UPC Implementation of an Unbalanced Tree Search Benchmark

Jan Prins, Jun Huan
Univ. of North Carolina
at Chapel Hill

Bill Pugh, Chau-Wen Tseng
Univ. of Maryland
at College Park

P. Sadayappan
Ohio State University

October 2003

Abstract

We have developed an unbalanced tree search problem to evaluate the ease of programming a parallel application requiring dynamic load balancing and to benchmark the performance of such an application on a variety of parallel systems. Here we describe the benchmark, and report on its implementation using Unified Parallel C (UPC). We then examine the performance of the UPC implementation on a number of parallel systems.

1 Description of the benchmark

The goal of the unbalanced tree search benchmark (UTS) is to traverse an implicitly constructed tree with parameterized size and imbalance. Implicit construction means that each node contains all information necessary to completely construct its subtrees. The balance of a tree is a measure of the similarity in the size of its subtrees. Highly unbalanced trees pose significant challenges for parallel traversal because the work required for a simple depth first traversal of different nodes may vary greatly.

The benchmark simply counts the total number of nodes in the tree, but to generate the correct result requires full traversal of the entire tree. Starting from the root node with a specified number of children, the tree can be traversed in parallel and in any order.

Trees are generated using a Galton-Watson process [Har60], in which the number of children for each node (other than the root) is generated following a distribution that is identical but independent for all nodes. We use a binomial distribution in which each node has either zero or m>0 children. To create deterministic results, the number of children is based on a cryptographic hash (SHA-1) associated with each node. Each node is represented by a 20-byte id viewed as a 160-bit unsigned integer, with byte 0 representing the 8 most significant bits and byte 19 representing the 8 least significant bits. The id of the root is r and the number of children of the root is n_r. For a given node v other than the root, the number of children $n(v)$ is determined from the least significant 32 bits of its id as follows

$$n(v) = \begin{cases} m, & \text{if } \dfrac{(v \bmod 2^{32})}{2^{32}} < q \\ 0, & \text{otherwise} \end{cases}$$

where q and m are two parameters chosen so that $0 < q < 1$ is the probability that a node will be an interior node and $1 \le m \le 256$ is the number of children of an interior node. To generate finite trees, we require $qm < 1$.

For an interior node v with m children, the id $c(v,i)$ of its child $0 \le i < m$ is defined as

$$c(v,i) = \text{SHA-1}\ (v \mathbin{+\!\!+}\ i)$$

where $v \mathbin{+\!\!+} i$ is the 24-byte sequence formed by appending the value of i as a four byte value (most significant byte first) to the least significant end of the 20-byte value of v, and SHA-1 is the cryptographic hash function that converts any sequence of bytes (so in particular our sequence of 24 bytes) into a 20 byte *digest* [NIST94]. The use of a good cryptographic hash is important for three reasons: (1) the values of the nodes are likely to be uniformly distributed, (2) the possibility of a collision among node ids (which could give rise to an infinite tree) is infinitesimally small, and (3) carefully validated implementations of SHA-1 exist which ensure that identical trees can be generated from the same parameters on different architectures.

When the least significant 32 bits of the node representation is uniformly distributed, the expected number of children of a node v is $E(n(v)) = qm$ and the expected size of the tree below v is $1/(1-qm)$. Therefore the expected size of the complete tree rooted at r is $S(r) = 1 + n_r/(1-qm)$. The variation in subtree sizes increases as m gets larger, and as qm approaches 1.

Since tree generation using SHA-1 is completely deterministic, a tree is completely specified by the four parameters (r, n_r, q, m), representing the id of the root, the number of children directly below the root, and the two parameters that govern the expected size and imbalance.

The table below lists the sizes of tree generated for three different values of the parameters. Trees T1 and T2 are used to assess correct operation of the algorithm, while tree T3 is a highly unbalanced tree used in performance evaluation.

