Int J Parallel Prog (2010) 38:341–360
DOI 10.1007/s10766-010-0140-7

Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs

Stephen L. Olivier · Jan F. Prins

Received: 27 April 2010 / Accepted: 27 April 2010 / Published online: 1 June 2010

Abstract The UTS benchmark is used to evaluate the expression and performance of 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. Expressiveness and scalability are compared for OpenMP 3.0, Cilk, Cilk++, Intel Thread Building Blocks, as well as an OpenMP implementation of the benchmark without tasks that performs all scheduling, load balancing, and termination detection explicitly. Current OpenMP 3.0 run time implementations generally exhibit poor behavior on the UTS benchmark. We identify inadequate load balancing strategies and overhead costs as primary factors contributing to poor performance and scalability.

Keywords Task parallelism · Run time systems · Load balancing · Performance evaluation · OpenMP

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

S. L. Olivier (✉) · J. F. Prins
Department of Computer Science, University of North Carolina at Chapel Hill, CB 3175, Chapel Hill, NC 27599, USA
e-mail: olivier@unc.edu

J. F. Prins
e-mail: prins@unc.edu

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 several different OpenMP 3.0 compiler and run time implementations (Intel icc 11, gcc 4.4, Mercurium 1.2, SunStudio 12). For comparison we also examine the performance of the UTS benchmark using Cilk [3] , Intel Cilk++ [4], Intel Thread Building Blocks [5]. We also compare with an OpenMP implementation without tasks that performs all scheduling, load balancing, and termination detection explicitly [6]. Throughout this paper we will refer to the latter as the custom 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.

UTS presents a stress test to the load balancing capabilities of the run time system, providing insight into OpenMP 3.0 implementations. It is not intended as a negative general assessment of such implementations; several more balanced and predictable task parallel applications have been shown to scale using OpenMP 3.0 tasks [7].

This paper extends work initially presented in [8]. It is organized as follows: Sect. 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 Sect. 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 [3,9]. 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 other tasks, and determining global completion of all tasks.

To decrease task execution overheads, various *coarsening strategies* are followed to aggregate multiple tasks together, or to execute serial versions of tasks that elide synchronization support and interaction with the runtime system 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 [10]. A lazy task creation approach, developed for parallel implementations of functional languages [11], 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 [12]. 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 parallel execution threads. The study in [12] 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 [13]. 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 [14]. 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 extension of OpenMP for task parallelism [15]. In addition to the `task` construct, a `taskq` construct defined queues of tasks explicitly. A noteworthy feature was support for reductions among the results of tasks in a task queue. Early evaluations of OpenMP tasking made comparisons to Intel workqueuing, showing similar performance on a suite of seven applications [16].

An extension of the Nanos Mercurium research compiler and run time [16] 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 [12]. That study concluded that in situations where each task is *tied*, i.e. fixed to the thread on which it first executes (though possibly different from the thread executing its parent task), 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 the programmer wishes to disallow migration of that task during its execution, e.g. 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 [17]. The recent version 4.4 release of the GNU compilers [18] includes 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.

In addition to OpenMP 3.0, there are currently several other task parallel languages and libraries available to developers, Microsoft Task Parallel Library [19] for Windows, Intel Thread Bulding Blocks (TBB) [5], and Intel Cilk++ [4], a younger sibling of Cilk based on C++ rather than C. We will use TBB and Cilk++, along with Cilk, as comparison points for our performance evaluation with OpenMP 3.0.

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 root. 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 [20] on the parent description together with 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 OpenMP Implementations of UTS

We have developed three implementations of UTS in OpenMP. Two implementations use the task parallel support in OpenMP 3.0, while the third explicitly implements load balancing between threads in an OpenMP parallel region.

