OpenMP 3.0 Tasking Implementation in OpenUH*

Cody Addison
Texas Instruments Incorporated, Stafford TX 77477, USA
James LaGrone Lei Huang Barbara Chapman
University of Houston, Houston TX 77004, USA

Abstract

As multicore technology dominates the processor market, new methodologies are being explored to exploit the parallelism inherent to these architectures and shared memory programming models are gaining in popularity. The ratification of the OpenMP 3.0 API has provided compiler developers with another challenge as the multicore revolution reshapes the landscape in scientific computing. The introduction of explicit tasking in this latest revision of the de facto *standard for shared memory programming introduces new capabilities for parallel programming. Tasking abilities in OpenMP now allow irregular applications with pointer based data and recursive algorithms to be executed in parallel, as well as providing alternative parallelization techniques for traditional loop-centric codes. This paper outlines the implementation of OpenMP 3.0 tasking features in OpenUH, a branch of Open64 compiler suite.*

1 Introduction

Multicore architectures have brought parallel programming to the masses. Parallel programming was previously the domain of scientific applications running on large clusters. Multicore technology is now providing at least two cores in laptops and dozens of cores per processor in research systems. Hundreds of cores per processor will be available in the near future. Mainstream applications are moving to the parallel world, which prompts the reevaluation of current parallel programming models. Developers must apply a parallel programming model to applications to exploit the available parallelism. However, most parallel programming models for shared memory architectures have focused on scientific applications using large arrays and loop-level parallelism. While this class of applications is prevalent in high performance computing, they are not representative of applications expected to be useful on multicore architectures.

OpenMP is the *de facto* standard in shared memory programming, which originally targeted scientific applications. However, it does not apply well to applications which employ certain types of irregular parallelism such as, recursion, pointer chasing, or large load imbalances. The OpenMP Architecture Review Board's recent release of the OpenMP 3.0 API [15] shifts its paradigm with the adoption of a new tasking model. Where the previous standards relied on worksharing constructs and loop-centric programming to exploit parallelism, the new standard allows the developer to dynamically create asynchronous units of work to be scheduled by the runtime. This is a powerful feature that has the potential to allow OpenMP to be used to write parallel code for a wide variety of applications.

In the new model, it is the programmer's responsibility to expose the opportunities for parallelism by denoting the tasks and their synchronization points. It is the system's responsibility to manage and schedule the tasks to exploit the maximum amount of parallelism and therefore performance. This allows the developer to focus on the higher level components of parallel programming, such as program decomposition, without being concerned with lower level constructs like threads and locks.

The focus of this work is the integration the new OpenMP tasking model into the OpenUH compiler framework. We focus on the three major components — compiler frontend support for tasks, compiler translation of tasks, and extensions to the runtime library. After an overview of tasking features in OpenMP 3.0 (Section 2) and some related work (Section 3), we present our imple-

*This work was supported by DOE Pmodels project under grant DE-FC02-06ER25759 and the National Science Foundation under contracts CCF-0833201 and CCF-0702775.

mentation (Section 4), evaluate it (Section 5), and conclude with ideas for future work (Section 6).

2 OpenMP 3.0 Tasking

The OpenMP Architecture Review Board ratified the latest standard of its shared memory programming model in May of last year. The acceptance of this new standard is evident by the numerous commercial compilers touting compliance within months of ratification. Recognizing that OpenMP already supports *implicit* tasks in the form of parallel regions, the new standard extends the programming model by allowing *explicit* tasks. This new tasking capability addresses the previous difficulties in parallelizing applications employing recursive algorithms or pointer based data structures. The new constructs, `omp task` and `omp taskwait` allow the designation and synchronization of tasks.

OpenMP defines a *task* as a specific instance of executable code and its data environment. A *task region* consists of all the code encountered during the execution of a task and is created when a thread executes the task. When a `parallel` construct is encountered, a set of *implicit* tasks is created, one task per thread. When a thread encounters a `task` construct, an *explicit* task is created. The execution of an explicit task may be immediate or delayed. By default, the execution of a task is *tied* to a thread, whereby its execution cannot resume on another thread if suspended. Alternatively, explicit tasks may be *untied*, whereby its execution may be suspended and later resumed by any thread in the current thread team. An `if` clause may be used to enforce immediate execution of an explicit task.

```
int fib(int n) {
   int x, y;
   if (n < 2)
     return n;
   else {
      #pragma omp task shared(x)
         x = fib(n - 1);
      #pragma omp task shared(y)
         y = fib(n - 2);
      #pragma omp taskwait
         return x + y;
   }
}
```

Figure 1: C Source for Fibonacci kernel

A `task` construct may appear anywhere within a `parallel` region in a program, including another `task` construct. However, worksharing, `barrier`, and `master` constructs may not be closely nested within a `task` construct. The `task` construct accepts the `private`, `firstprivate`, `shared`, and `default` clauses to influence the data environment. To avoid complications of data going out of scope, variables are `firstprivate` by default, and their values are captured at task creation time. Synchronization is achieved using the `omp barrier` and `omp taskwait` constructs. The `taskwait` construct causes the encountering task region to suspend and wait for all of its child tasks to complete before resuming execution. When a `barrier` is encountered, all threads must wait until all other threads reach the barrier and all tasks created prior to the barrier are completed. Figure 1 shows the Fibonacci kernel with two `task` directives with `shared` clauses.

