Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs

Stephen L. Olivier and Jan F. Prins

University of North Carolina at Chapel Hill, Chapel Hill NC 27599, USA
{olivier, prins}@unc.edu

Abstract. The UTS benchmark is used to evaluate task parallelism in OpenMP 3.0 as implemented in a number of recently released compilers and run-time systems. UTS performs parallel search of an irregular and unpredictable search space, as arises e.g. in combinatorial optimization problems. As such UTS presents a highly unbalanced task graph that challenges scheduling, load balancing, termination detection, and task coarsening strategies. Scalability and overheads are compared for OpenMP 3.0, Cilk, and an OpenMP implementation of the benchmark without tasks that performs all scheduling, load balancing, and termination detection explicitly. Current OpenMP 3.0 implementations generally exhibit poor behavior on the UTS benchmark.

1 Introduction

The recent addition of task parallelism support to OpenMP 3.0 [1] offers improved means for application programmers to achieve performance and productivity on shared memory platforms such as multi-core processors. However, efficient execution of task parallelism requires support from compilers and run time systems. Design decisions for those systems include choosing strategies for task scheduling and load-balancing, as well as minimizing overhead costs.

Evaluating the efficiency of run time systems is difficult; the applications they support vary widely. Among the most challenging are those based on unpredictable and irregular computation. The Unbalanced Tree Search (UTS) benchmark [2] represents a class of such applications requiring continuous load balance to achieve parallel speedup. In this paper, we compare the performance and scalability of the UTS benchmark on three different OpenMP 3.0 implementations (Intel icc 11, Mercurium 1.2.1, SunStudio 12) and an experimental prerelease of gcc 4.4 that includes OpenMP 3.0 support. For comparison we also examine the performance of the UTS benchmark using Cilk [3] tasks and using an OpenMP implementation without tasks that performs all scheduling, load balancing, and termination detection explicitly. Throughout this paper we will refer to the latter as the thread-level parallel implementation. Additional experiments focus on comparing overhead costs. The primary contribution of the paper is an analysis of the experimental results for a set of compilers that support task parallelism.

The remainder of the paper is organized as follows: Section 2 outlines background and related work on run time support for task parallelism. Section 3 describes the UTS benchmark. Section 4 presents the experimental results and analysis. We conclude in Section 5 with some recommendations based on our findings.

2 Background and Related Work

Many theoretical and practical issues of task parallel languages and their run time implementations were explored during the development of earlier task parallel programming models, such as Cilk [4, 3]. The issues can be viewed in the framework of the dynamically unfolding task graph in which nodes represent tasks and edges represent completion dependencies.

The *scheduling strategy* determines which ready tasks to execute next on available processing resources. The *load balancing strategy* keeps all processors supplied with work throughout execution. Scheduling is typically decentralized to minimize contention and locking costs that limit scalability of centralized schedulers. However decentralized scheduling increases the complexity of load balancing when a local scheduler runs out of tasks, determining readiness of tasks, and determining global completion of all tasks.

To decrease overheads, various *coarsening strategies* are followed to aggregate multiple tasks together, or to execute serial versions of tasks that elide synchronization support when not needed. However such coarsening may have negative impact on load balancing and availability of parallel work.

Cilk scheduling uses a *work-first* scheduling strategy coupled with a randomized work stealing load balancing strategy shown to be optimal[5]. A lazy task creation approach, developed for parallel implementations of functional languages [6], makes parallel slack accessible while avoiding overhead costs until more parallelism is actually needed. The compiler creates a fast and a slow clone for each task in a Cilk program. Local execution always begins via execution of the fast clone, which replaces task creation with procedure invocation. An idle processor may steal a suspended parent invocation from the execution stack, converting it to the slow clone for parallel execution.

In OpenMP task support, "cutoff" methods to limit overheads were proposed in [7]. When cutoff thresholds are exceeded, new tasks are serialized. One proposed cutoff method, *max-level*, is based on the number of ancestors, i.e. the level of recursion for divide-and-conquer programs. Another is based on the number of tasks in the system, specified as some factor k times the number of threads. The study in [7] finds that performance is often poor when no cutoff is used and that different cutoff strategies are best for different applications. Adaptive Task Cutoff (ATC) is a scheme to select the cutoff at runtime based on profiling data collected early in the program's execution [8]. In experiments, performance with ATC is similar to performance with manually specified optimal cutoffs. However, both leave room for improvement on unbalanced task graphs.

Iterative chunking coarsens the granularity of tasks generated in loops [9]. Aggregation is implemented through compiler transformations. Experiments show mixed results, as some improvements are in the noise compared to overheads of the run time system.

Intel's "workqueuing" model was a proprietary OpenMP extension for task parallelism [10]. In addition to the task construct, a taskq construct defined queues of tasks explicitly. A noteworthy feature was support for reductions among tasks in a task queue. Early evaluations of OpenMP tasking made comparisons to Intel workqueuing, showing similar performance on a suite of seven applications [11].

An extension of the Nanos Mercurium research compiler and run time [11] has served as the prototype compiler and run time for OpenMP task support. An evaluation of scheduling strategies for tasks using Nanos is presented in [7]. That

study concluded that in situations where each task is *tied*, i.e. fixed to the thread on which it first executes, breadth-first schedulers perform best. They found that programs using *untied* tasks, i.e. tasks allowed to migrate between threads when resuming after suspension, perform better using work-first schedulers. A task should be tied if it requires that successive accesses to a threadprivate variable be to the same thread's copy of that variable. Otherwise, untied tasks may be used for greater scheduling flexibility.