3.1.1 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. Each task consists of a function that returns the count of nodes in the subtree rooted at its node, recursively creating tasks to count the subtrees rooted at each of its children. In order to correctly accumulate the results, the `partialcount` array is maintained in the function to hold the result of the subtasks. The function must then stay on the stack and wait for all descendent tasks to complete using a `taskwait` statement before manually combining the results to arrive at the sum to return. A sketch of the implementation follows below:

```
long Generate_and_Traverse(Node* parentNode,
 int childNumber) {
  Node currentNode = generateID(parentNode, childNumber);
  int numChildren =  m with prob q, 0 with prob 1-q
```

```
  long partialCount[numChildren], nodeCount = 1;
  for (i = 0; i < numChildren; i++) {
    #pragma omp task untied firstprivate(i)
    partialCount[i] = Generate_and_Traverse(currentNode,
     i);
  }
  #pragma omp taskwait
  for (i = 0; i < numChildren; i++)
    nodeCount += partialCount[i];
  return nodeCount;
}
```

It is safe to use untied tasks here because task suspension and migration do not impact correctness of this implementation.

3.1.2 Task Parallel Implementation With Threadprivate Storage

We devised an alternate task parallel OpenMP implementation that maintains per-thread (threadprivate in OpenMP parlance) partial results which are combined in O(p) time only at the end of the computation.

```
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 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 each thread. Within the parallel region a single thread creates tasks to count the subtrees below the root. All threads synchronize at a barrier when all tasks are complete.

We also observed correct results and similar performance when this implementation is modified to use untied tasks. It is technically unsafe to use untied tasks, since a task could migrate amid the update to nodeCount.

3.1.3 Customized Parallel Implementation With Explicit Load Balancing

Unlike the task parallel implementation of UTS, the customized 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, values are combined at end
      int numChildren =  m with prob q, 0 with prob 1-q
      for (i = 0; i < numChildren; i++) {
          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. The elided work-stealing code is not trivial; for details see [6].

3.2 Implementations in Other Task Parallel Frameworks

Let us consider the implementation of UTS in other task parallel languages and libraries, first in terms of the how the implementation is expressed, and later, in Sect. 4, in terms of performance and scalability. Recall the two different task parallel implementations from Sect. 3.1: The implementation without threadprivate (Sect. 3.1.1) requires explicit level-wise synchronization and per-task accumulation of partial results, while the implementation with threadprivate (Sect. 3.1.2) does not. These differences also arise when comparing other task parallel frameworks, as we shall see.

3.2.1 Cilk Implementation

We created a Cilk implementation of UTS which is close to the OpenMP 3.0 task implementation without threadprivate storage. 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.

```
cilk long Generate_and_Traverse(Node* parentNode,
 int childNumber) {
  long nodeCount = 1;
  inlet void accumulate (long result) {
     count += result;
  }
```

```
  Node currentNode = generateID(parentNode, childNumber);
  int numChildren =  m with prob q, 0 with prob 1-q
  for (i = 0; i < numChildren; i++)
      accumulate(spawn Generate_and_Traverse(currentNode,
       i));
  sync;
  return count;
}
```

Execution is started by creating tasks to explore the children of the root node, followed by a `sync` statement.

3.2.2 Cilk++ Implementation

Cilk++ provides reduction facilities through objects called reducers. Synchronization for safe concurrent access to the reducer object is managed by the Cilk++ run time, as explained in detail in [21]. The `cilk_sync` is required only because Cilk++ does not provide a construct to wait on all tasks, a facility provided in OpenMP by the barrier construct.

```
cilk::hyperobject<cilk::reducer_opadd<long> > nodeCount;

void Generate_and_Traverse(Node* parentNode,
 int childNumber) {
  Node currentNode = generateID(parentNode, childNumber);
  nodeCount()++;
  int numChildren =  m with prob q, 0 with prob 1-q
  for (i = 0; i < numChildren; i++)
      cilk_spawn Generate_and_Traverse(currentNode, i);
  cilk_sync;
}
```

Execution is started by creating tasks to explore the children of the root node, followed by a `cilk_sync` statement.

3.2.3 Thread Building Blocks Implementation

Our final comparison point in the evaluation is an Intel Thread Building Blocks (TBB) implementation. TBB requires the declaration of a new class extending the task class, and the creation of task object instances. A member function executes the work of the task. As in the task parallel OpenMP implementation, partial results must be collected and summed manually. Level-wise synchronization is required via the function wait_for_all().