3 Related work

The Mercurium compiler, utilizing the Nanos runtime library (RTL), contains the first prototype implementation of OpenMP 3.0 tasks [2]. The Nanos RTL is an implementation of the nano-threads programming model [14]. Nano-threads are implemented using a modified version of the QuickThreads [10] library. The Nanos RTL, uses assembly language to implement user-level thread switching and supports the x86, IA-64, PowerPC, and Alpha architectures. The runtime library maintains a global queue, team queues, one for each team of threads, and local queues for each thread. At the start of execution a team of kernel-level threads is created. These threads execute an idle function looking for work. When a parallel region is encountered, a single nano-thread is created for each kernel-level thread and placed in the team queue where they are removed by the slave threads. The compiler extends the original Nanos runtime to support tasks. Tasks are represented as nano-threads. The first implementation used the same three-tiered queue organization as the original library. A task was placed on the team queue at creation. A thread searched for work in order of local queue, team queue, and finally global queue. However, they have recently implemented several different scheduling algorithms using local queues and work stealing [5]. When a task is suspended, it is placed back in a queue, and marked as suspended. When a thread accesses a queue for work it must iterate through the queue searching for a task that is ready to run. They also implement several cutoffs in order to limit the number of tasks created.

Cilk [6] is a language and runtime system developed at MIT to express task-level parallelism in a multithreaded environment. The runtime system uses a work stealing scheduling algorithm, where each thread maintains a ready queue of tasks. When a thread runs out of tasks, it attempts to steal a task from another thread in the system.

Work stealing algorithms resolve the tension between load balance and contention. A task is executed immediately upon creation while the current task is suspended. This achieves a "breadth-first theft, depth-first work" scheduling policy with minimal overhead and good data locality. Intel's Threading Building Blocks [17] uses a similar scheduling algorithm.

In Cilk and Intel's workqueueing model [18], a task's state is saved using a data structure to store local variables. When a task is suspended, all local variables are saved to the structure, and a *flag* is marked to indicate the position in the code where the task was suspended. Upon resumption, the flag is evaluated, and a *goto* statement is used to jump to the suspension point. This approach has difficulties if a task can be suspended inside a function call. This would require that a task's call stack be traversed upon its resumption. Cilk and Intel's workqueuing model get around this by not allowing tasks to be suspended within a function call. OpenMP 3.0 has no such restrictions, and therefore requires a more flexible method.

4 OpenMP 3.0 Implementation in OpenUH

In this section we will first describe the OpenUH compiler and its implementation of OpenMP 2.5. Then we will discuss the implementation of the OpenMP 3.0 tasking model; specifically the frontend, runtime library, and transaltion of the tasking constructs.

4.1 The OpenUH Compiler

The OpenUH [12] compiler is a branch of the open source Open64 compiler suite for C, C++, and Fortran 95 developed at the University of Houston. OpenUH contains a variety of state-of-the-art analyses and transformations, sometimes at multiple levels. Its interprocedural array region analysis uses the linear constraint-based technique proposed by Triolet [19] to create a DEF/USE region for each array access which are merged and summarized at the statement and basic block levels, and for an entire procedure. Dependence analysis uses array regions and procedure summarization to eliminate false or assumed dependencies in the loop nest dependence graph. Both the array region and dependence analyses use the symbolic analyzer, based on the Omega integer test [16]. We have enhanced the original interprocedural analysis module [20], designing and implementing a new call graph algorithm that provides exact call chain information. We have also created a graphical tool called Dragon [7] that provides the ability to request and view information on a submitted program and its data structures, typically in the form of graphs or text along with the corresponding source code.

OpenUH is compliant to OpenMP 2.5 for all three input languages in OpenUH, and we have designed and implemented novel extensions that provide greater flexibility and scalability when mapping work to many cores [4]. OpenMP is usually lowered relatively early in the translation process to enable optimization of the explicitly parallel code. The output makes calls to our PThreads-based runtime library. We are currently implementing the recent changes in OpenMP 3.0 and an explicit cost model for OpenMP in the compiler. OpenUH provides native code generation for IA-32, IA-64 and Opteron architectures. It has a source-to-source translator that translates OpenMP code into optimized portable C code with calls to the runtime library, which enables OpenMP programs to run on other platforms. Its portable, multithreading runtime library includes a built-in performance monitoring capability. Moreover, OpenUH has been coupled with external performance tools to support the analysis and visualization of a program's performance [8]. It is directly and freely available via the web (`http://www.cs.uh.edu/~openuh`).