Several production compilers have now incorporated OpenMP task support. IBM's implementation for their Power XLC compilers is presented in [12]. The upcoming version 4.4 release of the GNU compilers [13] will include the first production open-source implementation of OpenMP tasks. Commercial compilers are typically closed source, underscoring the need for challenging benchmarks for black-box evaluation.

3 The UTS Benchmark

The UTS problem [2] 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 $m = 8$ children. The description of each child is obtained by an evaluation of the SHA-1 cryptographic hash function [14] 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 not be retained throughout the search.

Load balancing of UTS is particularly challenging since the distribution of subtree sizes follows a power law. While the variance in subtree sizes is enormous, the expected subtree size is identical at all nodes in the tree, so there is no advantage to be gained by stealing one node over another. For the purpose of evaluating run time load-balancing support, the UTS trees are a particularly challenging adversary.

3.1 Task Parallel Implementation

To implement UTS using task parallelism, we let the exploration of each node be a task, allowing the underlying run time system to perform load balancing as needed. A sketch of the implementation follows below:

```
void Generate_and_Traverse(Node* parentNode, int childNumber) {
  Node* currentNode = generateID(parentNode, childNumber);
  nodeCount++;   // threadprivate, combined at termination
  int numChildren = m with prob q, 0 with prob 1-q
  for (i = 0; i < numChildren; i++) {
    #pragma omp task untied firstprivate(i)
      Generate_and_Traverse(currentNode, i);
  }
}
```

Execution is started by creating a parallel region (with a threadprivate counter for the number of nodes counted by the thread). Within the parallel region a single thread creates tasks to count the subtrees below the root. A single taskwait is used to end the parallel region when the entire tree has been explored.

3.2 Thread-Level Parallel Implementation Without Tasks

Unlike the task parallel implementation of UTS, the thread-level parallel implementation described in [2] using OpenMP 2.0 explicitly specifies choices for the order of traversal (depth-first), load balancing technique (work stealing), aggregation of work, and termination detection. A sketch of the implementation follows below:

```
void Generate_and_Traverse(nodeStack* stack) {
  #pragma omp parallel
    while (1) {
      if (empty(stack)) {
        ... steal work from other threads or terminate ...
      }
      currentNode = pop(stack);
      nodeCount++;   // threadprivate, gathered using critical later
      int numChildren = m with prob q, 0 with prob q-1
      for (i = 0; i < numChildren; i++) {
         ...initialize childNode...
         childNode = generateID(currentNode, i);
         push(stack, childNode);
      }
    }
}
```

Execution is started with the root node on the nodeStack of one thread; all other threads start with an empty stack. Note that the single parallel region manages load balancing among threads, termination detection, and the actual tree traversal.

3.3 Cilk Implementation

For comparison, we created a cilk implementation of UTS which is close to the OpenMP 3.0 task implementation. It differs mainly in its use of a Cilk inlet in the search function to accumulate partial results for the tree node count as spawned functions return. The Cilk runtime handles the required synchronization to update the count.

4 Experimental Evaluation

We evaluate OpenMP task support by running UTS and related experiments on an Opteron SMP system. The Opteron system consists of eight dual-core AMD Opteron 8220 processors running at 2.8 Ghz, with 1MB cache per core.

We installed gcc 4.3.2, the Intel icc 11.0 compiler, SunStudio 12 with Ceres C 5.10, and an experimental prerelease of gcc 4.4 (12/19/2008 build). We also installed the Mecurium 1.2.1 research compiler with Nanos 4.1.2 run time. Since Nanos does not yet natively support the x86-64 architecture, we built and used the compiler for 32-bit IA32. We used cilk 5.4.6 for comparison with the OpenMP implementations on both machines; it uses the gcc compiler as a back end. The -O3 option is always used. Unless otherwise noted, reported results represent the average of 10 trials.

For a few results in Section 4.4 of the paper, we used an SGI Altix running Intel icc 11 and Nanos/Mercurium built for IA64. Details for that system are presented in that section.

Implementation	gcc 4.3.2	Intel icc 11.0	Sun Ceres 5.10	gcc 4.4.0
Task Parallel	2.60	2.45	2.38	2.60
Thread-Level Parallel	2.48	2.49	2.17	2.39

Table 1: Sequential performance on Opteron (Millions of tree nodes per second)