```
class Generate_and_Traverse: public task {
public:
  Node *parentNode;
  int childNumber;
  long* const nodeCount;
  Generate_and_Traverse(Node *parentNode_,
   int childNumber_, long* nodeCount_) :
    parentNode(parentNode_), childNumber(childNumber_),
     nodeCount(nodeCount_) {}
task* execute() {
  long partialCount[numChildren];
  parTreeSearchTask* tArr[numChildren];
  Node currentNode = generateID(parentNode, childNumber);
  int numChildren =   m with prob q, 0 with prob 1-q
  for (i = 0; i < numChildren; i++) {
     partialCount[i] = 1;
     tArr[i] = new(allocate_child())
      Generate_and_Traverse
       (currentNode, childNumber, &partialCount[i]);
     spawn(*tArr[i]);
  }
  set_ref_count(numChildren+1);
  wait_for_all();
  for (i = 0; i < numChildren; i++)
     *nodeCount += partialCount[i];
 }
}
```

Execution is started by creating tasks to explore the children of the root node, followed by a wait_for_all() and final summation.

3.3 Code Comparison Summary

Table 1 summarizes the comparison of the different task parallel implementations. Note the similarity between the implementations using OpenMP tasks without threadprivate and Intel TBB. Inlets and reducers make Cilk and Cilk++ easy platforms for

Table 1 Comparison of task parallel UTS implementations

	OpenMP tasks without threadprivate	OpenMP tasks with threadprivate	Cilk	Intel Cilk++	Intel TBB
Explicit level-wise syncronization	Yes	No	Yes	Yes	Yes
Accumulation of partial results	Yes	No	No	No	Yes

UTS, while the need for task objects and significant task management make UTS implementation in TBB cumbersome. Beyond the perspective of our benchmark, OpenMP differs from the other languages and libraries in terms of memory model and data scoping rules, contributing some overheads seen in our evaluation (Sect. 4.2.1). As a tradeoff, these features benefit a host of other applications not covered in this paper.

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 the Intel icc 11.0 compiler, SunStudio 12 with Ceres C 5.10, and gcc 4.4.0. We also installed the Mercurium 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, using the gcc 4.3 compiler as a back end. On the Opteron, we also used Intel Cilk++ 1.0 (based on gcc 4.2.4) and Intel Thread Building Blocks 2.2 (compiled and used with Intel icc 11.0), The -O3 option (or equivalent) is always used. Unless otherwise noted, reported results represent the average of 10 trials.

For a few results in Sect. 4.4 of the paper, we used an SGI Altix using Intel icc and Mercurium. Details for that system are presented in that section.

4.1 Sequential and Parallel Performance on UTS