Most OpenMP compilers translate OpenMP into multithreaded code with calls to a custom runtime library in a straightforward manner, via outlining [3], inlining [12], or an SPMD scheme [9] for clusters. Since many details of execution, such as the number of iterations in a loop nest to be distributed to the threads and the number of threads that will participate in the work of a parallel region, are often not known in advance, much of the actual work of assigning computations must be performed dynamically. Part of the implementation complexity lies in ensuring that the presence of OpenMP constructs does not unduly impede sequential optimization in the compiler. An efficient runtime library to manage program execution is essential.

4.2 Frontend

In a compiler, the frontend is responsible for parsing the source program and creating an intermediate representation (IR) for the compiler to manipulate. This phase generally consists of lexical analysis, syntactic analysis, and semantic analysis. Lexical analysis is responsible for dividing the source into *tokens* for later use by the frontend. During this phase, illegal character sequences can be detected. The syntactic analysis phase actually generates the IR and symbol table from the tokens. This level uses a

grammar to parse the token string and is able to detect syntactic errors, such as a `lastprivate` clause on an `omp task` construct. Semantic analysis is used to detect static errors in the source, such as an `omp for` construct closely nested within an `omp task` region.

The intermediate representation used is WHIRL, consisting of five levels, from Very High to Very Low, each pertaining to a different phase in the compiler. Frontends for C, C++, and Fortran generate Very High level WHIRL nodes, which are later lowered in subsequent phases of the compiler. In this work, we have fully extended the C frontend to support tasks and all related constructs and clauses. The Fortran frontend has been extended to accept the `omp task` and `omp taskwait` constructs, but does not yet accept any clauses.

In OpenUH, OpenMP constructs are represented as a `REGION` in WHIRL. `REGION` nodes consist of three kid blocks: `REGION_EXITS`, `REGION_PRAGMAS`, and `REGION_BODY`. The task pragma and any clauses associated with it are added to the `REGION_PRAGMAS` section and the body of the task resides in the `REGION_BODY`. To illustrate the WHIRL representation, Figure 2 shows a piece of an ASCII representation of the WHIRL after parsing the Fibonacci kernel (Figure 1). The example shows the first task in the Fibonacci code. For illustration purposes, the `task` construct contains an untied clause. The `REGION_PRAGMAS` section contains a block with `PRAGMA` nodes representing the `task` construct, `untied` clause, and `shared` clause. The `shared` clause contains an entry to 'x' into the symbol table.

```
REGION 1 (kind=4)
 REGION EXITS
  BLOCK
  END_BLOCK
 REGION PRAGMAS
  BLOCK
  PRAGMA 2 184 <null-st> 0 (0x0) # BEGIN_TASK
  PRAGMA 2 187 <null-st> 0 (0x0) # UNTIED
  PRAGMA 2 54 <2,2,x> 0 (0x0) # SHARED
  END_BLOCK
 REGION BODY
  BLOCK
 LOC 1 7
    BLOCK
       I4I4LDID 0 <2,1,n> T<4,.predf_I4,4>
       I4INTCONST -1 (0xffffffffffffffff)
      I4ADD
     I4PARM 2 T<4,.predef_I4,4> #  by_value
    I4CALL 126 <1,41,fib> # flags 0x7e
    END_BLOCK
    I4I4LDID -1 <1,40,.preg_return_val> T<4,.predef_I4,4>
   I4COMMA
  I4STID 0 <2,2,x> T<4,.predef_I4,4>
  END_BLOCK
 END_REGION 1
```

Figure 2: WHIRL representation of the first `task` construct in the Fibonacci kernel

4.3 Runtime

OpenMP runtime library in OpenUH not only provides the OpenMP library calls specified in the API but also a layer of abstraction between the compiler and the underlying POSIX Thread API. The RTL is responsible for managing threads with responsibilities such as thread creation, synchronization, and scheduling. With the introduction of tasks, the OpenMP RTL now plays an important role in task management and is now responsible for task creation, switching, and scheduling as well as enforcing task dependencies.

Task Scheduling Our implementation uses a distributed, work-stealing task scheduler in the OpenUH RTL. The distributed queue organization is in line with our goals of keeping synchronization to a minimum and promoting data locality. Work stealing is used to provide load balance to the system. The queues are doubly-ended queues, or *deques* (pronounced like "decks"), to allow a maximum amount of flexibility when developing a scheduling algorithm. We chose a doubly-linked list to implement our deques.

Ideally, we would like to allow concurrent accesses to the head and tail of the queue, but without synchronization, this can potentially result in a data race. Presently we use locks around the entire queue to guarantee mutual exclusion. This results in a slight increase in overheads, but since we expect few steals the contention should be low.