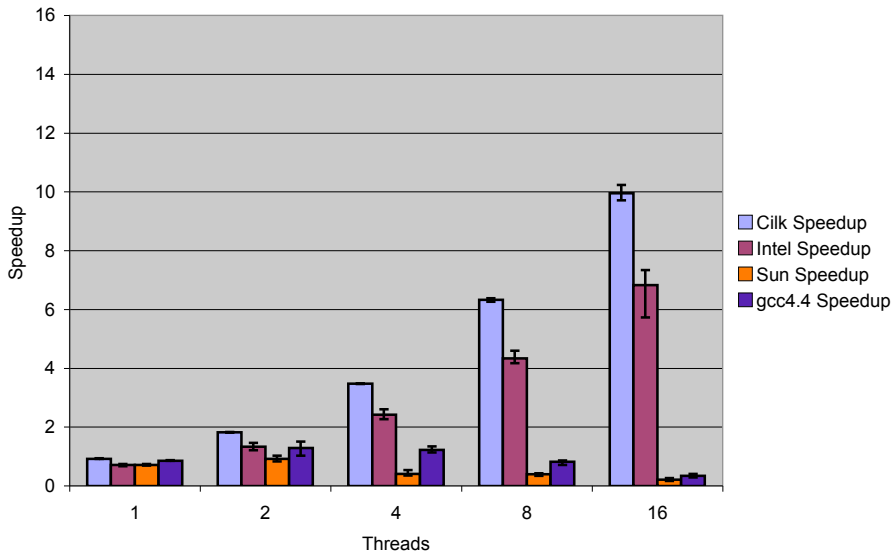


Fig. 1: UTS using cilk and several OpenMP 3.0 task implementations: Speedup on 16-way Opteron SMP. See Figure 7 and Section 4.4 for results using Nanos.

4.1 Sequential and Parallel Performance on UTS

Table 1 shows sequential performance for UTS on the Opteron SMP system; the execution rate represents the number of tree nodes explored per unit time. We use tree *T3* from [2], a 4.1 million node tree with extreme imbalance. This tree is used in experiments throughout the paper. The table gives results for both the task parallel and the thread-level parallel implementations. They were compiled with OpenMP support disabled.

Figure 1 shows the speedup gained on the task parallel implementation using OpenMP 3.0, as measured against the sequential performance data given in table 1. We observe no speedup from Sun Studio and gcc. Cilk outperforms the Intel OpenMP task implementation, but neither achieve more than 10X speedup, though both show improved speedup as up to 16 cores are used. Figure 2 shows the speedup, over 15X in most cases, using the thread-level parallel implementation.

4.2 Analysis of Performance

Two factors leading to poor performance are overhead costs and load imbalance. There is a fundamental tradeoff between them, since load balancing operations

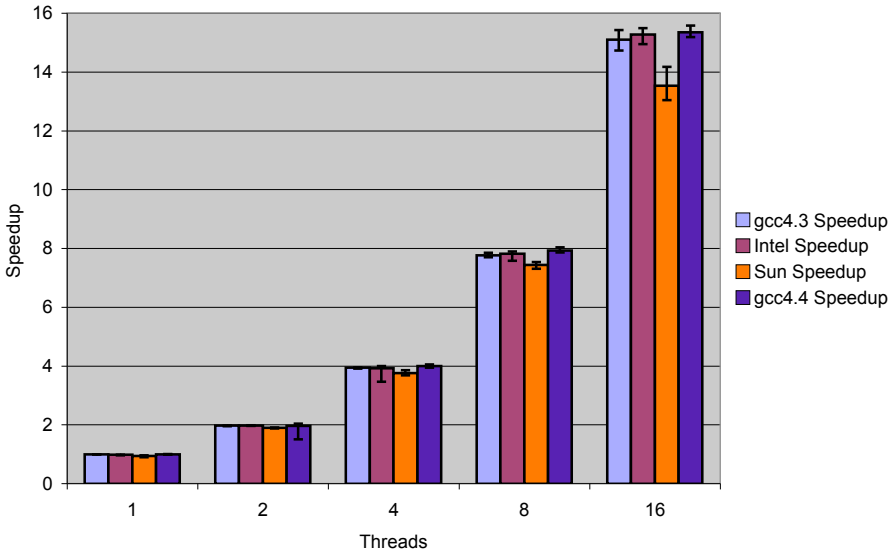


Fig. 2: UTS using thread-level OpenMP parallel implementation without tasks: Speedup on 16-way Opteron SMP. Work stealing granularity is a user-supplied parameter. The optimal value (64 tree nodes transferred per steal operation) was determined by manual tuning and used in these experiments.

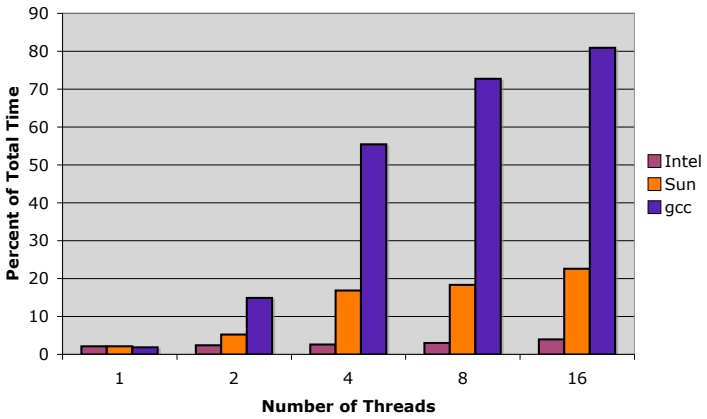


Fig. 3: Overheads (time not calculating SHA-1 hash) on UTS using 100 repetitions of the SHA-1 hash per tree node.

incur overhead costs. Though all run time implementations are forced to deal with that tradeoff, clever ones minimize both to the extent possible. Poor implementations show both crippling overheads and poor load balance.