Table 2 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. Results in Table 2 are for our most efficient task parallel serial implementation, similar to the task implementation without threadprivate in that results are passed as return values. No OpenMP or other parallel directives are used or enabled in the serial implementation, and it is faster than those in [8].

Figure 1 shows the speedup gained on the task parallel implementations using OpenMP 3.0, Cilk, Cilk++, and TBB, as measured against the sequential performance data given in Table 2. Reported gcc and Sun OpenMP results are from the implementation with threadprivate; results for the implementation without threadprivate are very similar: We observe no speedup from Sun Studio and gcc. Cilk, Cilk++, and TBB outperform the Intel OpenMP task implementation, and all three show improved speedup

Table 2 Sequential performance on the Opteron SMP (Millions of tree nodes explored per second)

Implementation	Intel icc 11.0	Sun Ceres 5.10	gcc 4.4.0	Mercurium 1.2.1
Serial Rate	3.78	3.57	3.35	2.05

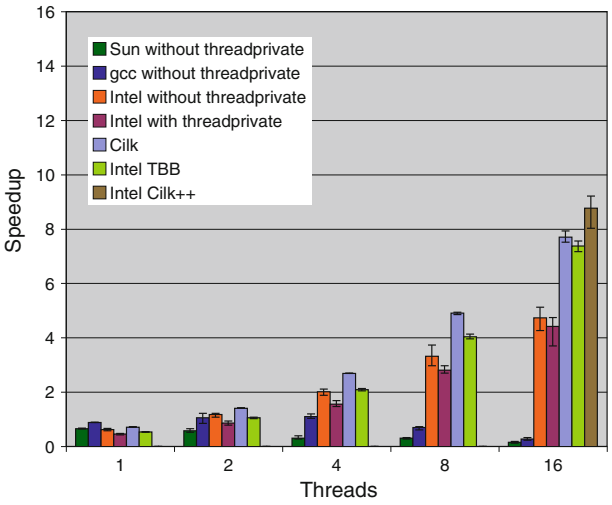

Fig. 1 UTS using several task implementations on OpenMP 3.0, Cilk, Thread Building Blocks, and Cilk++: Speedup on 16-way Opteron SMP. See Fig. 8 and Sect. 4.4 for results using the Nanos run time for OpenMP 3.0

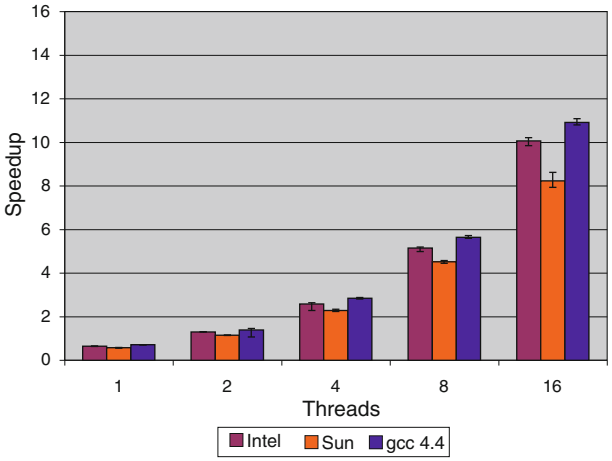

Fig. 2 UTS using custom OpenMP parallel implementation without tasks: speedup on 16-way Opteron SMP. Work stealing granularity is a user-supplied parameter. The optimal value (64 tree nodes transferred per steal operation) was determined by manual tuning and used in these experiments

as up to 16 cores are used. The Intel Cilk++ implementation ran to completion only with 16 threads due to limitations on the number of nested tasks.[1] No task parallel implementation averaged more than 9X speedup. Figure 2 shows the speedup, up to 11X, using the customized parallel implementation with manual load balancing.

[1] We have confirmed with the developers of Cilk++ that this is not a fundamental limitation of the Cilk++ run time and that future releases could increase the number of possible nested tasks or even allow the user to set this limit through a parameter to the run time.

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 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.

4.2.1 Overhead Costs

Even when only a single thread is used, there are some overhead costs incurred using OpenMP. For task parallel implementations of UTS, single thread efficiency ranges from 45 to 88%. Overhead costs are not unique to the scheduling of tasks, though more pronounced. For OpenMP single thread execution of a loop of 4M iterations each performing one SHA-1 hash, scheduled dynamically one iteration per chunk, efficiency ranges from 92% to 97% versus sequential execution.