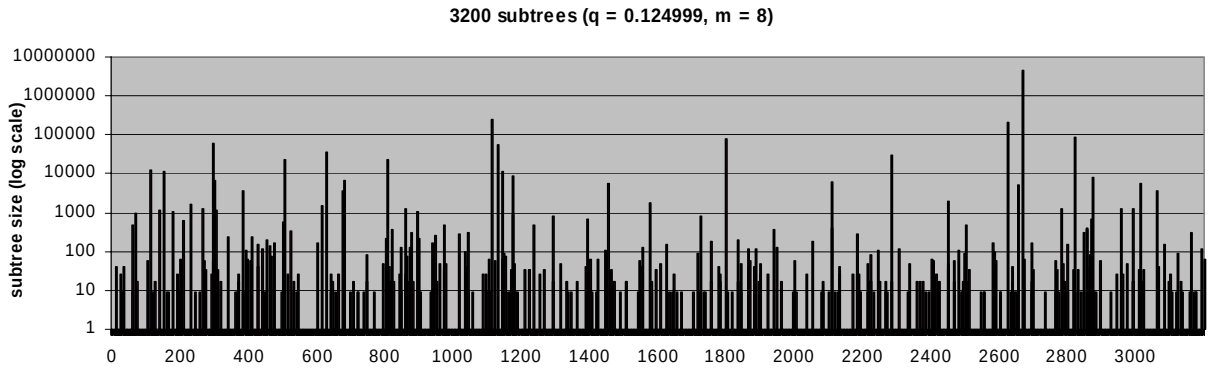
Tree	r (hexadecimal)	n_r	q	m	**S(r)**
T1	0000 … 00000000	3200	0.234375	4	**50,045**
T2	0000 … 00000101	3200	0.234375	4	**53,521**
T3	0000 … 00000000	3200	0.124999	8	**5,529,089**

Performance is reported in nodes evaluated per second.

1.1 Unbalanced Work

The tree generation method allows us to specify highly unbalanced trees by careful selection of the parameters. For example, tree T3 has $qm = 0.999992$ which is close to 1 and hence yields a large total size for T3 (about 5.5M nodes) with high variability in subtree size. The sizes of the 3200 subtrees are shown below (note the logarithmic scale): 82% of all nodes are in a single subtree, 98% of all nodes are contained in just 0.5% of all subtrees, and nearly 90% of the subtrees of have just a single node.

This extreme distribution of sizes necessitates a dynamic load balancing strategy for the efficient parallel traversal of trees. Good load balance using as little as two processors to traverse T3 already requires T3's largest subtree to be "split up" (as it holds 82% of the work). All nodes look alike in terms of their possible subtree size, hence it is impossible to identify a set of nodes that subtend a given amount of work without traversing the subtrees below those nodes. Efficient parallel traversal requires ongoing fine-grain interaction among parallel tasks to balance load.



1.2 Available parallelism

In gross terms the available parallelism is limited by the ratio of tree size to tree height. For example, tree T3 has about 5.5M nodes and a maximum height of about 1300. Since traversal to depth 1300 requires a chain of 1300 dependent SHA-1 evaluations, we cannot reduce running time when more than 5.5M/1300 ≈ 4000 processors are used. In practice the usable parallelism limit would be reached much earlier due to other serialization overheads such as those introduced by the load balancing strategy. We can adjust the available parallelism by increasing m while decreasing q to keep qm approximately constant.

2 Implementation

A variety of strategies have been proposed to dynamically balance load in parallel computation. 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 [BL94].

Our initial implementations of the UTS problem have used various forms of work-stealing. However, the UTS problem is a difficult adversary for a work-stealing strategy. By construction, the expected size of a subtree below an unexplored node is the same no matter where the node occurs in the tree, hence there is no way to maximize the expected work stolen other than to steal a lot of unexplored nodes. However, the likelihood that a depth first search of a tree has T unexplored nodes on the stack at a given time varies as $1/T$, hence it may be difficult to find large amounts of work to steal. This is one of the properties of the UTS problem that makes it a challenging benchmark. It is indeed our goal to construct a benchmark 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 with such parameter settings.