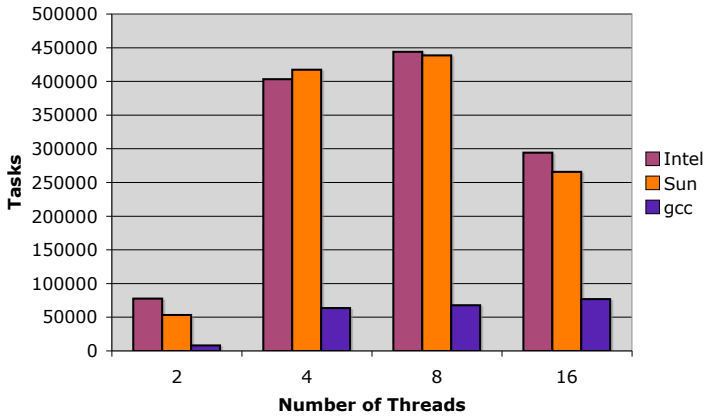


Fig. 4: Number of tasks started on different threads from their parent tasks or migrating during task execution, indicating load balancing efforts. 4.1M tasks are performed in each program execution.

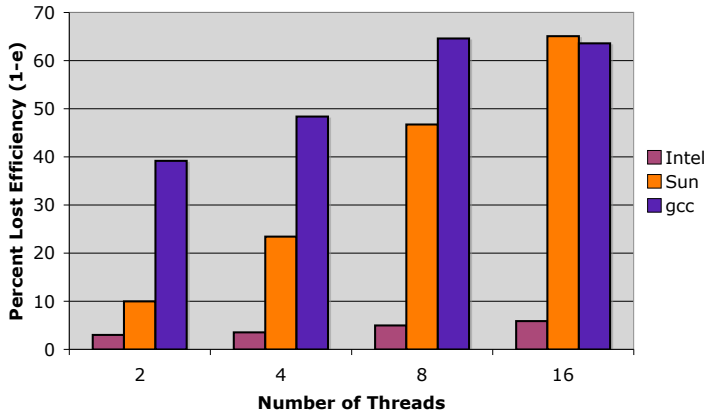


Fig. 5: UTS on Opteron: Lost efficiency due to load imbalance.

Overhead Costs Even when only a single thread is used, there are some overhead costs incurred using OpenMP. For task parallel implementation of UTS, Cilk achieves 96% efficiency, but efficiency is just above 70% using OpenMP 3.0 on a single processor using the Intel and Sun compilers and 85% using gcc 4.4. The thread-level parallel implementation achieves 97-99% on the Intel and gcc compilers and 94% on the Sun compiler.

To quantify the scaling of overhead costs in the OpenMP task run times, we instrumented UTS to record the amount of time spent on work (calculating SHA-1 hashes). To minimize perturbation from the timing calls, we increased the amount of computation by performing 100 SHA-1 hash evaluations of each node. Figure 3 presents the percent of total time spent on overheads (time not spent on SHA-1 calculations). Overhead costs grow sharply in the gcc implementation,

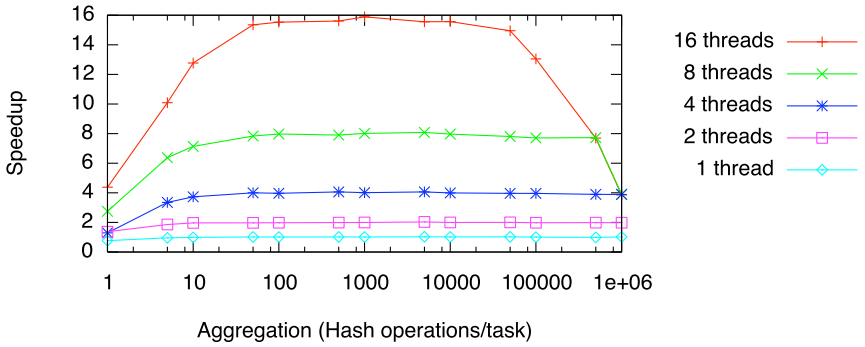


Fig. 6: Work aggregated into tasks. Speedup on Opteron SMP using the Intel OpenMP tasks implementation. Results are similar using the gcc 4.4 and Sun compilers, though slightly poorer at the lower aggregation levels.

dwarfing the time spent on work. The Sun implementation also suffers from high overheads, reaching over 20% of the total run time. Overheads grow slowly from 2% to 4% in the Intel run time. Note that we increased the granularity of computation 100-fold, so overheads on the original fine-grained problem may be much higher still.

Load Imbalance Now we consider the critical issue of load imbalance. To investigate the number of load balancing operations, we modified UTS to record the number of tasks that start on a different thread than the thread they were generated from or that migrate when suspended and subsequently resumed. Figure 4 shows the results of our experiments using the same 4.1M node tree (T3), indicating nearly 450k load balancing operations performed by the Intel and Sun run times per trial using 8 threads. That comprises 11% of the 4.1M tasks generated. In contrast, gcc only performs 68k load balancing operations. For all implementations, only 30%-40% of load balancing operations occur before initial execution of the task, and the rest as a result of migrations enabled by the *untied* keyword.