The scheduling strategy used in our runtime is inspired by the Cilk scheduler. As mentioned above Cilk uses a work stealing scheduler, where each thread maintains a single ready queue. When a task is created in Cilk, it is immediately executed and its parent is placed on the tail of the queue, resulting in a depth-first creation and execution of tasks. This can potentially benefit data locality and runtime overheads. The OpenMP 3.0 tasking model differs from Cilk; therefore deviations must be made from the Cilk scheduling algorithm in an OpenMP implementation. The OpenMP standard requires support for tied tasks. The use of a single queue per thread would require, during work stealing, that a thread iterate through the tasks in a victim's queue until a task that is eligible to be stolen was found. This greatly increases the theft time and, subsequently, the contention on a queue. Two queues, one public and one private, as shown in Figure 3, were chosen to separate tied versus untied tasks. The order of task creation and execution was also changed with respect to Cilk. The depth-first creation and execution are incompatible with tied OpenMP tasks. If this approach were used for tied tasks in OpenMP, all tasks would re-

main tied to a single thread.

In order to remedy this situation, we create tasks in a breadth-first manner, meaning that a task will execute, creating child tasks, until it reaches a synchronization construct. At that time it will be suspended and begin execution of its children. To maintain a depth-first execution, the tasks are placed on the queues in LIFO order. This process performs a breadth-first creation and a depth-first execution of tasks in the task graph, which allows child tasks to be stolen by other threads. Without this creation strategy, no parallelism would be achieved. With this approach, we can benefit from the data locality property of a depth-first traversal, but we require more memory than the Cilk scheduler, since we have more created tasks at any given time. This is not a problem unless large numbers of tasks are generated.

When looking for work, a thread first checks its private queue for work. If no work is found, it checks its public queue. If still no work is found, it attempts to steal work from another random queue. In the event the victim's queue is empty, the thread resumes execution of its implicit parallel task.

We first look at the private queue in order to finish execution of any tasks that have been started and allow more opportunities for other threads to steal tasks for better load balancing. Our experimental results show that attempting to steal only once per thread results in the best performance. We do not enqueue the implicit parallel task, but instead only switch back to it when no other work can be found. The reasoning behind this decision is that in the case of tasking, the parallel tasks will be at the root of the task graph and therefore should be executed last anyway. Furthermore, in the applications we tested, the parallel regions do not perform much work and are mainly responsible for generating more tasks.

In a tasking model, the system can quickly become overloaded with tasks. For instance, the Fibonacci kernel generates nearly three million tasks when computing the thirtieth term in the Fibonacci sequence. This puts considerable strain on the runtime library, especially regarding memory usage. To remedy this, the Nanos runtime library has used a cutoff [5] to serialize parts of the execution. Before a task is created, the runtime library is queried to determine whether the task should be created or executed immediately within the context of the current task.

We have adopted a similar technique in our implementation. In addition to the *num_tasks* and *depth* conditions introduced in the Nanos RTL, we have implemented two other conditions: *depthmod* and *queue*. The *depthmod* condition uses the depth of the current task modulo N to determine if the new task should be created. For an $N = 2$, every other depth will be skipped. The *queue* condition places an upper and lower limit on each queue and only creates a new task if the number of tasks in the queue are outside of the range. This effectively fills up the queue, depletes it, and repeats.

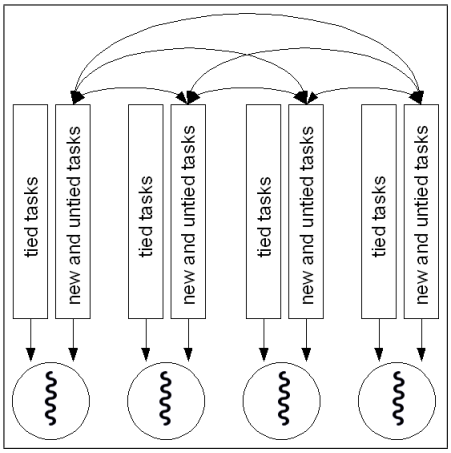


Figure 3: OpenUH queue organization

Task Synchronization In OpenMP, task synchronization is achieved through the use of the `omp taskwait` and `omp barrier` constructs. When a `taskwait` construct is encountered, a task must wait for all of its immediate children to complete before it can continue. This requires keeping track of the parent/child dependencies in the task graph. To accomplish this, each child maintains a pointer to its parent, with the root tasks' parent being `null`. The parent keeps track of the number of children it has created. A child decrements its parent's counter by one upon completion. Atomic operations are used to reduce contention. A task waiting on a `taskwait` construct is placed in a suspended state. It is the responsibility of the task's last completed child to place the task on queue. In the case of a *tied* task, it is placed on its starting thread's private queue. If the task is *untied*, we can place the task in any public queue. The two most intuitive choices are on the queue of the thread it was executing on, or the queue on which the child is executing. In our experiments we found that placing the task on the public queue of the child resulted in better performance. An overview of this algorithm is shown in Figure 4.

