performs well a chunk size of 8 to 80. T1 achieves no speedup at a chunk size of 50, but T3 achieves linear speedup at chunk size 250 on four processors and even at chunk size 1000 on two processors.

On the cluster, the performance falls off much more rapidly on either side of the peak values. As with the Origin, the acceptable chunk size range is much larger for T3 than for T1. The peak performance for T1 is achieved with a chunk size of 10 nodes, while chunk sizes of 50 to 150 work well for T3.

Corresponding results for T2 are omitted for lack of space. They resemble those for T3, since the great depth of both trees allows large chunks to accumulate on the stacks, a rare occurrence in the exploration of shallow T1.

**Stealing the Root.** One might suppose that no stealing occurs when chunk size is at or above 50 for T1. However, this is not the case. The poor performance can be attributed to T1's lack of depth, which is limited by the parameter $d$ to give an expected stack size of $b_0 d$. When the chunk size is at or above $\frac{b_0 d}{2}$ nodes, we expect that a chunk will only be released on the infrequent occasion that a node generates more than $b_0$ children, due to the long tail of the geometric distribution. The chunk that is released, and possibly stolen, in such a case will contain the older nodes from the bottom of the stack, i.e. the higher nodes in the tree. The nodes which were generated most recently and are deeper in the tree will be at the top of the stack for the local thread to continue working on.

Now suppose the chunk in the shared stack is stolen. The thief will then hold nodes which are high in the tree and be able to generate some amount of work before reaching the depth limitation. Meanwhile, the victim will quickly, if not immediately, run out of work, because it holds nodes which are already at or near the cutoff depth for the tree. Once the thief generates another chunk of work in the lower portion of the tree, it in turn becomes the victim, giving up the higher nodes. In this way, execution is effectively serialized. The number of

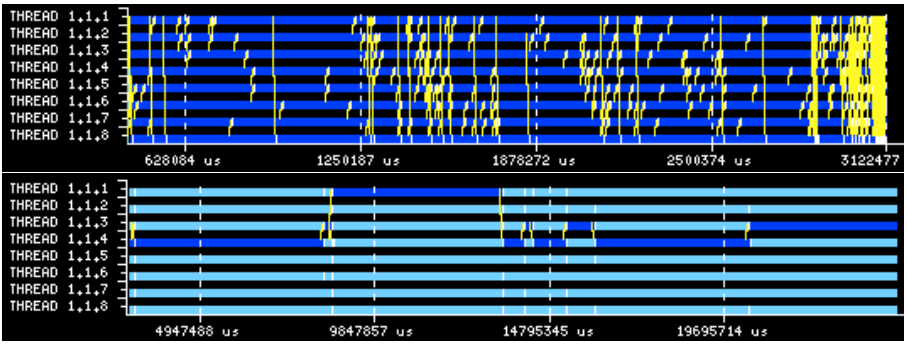


**Fig. 9.** PARAVER time charts for T1 exploration at chunk sizes of 10 nodes (top) and 60 nodes (bottom) on the Origin 2000. Note that when the higher chunk size is used, the work becomes serialized.

steals that occur over the course of the run is in fact a good indication of how many times a value much higher than the expected branching factor was drawn from the binomial distribution.

On the Origin, the root is most often passed back and forth between just two threads. This is a consequence of the non-uniform memory access architecture, in which some threads see shared variable modifications earlier than other threads.

Figure 8 shows ten runs at different chunk sizes for T1 and T3 on the Origin 2000. Note that for chunk sizes of 15 to 55 nodes, the work for T3 is spread evenly among the eight threads, about 12.5% each. The work for T1 is also evenly distributed when the chunk size is 15. With a chunk size of 35, a disproportionate amount of work is done by just a few threads. When chunk size is increased to 55 nodes, work distribution is even more biased. Figure 9 shows PARAVER time charts for runs for T1 with chunk sizes 10 (ideal) and 60 (pathological).

Recall from Table 1 that T1 has a depth limitation $d = 10$ and a constant expected branching factor $b = 4$, yielding an expected stack size of 40 nodes. For any chunk size greater than 20 nodes, the stack rarely accumulates the two-chunks worth of work needed to enable a release. In contrast, binomial trees like T3 have no depth limitation, allowing the stack to grow very deep and facilitating many chunk releases and subsequent steals, even at a large chunk size. This is also the case for deep geometric trees such as T2, which has a much lower branching factor than T1 and a less restrictive depth limitation. The example illustrates well the subtle interactions between the tree shape and the load balance strategy.

### 4.6 Comparing Absolute Performance of Various Machines

UPC is supported by a great variety of shared memory and distributed memory machines. In Fig. 10, a performance comparison for T3 on a variety of shared memory and distributed memory machines running the UPC version of UTS on 16 processors is shown. The OpenMP version is used for the SunFire 6800.

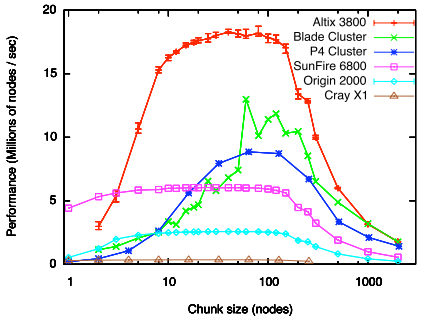


**Fig. 10.** Performance for T3 on 16 processors on various machines. Except on the SunFire, the UPC version was used.

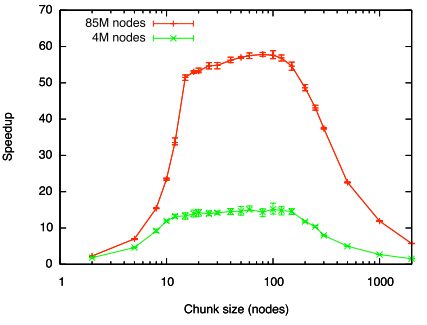


**Fig. 11.** Speedup on the SGI Altix 3800 using 64 processors for T3 and another, much larger, binomial tree

In terms of absolute performance, the Altix is the fastest and the Dell cluster is the second-fastest. If scaling on the cluster were 65% or better, it would overtake the Altix in this UPC comparison.