Given the substantial amount of load balancing operations performed, we investigated whether they are actually successful in eliminating load imbalance. To that end, we recorded the number of nodes explored at each thread, shown in Figure 11 (found on the last page of the paper). Note that since ten trials were performed at each thread count, there are 10 data points shown for trials on one thread, 20 shown for trials on two threads, etc. The results for the Intel implementation (a) show good load balance, as roughly the same number of nodes (4.1M divided by the number of threads) are explored on each thread. With the Sun implementation, load balance is poor and worsens as more threads are used. Imbalance is poorer still with gcc.

Even if overhead costs were zero, speedup would be limited by load imbalance. The total running time of the program is at least the work time of the thread that does the most work. Since each task in UTS performs the same amount of work, one SHA-1 hash operation, we can easily determine that efficiency

Name	Description
wfff	work-first with FIFO local queue access, FIFO remote queue access
wffl	work-first with FIFO local queue access, LIFO remote queue access
wflf	work-first with LIFO local queue access, FIFO remote queue access
wfll	work-first with LIFO local queue access, LIFO remote queue access
cilk	wflf with priority to steal parent of current task
bff	breadth-first with FIFO task pool access
bfl	breadth-first with LIFO task pool access

Table 2: Nanos scheduling strategies. For more details see [7].

e is limited to the ratio of average work per thread to maximum work per thread. The lost efficiency $(1-e)$ for the different OpenMP task implementations is shown in Figure 5. Poor load balance by the Sun and gcc implementations severely limits scalability. Consider the 16-thread case: neither implementation can achieve greater than 40% efficiency even if overhead costs were nonexistent. On the other hand, inefficiency in the Intel implementation cannot be blamed on load imbalance.

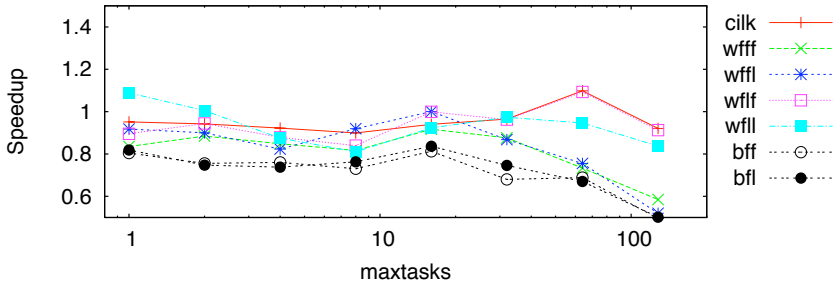
4.3 Potential for Aggregation

The thread parallel implementation reduces overhead costs chiefly by aggregating work. Threads do not steal nodes one at at time, but rather in chunks whose size is specified as a parameter. A similar method could be applied within an OpenMP run time, allowing chunks of tasks to be moved between threads at a time.

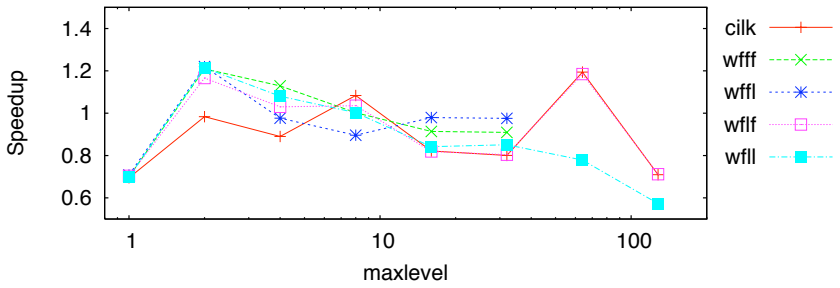
To test possible overhead reduction from aggregation, we designed an experiment in which 4M SHA-1 hashes are performed independently. To parallelize we use a loop nest in which the outer forall generates tasks. Each task executes a loop of k SHA-1 hashes. So k represents an aggregation factor. Since the outer forall has 4M / k iterations equally distributed by static scheduling, there should be little or no load balancing. Thus, performance measurements should represent a lower bound on the size of k needed to overcome overhead costs. Figure 6 shows speedup for aggregation of $k = 1$ to 100000 run using the Intel implementation. (Results for the gcc 4.4 and Sun compilers are very similar and omitted for lack of space.) Speedup reaches a plateau at $k = 50$. We could conclude that for our tree search, enough tasks should be moved at each load balancing operation to yield 50 tree nodes for exploration. Notice that for 8 and 16 threads, performance degrades when k is too high, showing that too much aggregation leads to load imbalance, i.e. when the total number of tasks is a small non-integral multiple of the number of threads.

4.4 Scheduling Strategies and Cutoffs

As mentioned in Section 2, the Mercurium compiler and Nanos run time offer a wide spectrum of runtime strategies for task parallelism. There are breadth-first schedulers with FIFO or LIFO access, and work-first schedulers with FIFO or



(a) maxtasks cutoff



(b) maxlevel cutoff

Fig. 7: UTS speedup on Opteron SMP using two threads with different scheduling and cutoff strategies in Nanos. Note that "cilk" denotes the cilk-style scheduling option in Nanos, not the cilk compiler.

LIFO local access and FIFO or LIFO remote access for stealing. There is also a "Cilk-like" work-first scheduler in which an idle remote thread attempts to steal the parent task of a currently running task. In addition, the option is provided to serialize tasks beyond a cutoff threshold, a set level of the task hierarchy (maxlevel) or a certain number of total tasks (maxtasks). Note that a maxtasks cutoff is imposed in the gcc 4.4 OpenMP 3.0 implementation, but the limit is generous at 64 times the number of threads.