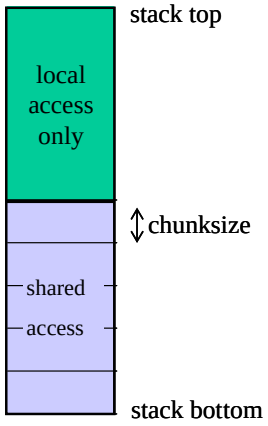
2.1 UPC implementation of UTS

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 [CDC99]. The model can be compiled for shared memory or distributed memory execution. It is the compiler's responsibility to translate memory addresses and insert inter-processor communication. A distinguishing feature of UPC is that global pointers may be cast into local pointers for efficient local access. 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 UPC. Instead of trying to steal procedure continuations as would be done in Cilk, which requires cooperation from the compiler, in our UPC implementation we steal units of work from a set of shared stacks, one for each parallel thread. The stacks hold tree nodes. 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 and push them onto its own stack for traversal. On completion the total number of nodes traversed in each thread can be combined to yield the size of the complete tree.

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 any overheads in the depth-first traversal performed by each thread as much as possible. Thus each thread must be able to perform *push* and *pop* operations at stack top without incurring UPC shared address translation overheads or requiring locking operations. The design at right shows the stack partitioned into two regions. 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 an access lock. To amortize the manipulation overheads, nodes can only move in chunks of a given size between the local and shared regions or between shared areas in different stacks.



The stack design is easily expressed in UPC:

```
struct stealStack_t
{
  NODE stack[MAXDEPTH];   /* array representation of stack */
  int sharedStart;        /* start index of shared portion */
  int local;              /* start index of local portion  */
  int top;                /* index of stack top */
  upc_lock_t *stackLock;  /* access lock for shared portion */
  int workAvail;          /* # nodes available to steal */
};

typedef struct stealStack_t STEALSTACK;
```

The *stealStack* shared array provides a STEALSTACK for each thread. A local STEALSTACK pointer *ss* provides direct access to the stack with affinity to the thread.

```
shared STEALSTACK stealStack[THREADS];
ss = (STEALSTACK *) &stealStack[MYTHREAD];
```

Using this data structure, the local push operation (for example) does not involve any shared address references or lock operations:

```
/* local push */
void push(STEALSTACK *s, NODE *c) {
  if (s->top >= MAXDEPTH)
    error("StealStack::push  overflow");
  memcpy(&s->stack[s->top], c, sizeof(NODE));
  s->top++;
}
```

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 2*k* 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 into the local stack when the local stack becomes empty.

```
/* release k nodes from local stack for shared access */
void release(STEALSTACK *s, int k) {
  upc_lock(s->stackLock);
  if (s->top - s->local >= k) {
    s->local += k;
    s->workAvail += k;
  }
  else
    error("StealStack::release  do not have k nodes to release");
  upc_unlock(s->stackLock);
}
```

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 near the end of the traversal, *workAvail* is examined in each stack without locking. Hence a subsequent steal operation may not succeed when the victim stack no longer has the chunk of nodes observed during the probe. In this case the probe is retried.

If the chunk is available to be stolen, it is reserved in the critical region, and then transferred outside of the critical region. This is to minimize the time the stack is locked. The strategy is safe because each successful steal operation raises the stack bottom for subsequent operations. Since the number of steals is quite small compared to the maximum stack size, this is reasonable. A more complex implementation could reuse the space at a slightly higher cost.