The `omp barrier` construct requires that all tasks created before the construct is reached be completed before any thread can continue. To accomplish this, each thread team maintains a counter denoting the number of incomplete tasks for that team. Upon entering a barrier, each thread decrements the counter by one and enters into

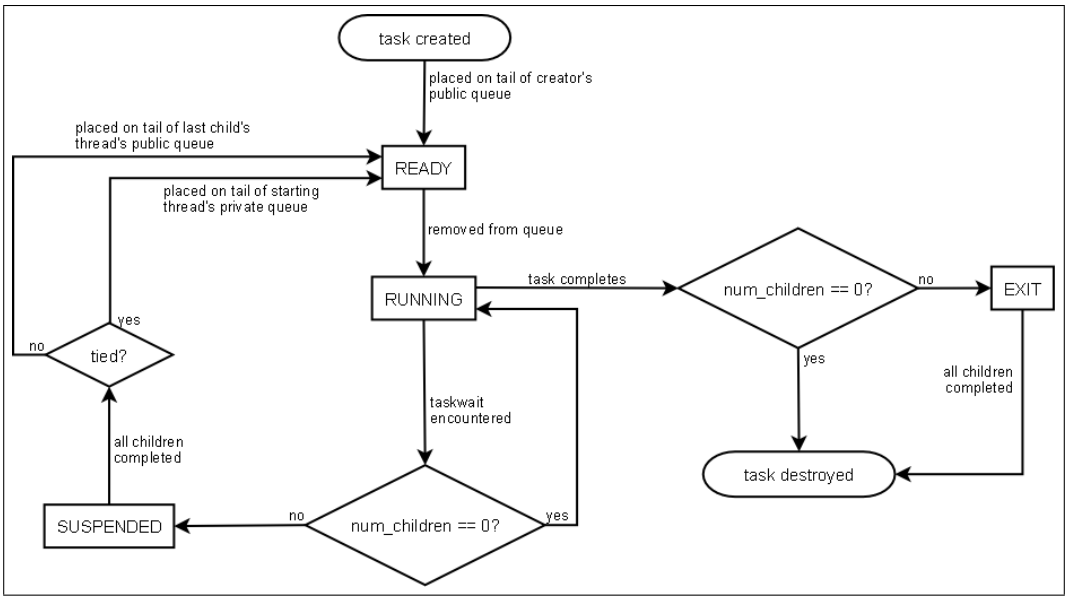


Figure 4: Flow of a task through the system

the scheduling algorithm discussed previously. When the counter reaches zero, each thread increments the counter and resumes execution.

Task Switching For simplicity and flexibility, we chose to implement task switching using a user-level thread library by building our implementation on top of the Portable Coroutines Library (PCL) [13]. The library uses either the `setjump`/`longjmp` or `ucontext` interfaces depending on the support of the system. We extended the library to make it thread safe as well as the basic data structure used to represent a coroutine to maintain the information needed for OpenMP tasks.

4.4 Translation

The translation of OpenMP constructs occurs in the LOWER_MP phase of compilation. This phase of the compiler is responsible for transforming code with OpenMP directives into multithreaded code that uses the runtime library. It takes a WHIRL tree as input with OpenMP constructs represented similarly as in Figure 2 and produces a WHIRL tree restructured to use the runtime library API.

The `omp taskwait` is directly translated into a function call, similar to the `omp barrier`. Other constructs, including `omp task` require much more complex transformations. The body of a task must be encapsulated in a procedure, its data environment must be setup, and the code to create the task must be inserted.

We are currently extending the OpenMP translation in OpenUH to support tasks. We have completed the translation of `omp taskwait` and are working on the translation of `omp task`. In the meantime, we are translating applications by hand to obtain performance results. Figure 5 shows how the OpenUH compiler will likely translate the Fibonacci kernel (Figure 1).

Any benefits of inlining tasks diminish as most variables will be `firstprivate` by default. The benefits of implicitly shared variables is lost and the data environment of a task must be captured at the time of a task's creation. Since the execution may be deferred, a copy of the variable at the time of task *creation*, not execution, is made. For simplicity, we have chosen to use the outlining approach with regard to tasks in our manual translation. As we implement the translation in OpenUH, we will investigate both inlining and outlining methods.