Figure 7 shows the results of running UTS using two threads in Nanos with various scheduling strategies and varying values for the maxtasks and maxlevel cutoff strategies. See Table 2 for a description of the scheduling strategies represented. The breadth-first methods fail due to lack of memory when the maxlevel cutoff is used. There are 2000 tree nodes at the level just below the root, resulting in a high number of simultaneous tasks in the breadth-first regime. As shown in the graphs, we did not observe good speedup using Nanos regardless of the strategies used. Though not shown, experiments confirm no further speedup using four threads.

Limiting the number of tasks in the system (maxtasks cutoff) may not allow enough parallel slack for the continuous load balancing required. At the higher end of the range we tested in our experiments, there should be enough parallel slack but overhead costs are dragging down performance. Cutting off a few levels

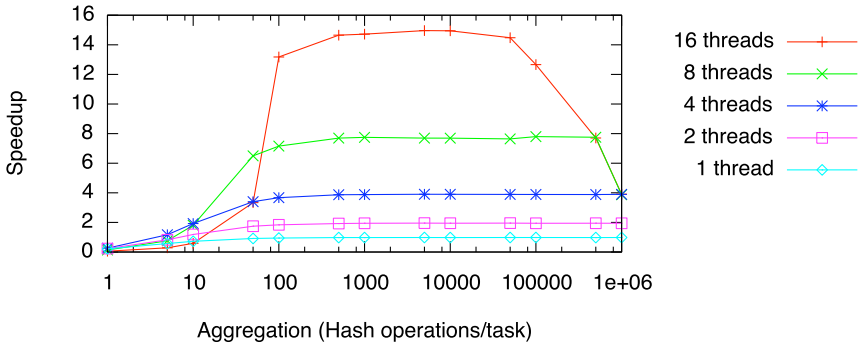


Fig. 8: Work aggregated into tasks. Speedup on Opteron SMP using cilk-style scheduling in Nanos. Results are similar using other work-first scheduling strategies.

below the root (maxlevel cutoff) leaves highly unbalanced work, since the vast majority of the nodes are deeper in the tree, and UTS trees are imbalanced everywhere. Such a cutoff is poorly suited to UTS. For T3, just a few percent of the nodes three levels below the root subtend over 95% of the tree. Adaptive Task Cutoff [8] would offer little improvement, since it uses profiling to predict good cutoff settings early in execution. UTS is unpredictable: the size of the subtree at each node is unknown before it is explored and variation in subtree size is extreme.

We also repeated aggregation experiments from Section 4.3 using Nanos. Figure 8 shows speedup using the cilk-like scheduling strategy with no cutoffs imposed. Results for other work-first schedulers is similar. Noticed that compared to the results from the same experiment using Intel compiler (Figure 6), speedup is much poorer at lower levels of aggregation with Nanos. Whereas speedup at 10 hash operations per second is about 13X with 16 threads using the Intel compiler, Nanos speedup is less than 1X.

Since the breadth-first methods struggle with memory constraints on the Opteron SMP, we tried the aggregation tests on another platform: an SGI Altix with lightweight thread support on the Nanos-supported 64-bit IA64 architecture. The Altix consists of 128 Intel Itanium2 processors running at 1.6 Ghz, each with 256kB of L2 cache and 6MB of L3 cache. We installed the Mecurium 1.2.1 research compiler with Nanos 4.1.2 run time and the Intel icc 11.0 compiler. Even using the breadth-first schedulers and no cutoffs, the tasks are able to complete without exhausting memory. Figure 9 shows experiments performed on two threads of the Altix. The Intel implementation outperforms Nanos at fine-grained aggregation levels. Among the Nanos scheduling options, the work-first methods are best.

4.5 The *if()* Clause

The OpenMP task model allows the programmer to specify conditions for task serialization using the *if()* clause. To evaluate its impact, we used the *if()* clause in a modified version of our task parallel implementation so that only $n\%$ percent

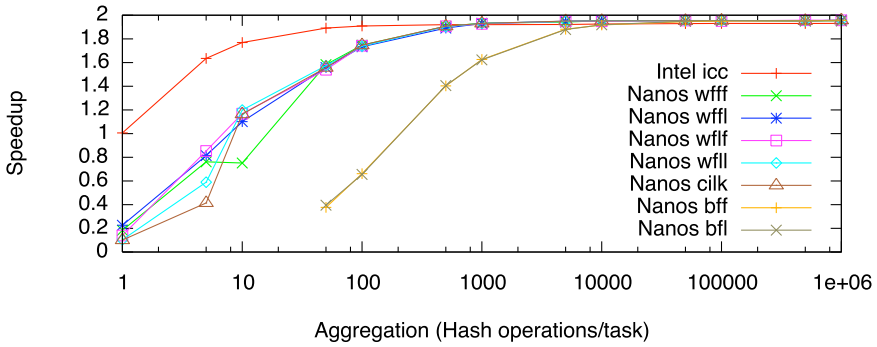


Fig. 9: Work aggregated into tasks. Speedup on an SGI Altix for 4M hash operations performed; work generated evenly upon two threads. The various Nanos scheduling strategies are used without cutoffs, and Intel icc is shown for comparison. Note that "cilk" denotes the cilk-style scheduling option in Nanos, not the cilk compiler.