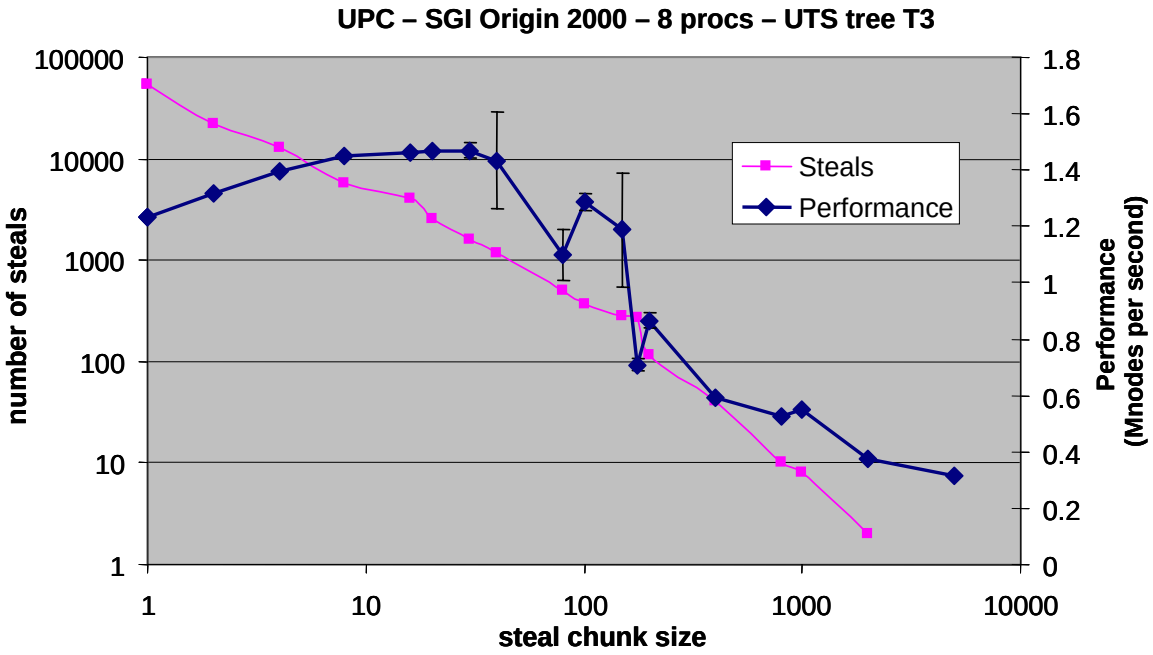
When a thread out of work is unable to find any available work in any other stack, it enters a barrier and contributes its count to the total. Because there may be a few nodes still being explored in the local portion of a stack of some other thread, and these nodes might subsequently produce a large subtree, it is possible that with the current implementation a thread could stop prematurely, although this is highly unlikely. That eventuality would only have an effect in the overall performance, not the correctness of the implementation. A proper two-phase protocol for detecting quiescence of the computation is needed but was not implemented in the interest of expediency.

```
/* steal k elts from thread i onto local stack
 * return false if k elts were not available in thread i
 */
int steal(STEALSTACK *s, int i, int k) {
  int victimLocal, victimShared, victimWorkAvail;
  int ok;

  /* lock stack in thread i and try to reserve k elts */
  upc_lock(stealStack[i].stackLock);
  victimLocal = stealStack[i].local;
  victimShared = stealStack[i].sharedStart;
  victimWorkAvail = stealStack[i].workAvail;
  ok = victimWorkAvail >= k;
  if (ok) {
    stealStack[i].sharedStart =  victimShared + k;
    stealStack[i].workAvail = victimWorkAvail - k;
  }
  upc_unlock(stealStack[i].stackLock);

  /* if reservation succeeded, move elts to local stack */
  if (ok) {
    upc_memcpy(&stealStack[MYTHREAD].stack[s->top],
               &stealStack[i].stack[victimShared],
               k * sizeof(NODE)
               );
    s->top += k;
  }

  return (ok);
}
```