In the translation, the body of the task is copied into an independent function taking a `void` pointer as an argument. The data environment is created by defining a structure which contains all `firstprivate` and `shared` variables. Values of `firstprivate` variables are copied into the structure while the address of `shared` variables is copied and passed into the function. When the task is executed, all variables are copied from the structure into local variables. Originally we created two versions of every task function, one for serial execution and the other for parallel execution. This required a structure to be created regardless of whether `__ompc_task_create_cond()`

```
struct omp_task_1_args_ty{
   int n;
   int *x;
};

void __ompc_task_1(void *fp){
  struct omp_task_1_args_ty *args;
  int n, *x;

  args = (struct omp_task_1_args_ty *) fp;
  n = args->n;
  x = args->x;
  *x = fib( n - 1 );
  __ompc_task_exit();
}

int fib(int n){
  int x, y, sum;
  if (n < 2)
     return n;

  if (__ompc_task_create_cond() ){
     struct omp_task_1_args_ty *__omp_task_args_1;
     __omp_task_args_1 =
              malloc( sizeof(struct omp_task_1_args));
     __omp_task_args_1 -> n = n;
     __omp_task_args_1 -> x = &x;
     __ompc_task_create( __ompc_task_1,
                         (void*)omp_task_1_args, 1 );
  }else{
     int __omp_local_n;
     __omp_local_n = n;
     fib(__omp_local_n - 1);
  }
  .....
  __ompc_task_wait();
  return x + y;
}
```

Figure 5: Translation of the Fibonacci kernel

returned true or false. By only using the structure to pass arguments to the function in the case it returns true, we were able to reduce overheads. If false, we set up the data environment by creating temporary variables to store `firstprivate` variables and execute the code sequentially.

5 Evaluation

Testing of the implementation involved four applications originally written using either Cilk or Intel's Workqueueing model. The *NQueens* algorithm is a recursive depth-first backtracking algorithm to find all arrangements of N queens on an $N \times N$ chess board such that no queen can attack any other. The *Multisort* benchmark is a variation of a merge sort algorithm which recursively divides an array into four parts, sorting each one in parallel. The *SparseLU* benchmark performs an LU matrix factorization on a sparse matrix. The *Strassen* benchmark performs a matrix multiplication using recursion.

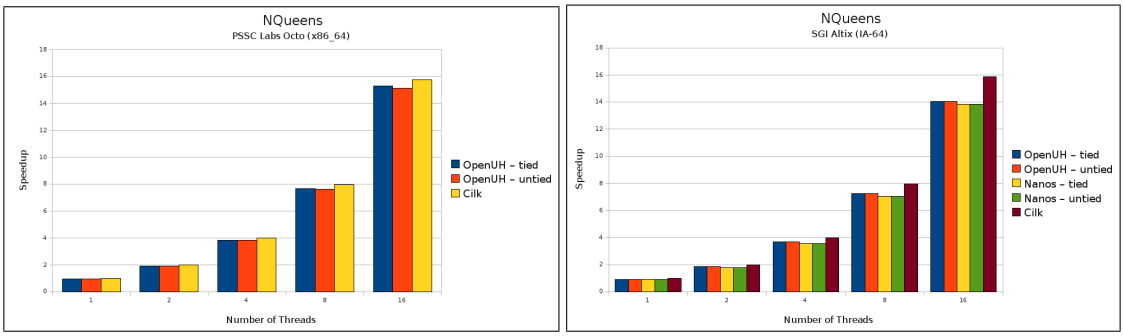
Our experiments consisted of comparisons to Cilk 5.4.6 and the Nanos 4.2 implementation of OpenMP 3.0. All results from the Nanos runtime used their default scheduling policy, the three tier queue organization discussed previously. For all benchmarks, except for NQueens, no task create condition was used. The *depth* condition was used for NQueens due to the large number of tasks created, which slowed down every implementation. All tests were performed on two machines. The first is a Professional Service Super Computers (PSSC) Labs Octo server with eight dual core Opteron processors (total of sixteen cores), each running at 2.6 GHz and 64GB of main memory. The second is an SGI Altix 350 consisting of eight nodes. Each node is an SMP with two Itanium2 processors running at 1.6 GHz with 16GB of main memory (128 GB total). All implementations were compiled with GCC 4.2.3 using O2 optimization levels.

We used the Nanos compiler for a source-to-source translation with OpenUH as the backend compiler on both machines. Cilk used GCC 4.2.3 as its native compiler on the Opteron server and GCC 4.1.2 on the Altix. All applications were compiled with the O3 optimization level. All measurements of speedup are with respect to the sequential versions without language extensions. In our NQueens comparison with Cilk, we inserted a depth cut-off into the Cilk source code in order to generate a similar number of tasks. The Nanos library was only evaluated on the Altix due to a lack of support for x86_64 architectures.

NQueens and SparseLU benchmarks serve as good indicators of the performance overheads caused by task management. In both applications there is little communication between tasks. Therefore, the less than linear speedup can be attributed to the tasking implementation. The Multisort (Figure 8), and Strassen (Figure 9) benchmarks are more data intensive than the previous two benchmarks leading to reduced performance. Significant performance degradation realized on the Opteron dual core system for all implementations when all cores were used, likely caused by increased cache pressure due to shared L2 caches. The Altix machine does not show this behavior, although we still see a less than linear speedup.