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, 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 is much higher still.

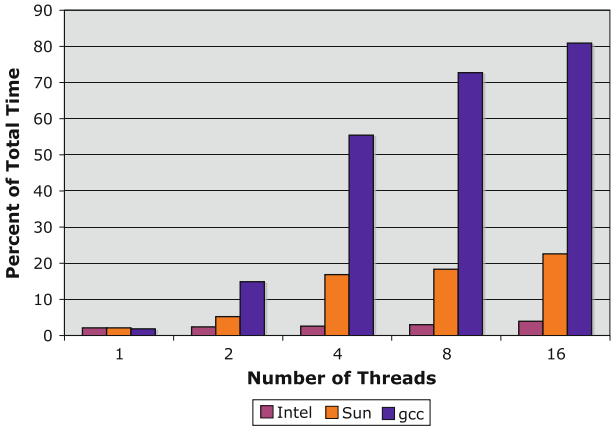

Fig. 3 Overheads (time not calculating SHA-1 hash) on UTS using 100 repetitions of the SHA-1 hash per tree node

4.2.2 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 4.1 M node tree (T3), indicating nearly 450 k load balancing operations performed by the Intel and Sun run times per trial using 8 threads. That comprises 11% of the 4.1 M tasks generated. In contrast, gcc only performs 68k load balancing operations. By running trials of the UTS task parallel OpenMP implementation both with all untied tasks and with all tied tasks, we have confirmed that load balancing operations occur before initial execution of the task. We do not observe task migrations as enabled by the *untied* keyword as determined by querying the thread number at multiple points during each task's execution. This is not the case for the Nanos run time (Sect. 4.4). Performance is also unaffected by the use of tied vs. untied.

Given the substantial number 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 Fig. 5. 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 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 Fig. 6. 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

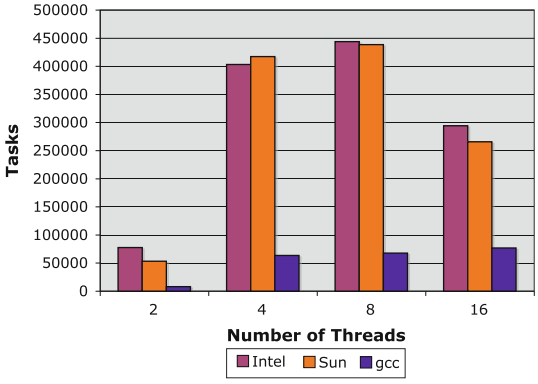

Fig. 4 Number of tasks started on different threads from their parent tasks or migrating during task execution, indicating load balancing efforts. 4.1 M tasks are performed in each program execution

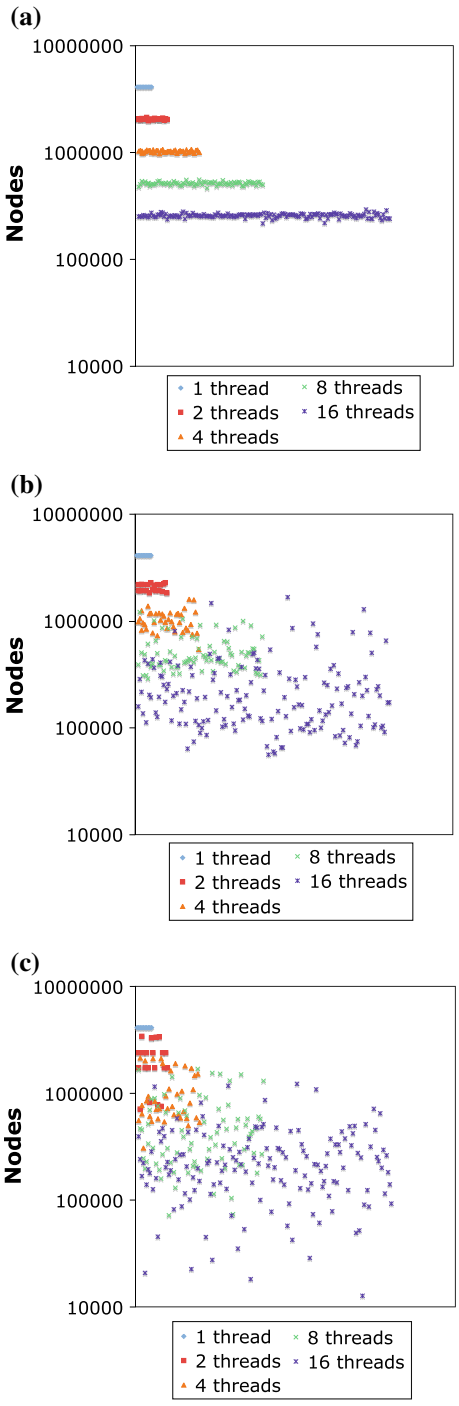

Fig. 5 UTS on Opteron SMP: Number of nodes explored at each thread. **a** Intel icc. **b** Sun. **c** gcc 4.4

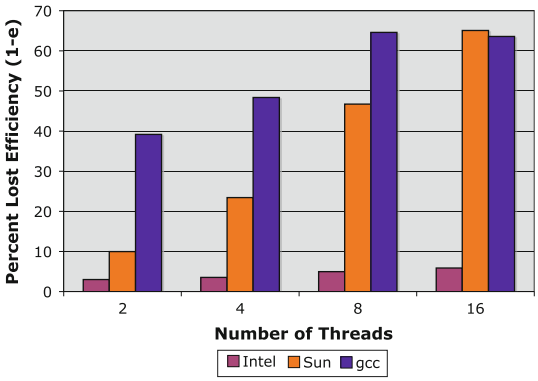

Fig. 6 UTS on Opteron: lost efficiency due to load imbalance

costs were nonexistent. On the other hand, inefficiency in the Intel implementation cannot be blamed on load imbalance.

4.3 Potential for Aggregation

The custom 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 for loop 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 7 shows speedup for aggregation of $k = 1$ to 1000000 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 Sect. 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 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

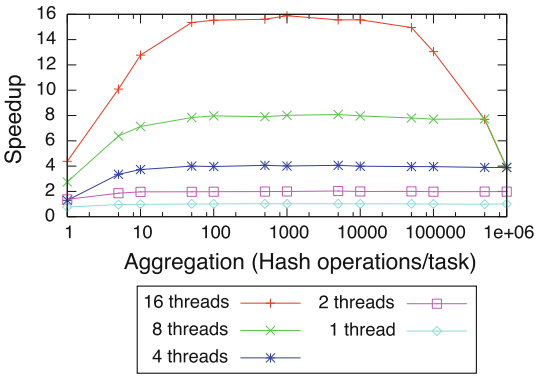

Fig. 7 Work aggregated into tasks. Speedup on Opteron SMP using the Intel OpenMP tasks implementation. Results are similar using the gcc 4.4 and Sun compilers, though slightly poorer at the lower aggregation levels