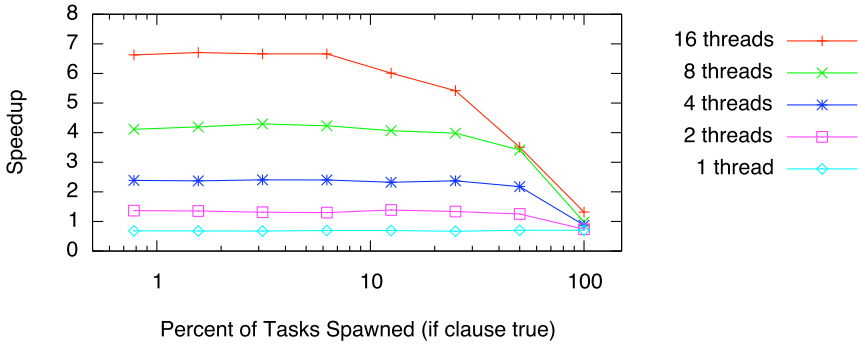


Fig. 10: UTS Speedup on Opteron SMP using the Intel OpenMP 3.0 task implementation with user-defined inlining specified using the *if()* clause.

of the tree nodes are explored in new tasks while the rest are explored in place. We varied n exponentially from less than 0.01% to 100%. Figure 10 shows the results on the Opteron SMP using the Intel compiler. Using the *if()* clause to limit the number of tasks seems to improve speedup. However, Figure 1 showed similar speedups using the same compiler and UTS implementation but with no *if()* clause. Why would setting $n = 100\%$ not yield the same results? We suspect that the use of the *if()* clause may disable a default internal cutoff mechanism in the Intel run time system.

5 Conclusions

Explicit task parallelism provided in OpenMP 3.0 enables easier expression of unbalanced applications. Consider the simplicity and clarity of the task parallel

UTS implementation. However, there is clearly room for further improvement in performance for applications with challenging demands such as UTS.

Our experiments suggest that efficient OpenMP 3.0 run time support for very unbalanced task graphs remains an open problem. Among the implementations tested, only the Intel compiler shows good load balancing. Its overheads are also lower than other implementations, but still not low enough to yield ideal speedup. Cilk outperforms all OpenMP 3.0 implementations; design decisions made in its development should be examined closely when building the next generation of OpenMP task run time systems. A key feature of Cilk is its on-demand conversion of serial functions (fast clone) to concurrent (slow clone) execution. The "Cilk-style" scheduling option in Nanos follows the work stealing strategy of Cilk, but decides before task execution whether to inline a task or spawn it for concurrent execution.

We cannot be sure of the scheduling mechanisms used in the commercial OpenMP 3.0 implementations. The gcc 4.4 implementation uses a task queue and maintains several global data structures, including current and total task counts. Contention for these is a likely contributor to overheads seen in our experiments. Another problematic feature of the gcc OpenMP 3.0 implementation is its use of barrier wake operations upon new task creation to enable idle threads to return for more work. These operations are too frequent in an applications such as UTS that generates work irregularly. Even with an efficient barrier implementation, they may account for significant costs.

Experiments using several different scheduling strategies with cutoffs also show poor performance. Unbalanced problems such as UTS are not well suited to cutoffs because they make it difficult to keep enough parallel slack available. Aggregation of work should be considered for efficient load balancing with reduced overhead costs. Further work is needed to determine other ways in which OpenMP 3.0 run time systems could potentially be improved and whether additional information could be provided to enable better performance.

While the UTS benchmark is useful as a benchmarking and diagnostic tool for run time systems, many of the same problems it uncovers impact real world applications. Combinatorial optimization and enumeration lie at the heart of many problems in computational science and knowledge discovery. For example, protein design is a combinatorial optimization problem in which energy minimization is used to evaluate many combinations of amino acids arranged along the backbone to determine whether a desired protein geometry can be obtained [15]. An example of an enumeration problem in knowledge discovery is subspace clustering, in which subsets of objects that are similar on some subset of features are identified [16]. Another example is the Quadratic Assignment Problem (QAP) at the heart of transportation optimization. These sorts of problems typically 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.

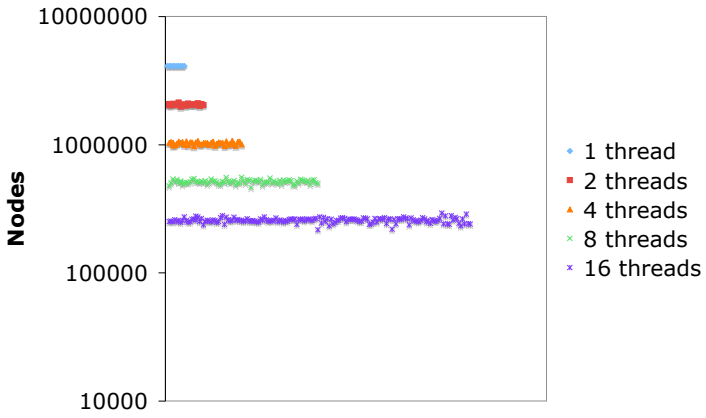
Evaluation on a wider range of applications is needed to determine the shared impact of the compiler and run time issues that UTS has uncovered. One issue that we have not addressed in our experiments is locality. UTS models applications in which a task only requires a small amount data from its parent and no other external data. We anticipate future work in which we consider applications with more demanding data requirements.