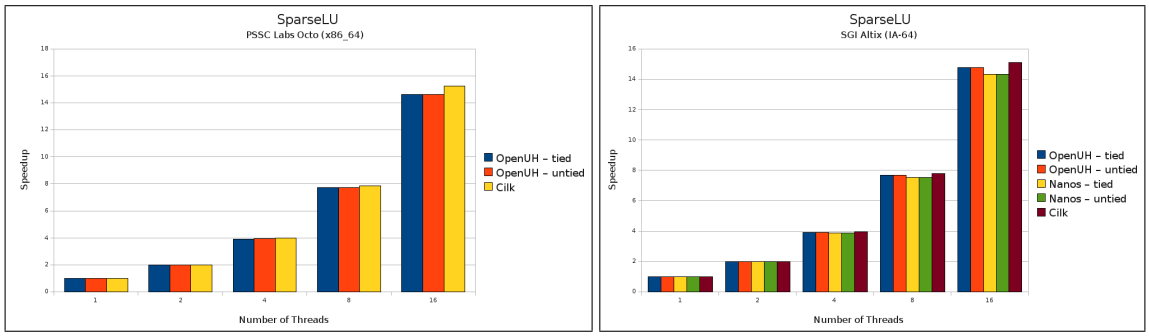
As can be seen in the graphs, the OpenUH implementation of tasks performs comparably with Cilk, and consistently outperforms the default Nanos implementation. The performance differences between Cilk and OpenUH are most likely due to less overhead in the Cilk implementation. This is mainly attributed to Cilk's more efficient memory allocation. The Nanos implementation evaluated uses a single queue implementation, which accounts for the performance gap.

An important observation is the differences between tied and untied tasks. In principal, *untied* tasks should provide better load balancing, which leads one to expect



(a) PSSC Labs Octo (b) SGI Altix

Figure 6: NQueens Speedup



(a) PSSC Labs Octo (b) SGI Altix

Figure 7: SparseLU Speedup

better performance. However, in the Multisort benchmark, *tied* tasks performed better. The reasons for this are not yet known, but most likely due to decreased contention on the task queues. When using tied tasks, the number of queues in the system is doubled, with work stealing only occurring among the public queues.

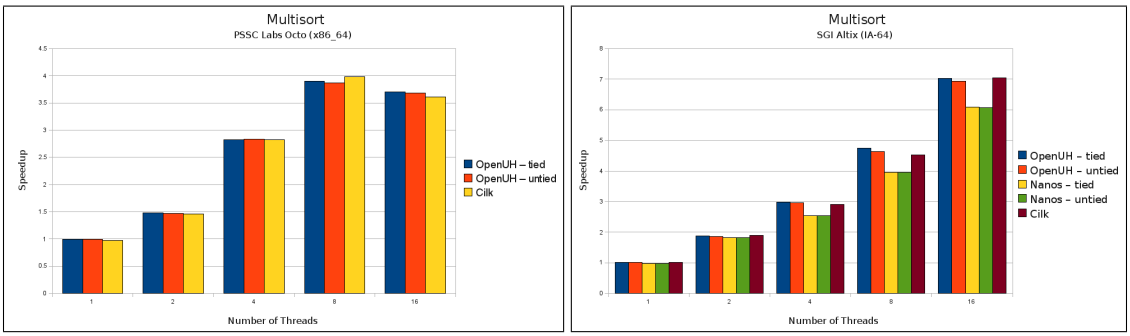
6 Conclusion and Future Work

The new OpenMP 3.0 standard provides the programmer with a new tasking interface. This new model allows for a more flexible environment in which the programmer can parallelize irregular algorithms. It can also significantly simplify the transition of algorithmic design to implementation due to a higher level of abstraction than threads provide. The new model, however, places more responsibility on the runtime library, like task management and scheduling.

The culmination of this work is a portable implementation of the OpenMP 3.0 tasking model. The frontends of the OpenUH compiler are being extended to accept the new constructs and generate IR that represents these constructs. The OpenUH runtime library was extended to facilitate the creation, scheduling, and synchronization of tasks. In particular, our scheduling algorithm utilizes a distributed queue organization with work stealing. The scheduler honors the tied restriction on tasks by using public and private queues. An outline for the translation of the OpenMP tasking construct has been presented to aid future compiler development in implementing the translation.