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 8 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 3 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

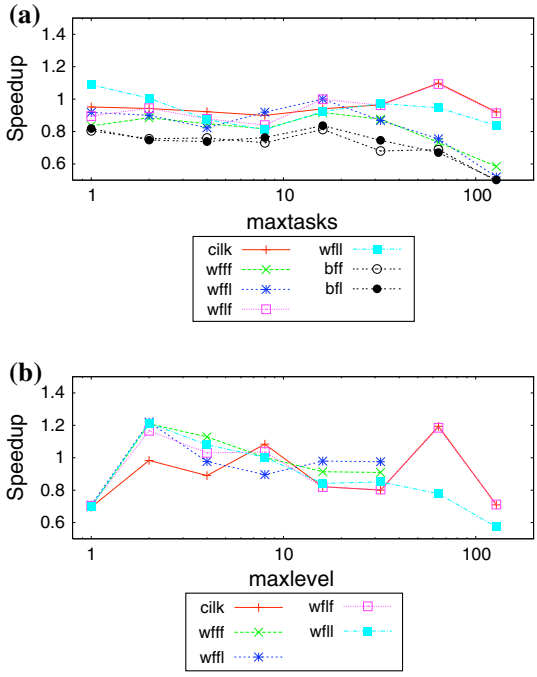

Fig. 8 UTS speedup on Opteron SMP using two threads with different scheduling and cutoff strategies in Nanos. Note that "cilk" denotes the Cilk-style scheduling option in Nanos, not the Cilk compiler. **a** Maxtasks cutoff. **b** Maxlevel cutoff

Table 3 Nanos scheduling strategies. For more details see [12]

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

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. Unlike the other OpenMP 3.0 run times, we do observe migrations of untied tasks during task execution using Nanos; work-first schedulers migrate untied tasks quite frequently, as expected.

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 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 [13] 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 Sect. 4.3 using Nanos. Figure 9 shows speedup using the cilk-like scheduling strategy with no cutoffs imposed. Results

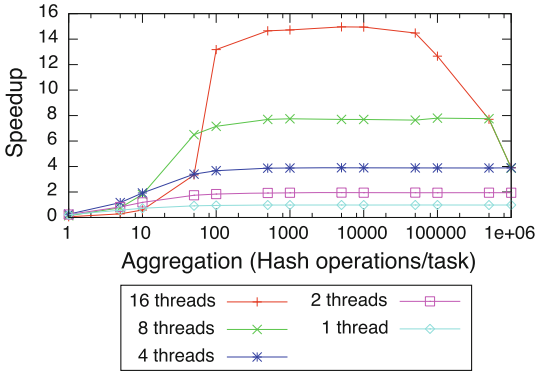

Fig. 9 Work aggregated into tasks. Speedup on Opteron SMP using Cilk-style scheduling in Nanos. Results are similar using other work-first scheduling strategies

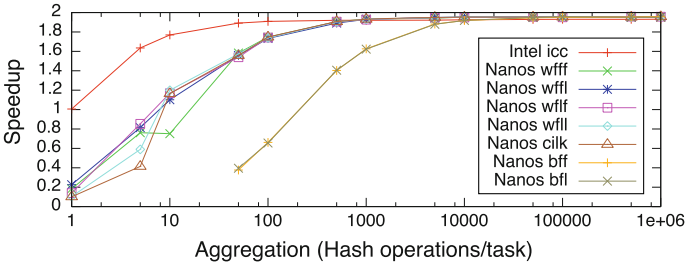

Fig. 10 Work aggregated into tasks. Speedup on an SGI Altix for 4M hash operations performed; work generated evenly upon two threads. The various Nanos scheduling strategies are used without cutoffs, and Intel icc is shown for comparison. Note that "cilk" denotes the Cilk-style scheduling option in Nanos, not the Cilk compiler

for other work-first schedulers is similar. Notice that compared to the results from the same experiment using Intel compiler (Fig. 7), 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 10 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 $n\%$ percent of the tree nodes are explored in place while the rest are explored in new tasks. We varied n exponentially from less than 0.01% to 100%. Figure 11 shows the results on the Opteron SMP using the Intel compiler. Using the *if()* clause to limit the number of tasks actually reduces speedup. The *if()* clause reduces available parallelism and limits the scheduling options of the run time, which can be detrimental for a problem such as UTS.

5 Conclusions

Explicit task parallelism provided in OpenMP 3.0 enables easier expression of unbalanced applications as can be seen from the simplicity and clarity of the task parallel

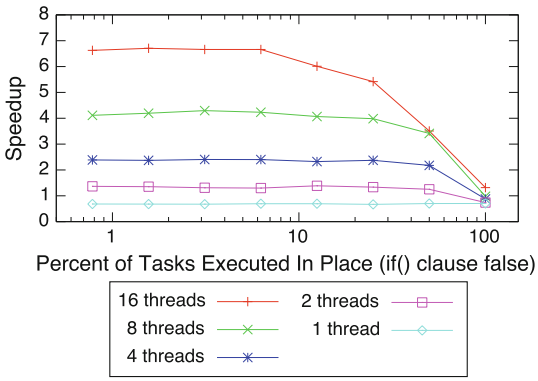

Fig. 11 UTS Speedup on Opteron SMP using the Intel OpenMP 3.0 task implementation with user-defined inlining specified using the *if()* clause