### 4.7 Larger Trees for More Processors

When running UTS on a large number of processors, the tree must be large enough to produce work for all the threads. Figure 11 shows the speedup of the UPC version of UTS on the Altix 3800 for binomial trees of different sizes using 64 processors. The smallest, T3, is 4.1 million nodes. The other is 85 million nodes in size. The speedup of T3 never approaches that of the larger tree. However, the Altix achieves a near-linear parallel speedup of nearly 60 on the 85M node tree, searching over 67M nodes/sec. Larger trees are able to generate large chunks of work more often. Running at a higher chunk size further amortizes the cost of stealing, as more work is procured per steal than at lower chunk sizes. Thus, a greater number of processors can be effectively utilized.

## 5 Conclusions and Future Work

The primary contributions of our work are twofold: First, we have introduced a novel and challenging benchmark to measure a parallel system's ability to perform substantial dynamic load balancing. Second, we have used the benchmark to investigate the performance portability of shared memory programming models to distributed memory systems. Comparison of the OpenMP and UPC versions showed that both of these easily-programmed implementations exhibit near-ideal scaling and comparable absolute performance on shared memory systems, but the UPC implementation scaled poorly on distributed memory systems. While UPC, with its shared memory abstractions, can simplify programming for clusters, severe performance penalties can be incurred by the implicit

communication costs of a dynamic load-balancing implementation based upon liberal shared variable accesses. We have also presented analysis of situations in which data locality, horizon effects, granularity of load balance, and problem size can dramatically impact the effectiveness of distributed load balancing.

One of our early versions of UTS, *psearch*, was one of several benchmarks used to test recent communications optimizations for UPC [13]. We encourage testing of our implementations with different optimization schemes and on a variety of systems, as well as the development of new implementations. In some future implementations, we will seek to improve performance on distributed memory machines without degrading the performance on shared memory. We will be turning the tables to ask whether a program designed for efficiency on distributed memory can compete with our shared memory implementation on shared memory machines in terms of scaling and absolute performance. In that scenario, just as in those presented in this paper, the UTS benchmark's dynamic load balancing makes it a daunting task for performance portability.

## Acknowledgements

The authors would like to thank the European Center for Parallelism in Barcelona for the use of their PARAVER visualization tool. We thank the Ohio Supercomputer Center for use of their Pentium cluster and the University of North Carolina at Chapel Hill for the use of their Dell blade cluster and Altix. We would also like to thank William Pugh for his involvement in the early stages of the UTS project.

## References

1. T. Harris, *The Theory of Branching Processes*. Springer, 1963.
2. D. Eastlake and P. Jones, "US secure hash algorithm 1 (SHA-1)," Internet Engineering Task Force, RFC 3174, Sept. 2001. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3174.txt
3. J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proc. 11th ACM SIGKDD Int'l Conf. Know. Disc. Data Mining (KDD '05)*, 2005, pp. 177–187.
4. R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proc. 35th Ann. Symp. Found. Comp. Sci.*, Nov. 1994, pp. 356–368.
5. V. Kumar, A. Y. Grama, and N. R. Vempaty, "Scalable load balancing techniques for parallel computers," *J. Par. Dist. Comp.*, vol. 22, no. 1, pp. 60–79, 1994.
6. V. Kumar and V. N. Rao, "Parallel depth first search. part ii. analysis," *Int'l J. Par. Prog.*, vol. 16, no. 6, pp. 501–519, 1987.
7. UPC Consortium, "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.
8. M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. 1998 SIGPLAN Conf. Prog. Lang. Design Impl. (PLDI '98)*, 1998, pp. 212–223.

9. A. Marowka, "Analytic comparison of two advanced c language-based parallel programming models." in *Proc. Third Int'l Symp. Par. and Dist. Comp./Int'l Workshop Algorithms, Models and Tools for Par. Comp. Hetero. Nets. (IS-PDC/HeteroPar'04)*, 2004, pp. 284–291.
10. J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccnuma systems," in *Proc. 11th ACM SIGPLAN Symp. Princ. Pract. Par. Prog. (PPoPP '06)*, 2006, pp. 90–99.
11. 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 *Proc. LCPC 2003*, ser. LNCS, L. Rauchwerger, Ed., vol. 2958, 2003, pp. 194–208.
12. European Center for Parallelism, "PARAVER," 2006. [Online]. Available: http://www.cepba.upc.edu/paraver/
13. W. Chen, C. Iancu, and K. A. Yelick, "Communication optimizations for fine-grained UPC applications." in *Proc. Int'l Conf. Par. Arch. Compilation Tech. (PACT 2005)*, 2005, pp. 267–278.
14. J. Prins, J. Huan, B. Pugh, C. Tseng, and P. Sadayappan, "UPC implementation of an unbalanced tree search benchmark," Univ. North Carolina at Chapel Hill, Tech. Rep. TR03-034, Oct. 2003.