Unbalanced Tree Search Benchmark

Jan Prins
UNC Chapel Hill
August 2003

1 Description of 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 imbalance of a tree is a measure of the dissimilarity in the size of its subtrees.

The benchmark itself does not compute anything useful, but requires the full traversal of the tree in order to generate the correct result. The result to be reported is the total number of nodes in the tree. Starting from a root with a specified number of children, the tree can be traversed in parallel and in any order.

Each node in the tree 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, 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 must have $qm < 1$.

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. 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*. The use of a good cryptographic hash is important for three reasons: (1) the values of the nodes will 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 the 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.

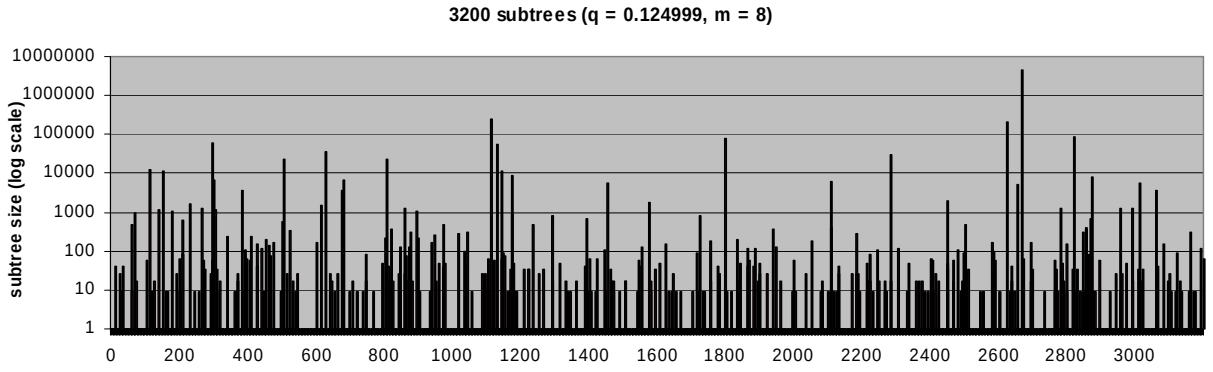
Tree	r (hexadecimal)	n_r	q	m	**S*(r)***
T1	0000 … 00000000	3200	0.234375	4	**50,045**
T2	0000 … 00000101	3200	0.234375	4	**53,521**
T3	0000 … 00000000	3200	0.124999	8	**5,529,089**

Performance is reported in nodes evaluated per second.

2 Unbalanced Work

The tree generation method allows us to specify highly unbalanced trees by careful selection of the parameters. For example, tree T3 is generated using $m = 8$ and $qm = 0.999992$ which is very close to 1 and which 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. Unlike the n-queens problem, then, efficient parallel traversal requires ongoing fine-grain interaction among parallel tasks to balance load.



3 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. Further

investigation is needed in this area of benchmark control; thus far we have determined satisfactory parameter settings empirically. Limiting tree height directly is possible, but relinquishes the uniform tree property. In any case it has been possible so far to find parameter settings that can be used to specify arbitrarily large UTS problem instances with available parallelism suited for large parallel machines.

4 Parameter settings

One interesting characteristic of the tree search benchmark is that we can vary the amount of data and work associated with each node in order to model a variety of applications. Generally speaking, increasing the amount of work associated with a node increases the granularity of the benchmark and makes it more amenable to efficient execution on a larger class of machines. The most challenging version of the benchmark minimizes the additional work per node. In comparison, varying the method for generating children nodes changes the shape of the search tree and amount of load imbalance.

Generating parameter settings that yield appropriately large problem sizes with specified imbalance and available parallelism for this problem is currently a matter of trial and error. Further work is needed to improve the understanding and use of the controls to generate suitable problem instances, and to configure the benchmark to more accurately predict the performance of classes of applications.

5 Implementations

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.

Our initial implementations of the UTS problem have used various forms of work-stealing strategies. 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.

Further work is required to more completely explore load-balancing strategies best suited to this problem. It is indeed our goal to construct a benchmark that in its most challenging form challenges all load balancing strategies, since such a parameter setting can be used to assess some key characteristics of the underlying computing system. For example, distributed-memory systems that require coarse-grain communication to achieve high performance may be fundamentally disadvantaged with such parameter settings.