UTS implementations. 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 on UTS. Its overheads are also lower than other implementations, but still not low enough to yield ideal speedup. Cilk, Cilk++, and TBB outperform all OpenMP 3.0 task parallel implementations; design decisions made in their 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 generate work irregularly. Even with an efficient barrier implementation, they may account for significant costs. A notable feature of TBB is its use of low-level primitives for thread management and synchronization, and Cilk++ offers the helpful reducer construct.

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 do impact real world applica-

tions. These include evaluation of amino acid combinations for protein design [22], subspace clustering for knowledge discovery [23], and the Quadratic Assignment Problem (QAP) at the heart of transportation optimization. We reiterate that several applications with less extreme imbalance and unpredictability have been shown to scale using OpenMP 3.0 tasks [7]. Some features of the OpenMP memory model and data scoping rules that differ from the other task parallel languages and contribute to overheads are beneficial to those applications.

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. Given the NUMA layout of memory between chips, scheduling decisions should be informed by explicit knowledge of locality. If, as in many multicore designs, cores on the same chip share a cache, it will also be advantageous to co-schedule tasks with the same working set onto threads running on cores in the same chip. In future work we consider applications with more demanding data requirements, including applications such as adaptive fast multipole n-body, collision detection, and sorting.

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 funded by the National Institutes of Health, award number NIH 1S10RR023069-01. We are grateful for helpful comments from Alejandro Duran and the reviewers.

References

1. OpenMP Architecture Review Board: OpenMP API, 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.) Proceedings of LCPC 2006. vol. 4382 of LNCS., pp. 235–250. Springer (2007)
3. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings 1998 SIGPLAN Conf. Prog. Lang. Design Impl. (PLDI '98), pp. 212–223 (1998)
4. Intel Corp.: Intel Cilk++ SDK. http://software.intel.com/en-us/articles/intel-cilk/
5. Kukanov, A., Voss, M.: The foundations for scalable multi-core software in intel threading building blocks. Intel Technol J **11**(4) (November 2007)