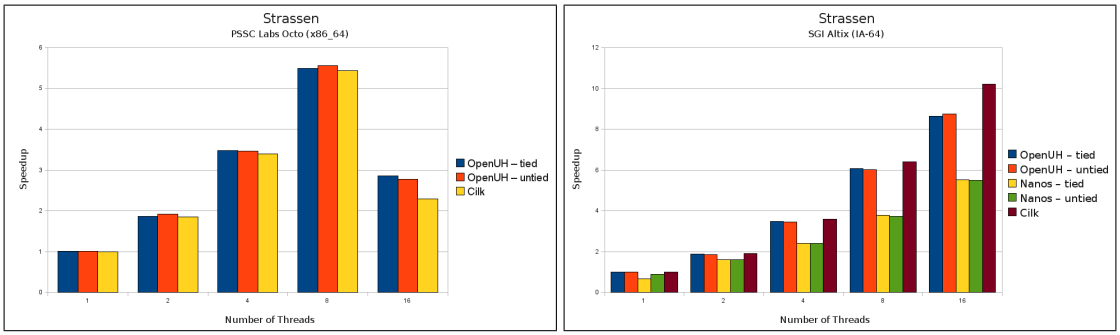
We have compared our implementation to the Nanos implementation of OpenMP 3.0 as well as Cilk. In this comparison we have shown good speedups as compared with the other implementations.

With the tasking model being new to OpenMP, much work needs to be done to investigate the implications for programmers, as well as exploring new techniques for implementation in order to achieve optimal performance. With regard to our implementation, the compiler translation must be completed. We are currently investigat-



(a) PSSC Labs Octo (b) SGI Altix

Figure 8: Multisort Speedup



(a) PSSC Labs Octo (b) SGI Altix

Figure 9: Strassen Speedup

ing better options in regard to inlining versus outlining and hope to complete the translation and the rest of the OpenMP 3.0 API in the coming months. The results presented in this paper show that new scheduling techniques must be developed for multicore architectures. Static and runtime analysis and performance data can be used to achieve better performance on these architectures. In particular, the shared resources of the cores, namely caches and memory buses, must be taken into account. To increase data locality on NUMA architectures, it might be possible to design a more intelligent work stealing policy in which threads are more likely to steal work from a thread which is "close" with respect to latency. Our implementation has focused on the scheduling of tied tasks. It treats tied and untied tasks the same at task creation and when scheduling. It could be advantageous to take Cilk's approach of executing a task immediately and placing its parent on the task pool when creating untied tasks.

A major benefit of OpenUH is it is an optimizing compiler. Its analysis capabilities could allow for compile time optimizations to be applied to tasking programs. The cost model [11] must be extended to account for multicore technologies as well as tasks. With this extension we can assign computational weights to tasks. This can lead to a partial static scheduling of tasks at compile time, as well as passing this information to the runtime library to allow it to make more intelligent scheduling decisions.

References

[1] C. Addison. Implementing the OpenMP 3.0 tasking model in the OpenUH compiler suite. Master's thesis, University of Houston, August 2008.

[2] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An Experimental Evaluation of the New OpenMP Tasking Model. In *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[3] C. Brunschen and M. Brorsson. OdinMP/CCp - a portable implementation of OpenMP for C. *Concur-*

rency - Practice and Experience, 12(12):1193–1203, 2000.

[4] B. M. Chapman, L. Huang, H. Jin, G. Jost, and B. R. de Supinski. Toward enhancing OpenMP's worksharing directives. In *Europar 2006*, pages 645–654, 2006.

[5] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *Proceedings of the 4th International Workshop on OpenMP (IWOMP '08)*, pages 101–110, May 2008.

[6] M. Frigo, C. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–233, 1998.

[7] O. Hernandez, C. Liao, and B. Chapman. Dragon: A static and dynamic tool for OpenMP. In *Workshop on OpenMP Applications and Tools (WOMPAT 2004)*, Houston, TX, 2004. University of Houston.

[8] O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore, and F. Wolf. Performance instrumentation and compiler optimizations for MPI/OpenMP applications. In *Second International Workshop on OpenMP*, 2006.

[9] L. Huang, B. Chapman, and Z. Liu. Towards a more efficient implementation of OpenMP for clusters via translation to Global Arrays. *Parallel Computing*, 31(10-12), 2005.

[10] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[11] C. Liao and B. Chapman. Invited paper: A compile-time cost model for OpenMP. In *Proceedings of the 12th International Workshop on High-Level Parallel Programming Models and Supportive Environment*, March 2007.

[12] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler. In *12th Workshop on Compilers for Parallel Computers*, 2006.

[13] D. Libenzi. Portable coroutine library. `http://www.xmailserver.org/libpcl.html`.

[14] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé. Nano-threads library design, implementation and evaluation. Technical report, DAC/UPC, September 1995.

[15] OpenMP ARB. *OpenMP Application Program Interface*, 3.0 edition, May 2008.

[16] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35:102–114, August 1992.

[17] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[18] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen. Compiler support of the workqueueing execution model for intel SMP architectures. In *European Workshop on OpenMP (EWOMP'02)*, 2002.

[19] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 176–185, New York, NY, USA, 1986. ACM Press.

[20] T. Weng. *Translation of OpenMP to Dataflow Execution Model for Data locality and Efficient Parallel Execution*. PhD thesis, Department of Computer Science, University of Houston, 2003.