6 Acknowledgements

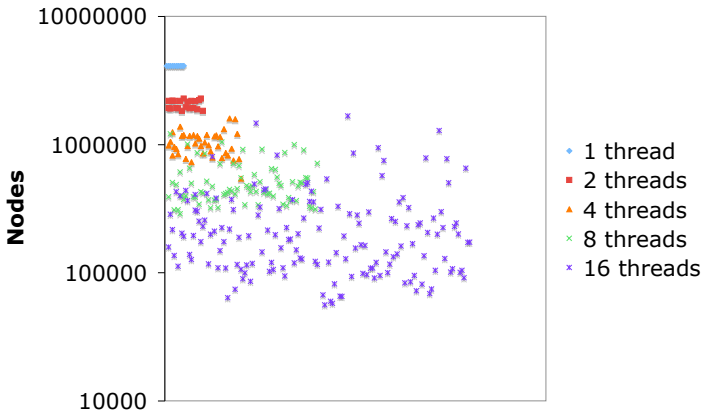
Stephen Olivier is funded by a National Defense Science and Engineering Graduate Fellowship. The Opteron SMP used in this paper is a part of the BASS cluster purchased with funding from the National Institutes of Health's National Center for Research Resources, through their High-End Instrumentation program award number NIH 1S10RR023069-01 and administered by the UNC Department of Computer Science. The authors would like to thank the anonymous reviewers for their insightful comments, many of which have led to important improvements in the paper.

References

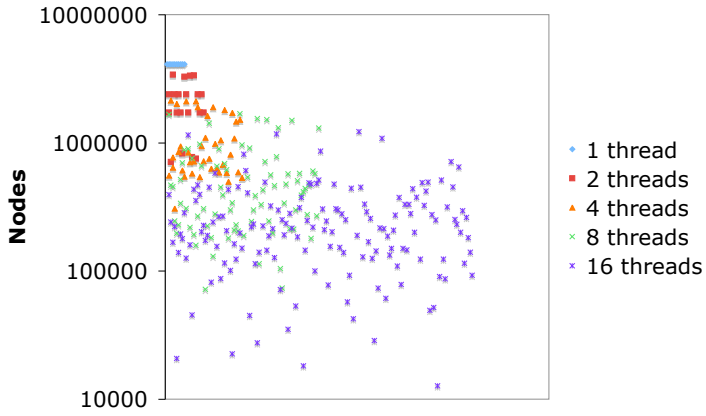
1. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.0 (May 2008)
2. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An unbalanced tree search benchmark. In Almási, G., Cascaval, C., Wu, P., eds.: Proc. LCPC 2006. Volume 4382 of LNCS., Springer (2007) 235–250
3. 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
4. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: PPoPP '95: Proc. 5th ACM SIGPLAN symp. Princ. Pract. Par. Prog. (1995)
5. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proc. 35th Ann. Symp. Found. Comp. Sci. (Nov. 1994) 356–368
6. Mohr, E., Kranz, D.A., Robert H. Halstead, J.: Lazy task creation: a technique for increasing the granularity of parallel programs. In: LFP '90: Proc. 1990 ACM Conf. on LISP and Functional Prog., New York, NY, USA, ACM (1990) 185–197
7. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP task scheduling strategies. In Eigenmann, R., de Supinski, B.R., eds.: IWOMP '08. Volume 5004 of LNCS., Springer (2008) 100–110
8. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: SC '08: Proceedings of the 2008 ACM/IEEE Conf. on Supercomputing, Piscataway, NJ, USA, IEEE Press (2008) 1–11
9. Ibanez, R.F.: Task chunking of iterative constructions in openmp 3.0. In: First Workshop on Execution Environments for Distributed Computing. (July 2007)
10. Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., Petersen, P.: Compiler support of the workqueuing execution model for Intel(R) SMP architectures. In: European Workshop on OpenMP (EWOMP'02). (2002)
11. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In Adve, V.S., Garzarán, M.J., Petersen, P., eds.: LCPC. Volume 5234 of LNCS., Springer (2007) 63–77
12. Teruel, X., Unnikrishnan, P., Martorell, X., Ayguadé, E., Silvera, R., Zhang, G., Tiotto, E.: OpenMP tasks in IBM XL compilers. In: CASCON '08: Proc. 2008 Conf. of Center for Adv. Studies on Collaborative Research, ACM (2008) 207–221
13. Free Software Foundation, I.: GCC, the GNU compiler collection. http://www.gnu.org/software/gcc/
14. Eastlake, D., Jones, P.: US secure hash algorithm 1 (SHA-1). RFC 3174, Internet Engineering Task Force (September 2001)
15. Baker, D.: Proteins by design. The Scientist (July 2006) 26–32
16. Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P.: Automatic subspace clustering of high dimensional data. Data Min. Knowl. Discov. **11**(1) (2005) 5–33



(a) Intel icc



(b) Sun



(c) gcc 4.4

Fig. 11: UTS on Opteron SMP: Number of nodes explored at each thread.