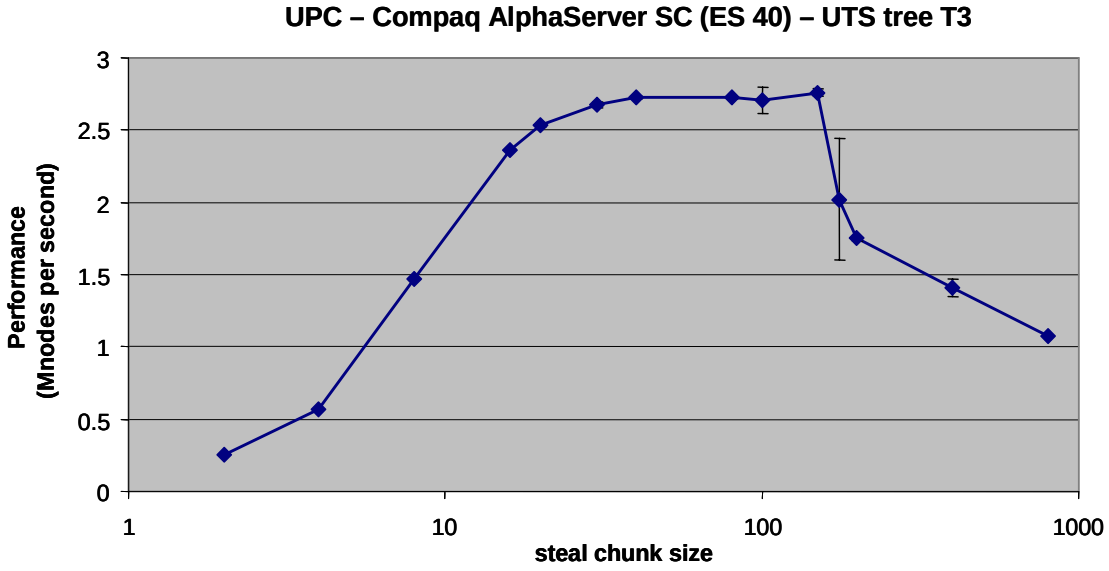
Implementation size. Excluding the implementation of SHA-1, the total size of the UPC implementation of UTS is 517 lines. If the shortcuts mentioned above were implemented properly, this would likely add another 50% in length to the UPC code.

Performance characteristics. An important feature of the work stealing implementation is that the chunksize can be varied independent of the problem size. The chunksize has a noticeable performance impact, particularly in the case of distributed memory machine. If the chunksize is too small, the overheads encountered in work stealing are not well amortized, and the performance decreases. If the chunksize is too large, then we will only occasionally find a subtree that grows large enough to generate a stealable chunk of work. In this case the load balance is poor, leading to lower performance.

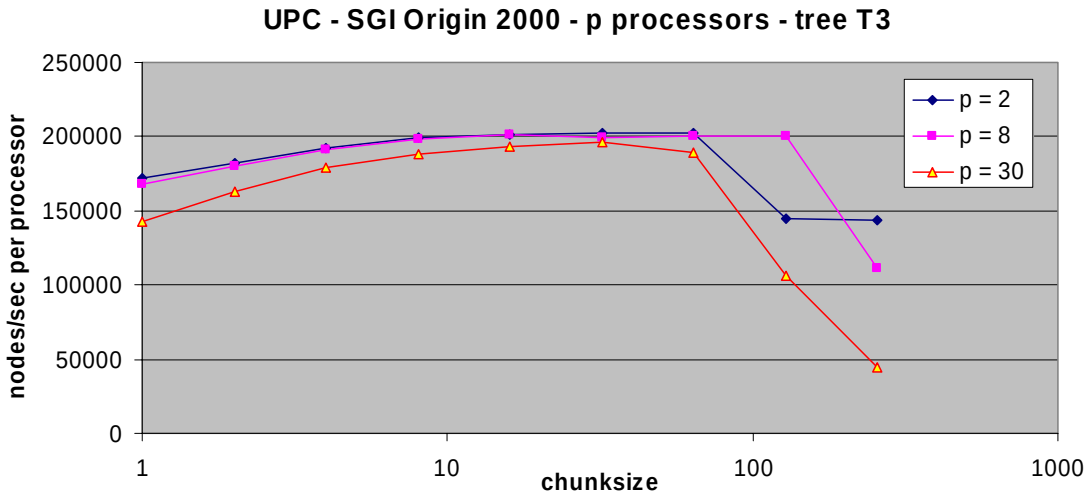
The graph below illustrates this effect. In this case the performance and number of steals in the UTS benchmark for tree T3 are reported using 8 processors of an SGI Origin 2000. The optimal chunk size is around 20 in this case. Since the Origin is a shared-memory machine and efficiently supports fine grain operations, the deterioration of performance at very small chunk size is limited, but still noticeable. The performance impact of a large chunk size is clearly visible as the number of available steals starts to approach the number of processors. The large variation in performance of runs with chunksize in the range 50-200 (as shown by the error bars in the performance curve) indicate that work stealing becomes unstable in the transition from small to large chunks.