6. Olivier, S., Prins, J.: Scalable dynamic load balancing using UPC. In: ICPP '08: Proceedings of 37th Intl. Conf. on Par. Processing, pp. 123–131. IEEE (Sept. 2008)
7. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona OpenMP Tasks Suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proceedings of 38th Intl. Conf. on Par. Processing (ICPP'09), pp. 124–131. IEEE Computer Society, Vienna, Austria (September 2009)
8. Olivier, S.L., Prins, J.F.: Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In: IWOMP '09: Proceedings 5th International Workshop on OpenMP, pp. 63–78. Springer-Verlag, Berlin, Heidelberg (2009)
9. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: PPoPP '95: Proceedings of 5th ACM SIGPLAN symp. Princ. Pract. Par. Prog. (1995)
10. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proceedings of 35th Ann. Symp. Found. Comp. Sci., pp. 356–368. (Nov. 1994)
11. Mohr, E., Kranz, D.A., Robert, H., Halstead, J.: Lazy task creation: a technique for increasing the granularity of parallel programs. In: LFP '90: Proceedings 1990 ACM Conf. on LISP and Functional Prog., pp. 185–197. ACM, New York, NY, USA (1990)

12. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP task scheduling strategies. In: Eigenmann, R., de Supinski, B.R., (eds.) IWOMP '08. Vol. 5004 of LNCS., pp. 100–110. Springer (2008)
13. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: SC08: ACM/IEEE Supercomputing 2008, Piscataway, pp. 1–11. IEEE Press, NJ, USA (2008)
14. Ibanez, R.F.: Task chunking of iterative constructions in OpenMP 3.0. In: First Workshop on Execution Environments for Distributed Computing. (July 2007)
15. Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., Petersen, P.: Compiler support of the workqueuing execution model for Intel SMP architectures. In: European Workshop on OpenMP (EWOMP'02). (2002)
16. 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. vol. 5234 of LNCS., pp. 63–77. Springer (2007)
17. 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, pp. 207–221. ACM (2008)
18. Free Software Foundation Inc.: GCC, The GNU Compiler Collection. http://www.gnu.org/software/gcc/
19. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. SIGPLAN Notices: OOPSLA'09 **44**(10), 227–242 (2009)
20. Eastlake, D., Jones, P.: US secure hash algorithm 1 (SHA-1). RFC 3174, Internet Engineering Task Force (September 2001)
21. Frigo, M., Halpern, P., Leiserson, C.E., Lewin-Berlin, S.: Reducers and other Cilk++ hyperobjects. In: Proc. SPAA '09, ACM Press (August 2009)
22. Baker, D.: Proteins by design. The Scientist, 26–32 (July 2006)
23. Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P.: Automatic subspace clustering of high dimensional data. Data Min. Knowl. Discov. **11**(1), 5–33 (2005)