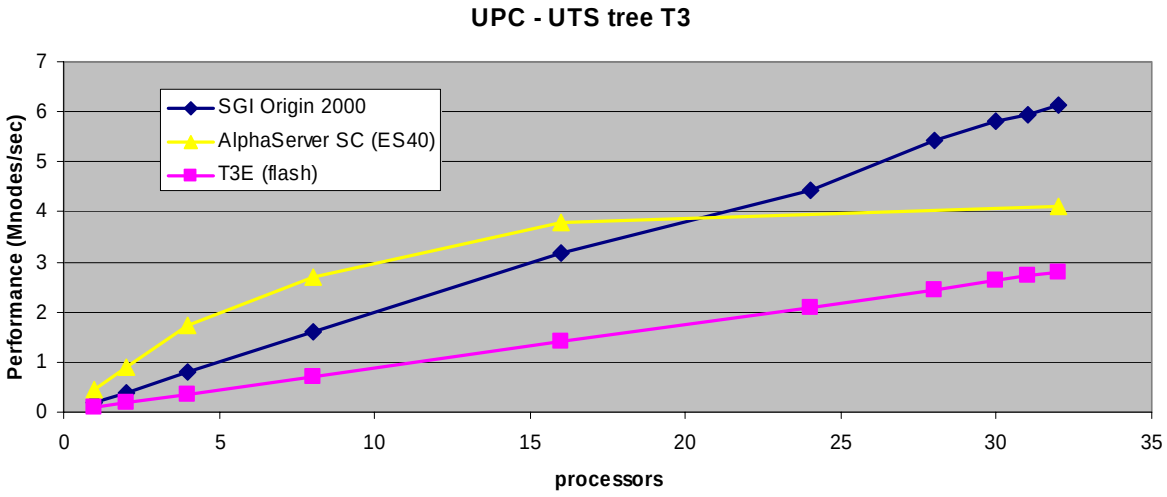
The impact of an overly small chunksize on the communication performance on UTS is better illustrated when the same experiment is run using 8 processors of a Compaq Alphaserver SC (ES40), a machine with a much higher UPC communication latency. In this case the optimal chunksize for maximal performance is closer to 100 than 20. At this chunksize, the AlphaServer has an overall performance that is nearly double the Origin 2000 performance at the same processor count.



The optimal chunksize also increases slightly with the *number* of processors due to higher contention overheads in work stealing at larger processor counts. Note that in the following graph, the performance is normalized per processor.



Performance variation with architectuire. We summarize the performance of the UPC implementation of UTS on a number of different machines in the graph below. Both the T3E and the Origin 2000 exhibit excellent scaling. The The AlphaServer SC, while offering very high single processor performance, is handicapped by very high latencies once communication between nodes is required.

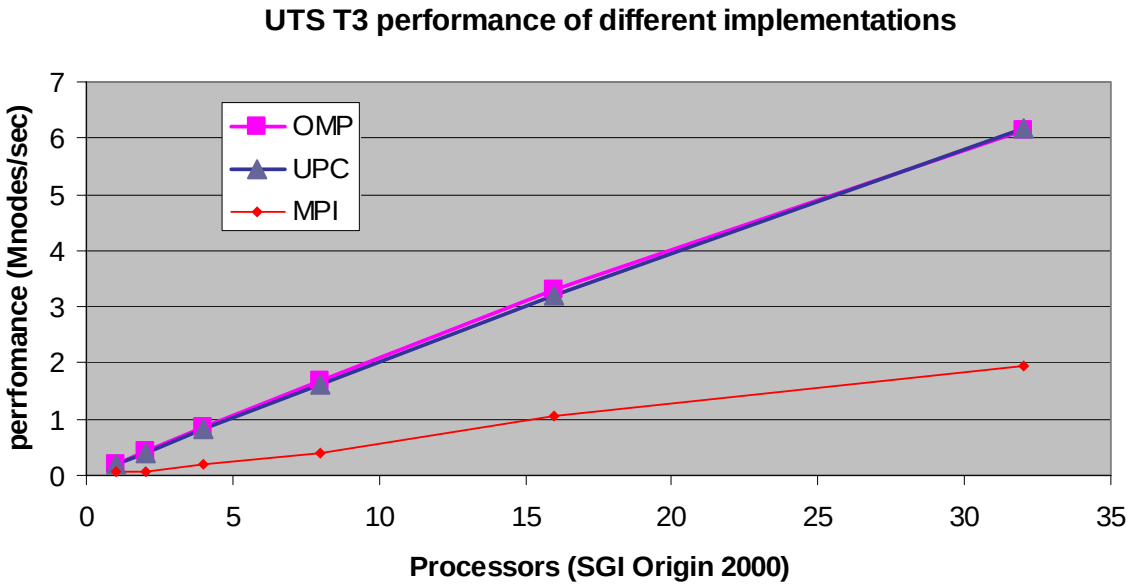


A UPC optimization strategy that might provide a large payoff on the AlphaServer would be to compile a sequence of statements of the form lock – update –unlock on shared data, (as found in the *steal* procedure above) to use an "active message" that runs the body of the update on the remote processor once the lock is required. On the AlphaServer it appears that each reference in the critical section was sent as a separate communication once the lock was acquired, and this long –lived critical section resulted in increased contention from other processors. On the Origin and the T3E, with low-latency fine-grain messages, the overhead of sending several non local references in the critical section is not as large hence did not exhibit the poor scaling behavior.

Performance variation with programming paradigm. The UTS benchmark was implemented using OpenMP and MPI, and the performance of these implementations was compared with the UPC implementation using a 32 processor Origin 2000. A Cilk implementation developed by Bradley Kuszmaul at MIT was run on a slightly older model of the Origin 2000 utilizing slower processors. While we can not directly compare the performance of the Cilk and the UPC implementations at the moment, our preliminary results indicate their performance is very similar when taking into account the processor speed variations.

The initial OpenMP implementation based on a work-stealing strategy that used some centralized data structures exhibited poor performance and scaling. An alternate OpenMP version was derived directly from the UPC implementation, and follows it quite closely (the code size is very similar at 540 lines). As shown below, this OpenMP implementation exhibits performance very similar to the UTS implementation on the shared-memory Origin 2000. The OpenMP version can of course not be run in a competitive fashion on a distributed memory machine.

Another version of the work-stealing strategy developed for a distributed memory model using MPI, exhibits lower performance and scalability. The size of this version is 1068 lines (excluding the SHA-1 implementation).



3 Bibliography

[BL94] R. Blumofe, C. Leiserson, "Scheduling multithreaded computations by work stealing", *Proc. 35th Annual Symposium on Foundations of Computer Science* (FOCS), ACM, 1994.

[CDC99] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification", Center for Computing Sciences Technical Report CCS-TR-99-157, May 1999.

[Har60] T. E. Harris, *The Theory of Branching Processes*, Springer (Berlin), 1960.

[NIST94] Secure Hash Algorithm (Technical revision SHA-1), Federal Information Processing Standards Publication 180-1 (1994), `http://www.itl.nist.gov/fipspubs/fip180-1.htm`