TaskMan: Simple Task-Parallel Programming

Derek Hower
University of Wisconsin-Madison
Computer Sciences Dept.
1210 W. Dayton St. Madison, WI
drh5@cs.wisc.edu

Steve Jackson
University of Wisconsin-Madison
Computer Sciences Dept.
1210 W. Dayton St. Madison, WI
sjackso@cs.wisc.edu

Abstract

Programmers need better, more reliable techniques to build parallel programs. In the task-parallel programming paradigm, the programmer defines independent pieces of work, called tasks, that may run in parallel. The runtime library then takes responsibility for dynamically scheduling these tasks with whatever processor resources are available, keeping low-level synchronization and threading details out of view. For certain classes of programs, this paradigm can be extremely effective.

We present TaskMan, a C++ library for task-parallel programming designed with emphasis on programmability and simplicity. We discuss the TaskMan programming model and describe the Taskman runtime. This runtime is rather simple, but by exploiting fast communication between cores on a CMP, it can perform comparably to mature, heavily optimized task systems.

1 Introduction

With the advent of the multicore era, making parallel programming accessible to the masses has become a top priority for both industry and academia. One technique in the parallel programmer's toolbox is the task-parallel programming model. In this model, the programmer identifies independent units of work, called *tasks*, within her application. These tasks may be executed in parallel, but the burden of doing so is left to a runtime system that handles all task synchronization and scheduling. Recently, this model of task-parallel programming has gained significant momentum with the release of several industry supported tools. [3, 9]

Task-parallel programming brings two primary benefits to the parallel programmer. First, it abstracts away the details of synchronization, allowing the programmer to focus instead on the interesting portions of an algorithm. Programming with the more traditional explicit threads-and-locks model is difficult and error-prone, and can lead to unreadable code that is hard to maintain or port to new systems. In contrast, task parallel code can be kept relatively simple, and need not be tuned for a particular platform to get good performance.

Second, a task model gives the programmer greater power to express parallelism at relatively fine granularities. While it can be difficult to find enough parallel work to justify adding a new dedicated thread to a program's structure, it is often easy to find task-sized parallel work. Furthermore, if an efficient task system is available, it may become possible to parallelize library code without significant modification to programs that use the library.

Many existing task-parallel programming environments were designed with *performance* as their primary goal. Unfortunately, the performance gained has occasionally come at the programmer's expense. Task-parallel programming environments have either featured limited tools and unfamiliar languages, or they have introduced challenging syntax and usage patterns into existing languages. But as the architectural landscape changes, so too change the demands upon programmers. The question is no longer "How do we write the most efficient parallel code possible?" Instead, the question is "How do we write modestly efficient parallel code cheaply and quickly?"

In this paper we present a new task-parallel programming model and runtime designed with the goals of *simplicity* and *programmability*. The system, called TaskMan, exploits low latency interconnect in CMPs to provide an efficient yet intuitive model to the programmer. TaskMan is a pure C++ library that can be built and used on a variety of common platforms. The programming model makes use of futures [10] to present a call/return interface familiar to programers of imperative languages while still providing plenty of potential parallelism to the runtime environment.

We compare TaskMan to two existing, relatively mature task-based runtime systems: Cilk [6] and Intel's Thread Building Blocks (TBB) library[3]. We find that TaskMan, which has yet to be optimized, performs comparably to

these systems, while still delivering a lightweight and intuitive interface to the programmer.

The rest of the paper is organized as follows. In Section 2, we give an overview of task-based programming and explain many of the common characteristics of task-based runtimes. In Section 3 we discuss prior systems. Section 4 introduces the TaskMan programing model, and Section 5 gives details of the runtime implementation. We present our evaluation methodology in Section 6 and discuss results in Section 7. Section 8 concludes.

2 An Overview of Task-Based Programming

> *He could take up a task the moment one bell rang, and lay it aside promptly the moment the next one went, all tidy and ready to be continued at the right time.*
> – From "Leaf by Niggle," by J.R.R. Tolkien

Task-based programming is centered around the decomposition of code into independent tasks. In most existing systems, including our own, is the burden of the programmer to identify these independent regions and communicate them to a specialized runtime environment. The runtime, then, becomes responsible for scheduling and synchronizing the work. This runtime consists of a number of helper threads, which execute tasks in parallel as they become available.

Abstractly, the set of tasks that are ready to run form a set, from which an idle helper thread may take any element. However, to improve locality and minimize contention across distributed processing cores, this set is usually implemented as a distributed structure. Specifically, task-parallel runtimes employ *work-stealing task queues* [1, 2]. In the common case, a thread performs tasks available in its local queue. If it finds its queue empty, the thread steals work from the queue of another thread.

Within a queue, a tasks are represented as a tuple containing three types information: the task to be run (typically a function pointer), arguments needed by the task, and (optionally) extra data required by the runtime scheduler. When a task is spawned by a thread, a new task tuple is placed at the head of that thread's ready queue. Once placed in the queue, the task may be later executed by its parent thread, or stolen and executed by a different thread.

When a helper thread completes a task, it attempts to pop a task from the head of its work queue.[1] In contrast, stealing occurs from the tail of the queue. This helps to exploit locality that often arises due to a correlation between a parent task and its immediate children (which are the tasks closest to the head of the queue). Additionally, in recursive situations, the tasks closer to the queue's tail may take longer because they may spawn more subtasks. Stealing from the tail of a queue can thus enhance the likelihood of stealing a larger chunk of work, minimizing communication.

[1] "Queue" is something of a misnomer in this context, because the structure in question is really a double-ended queue, or deque. New tasks are always added to a queue's head, but they can be removed from either end.

```
cilk int fib(int n) {
    if (n<2) return n;
    else {
        int x, y;

        x = spawn fib(n-1);
        y = spawn fib(n-2);

        sync;

        return (x+y);
    }
}
```

Figure 1. An example of a Cilk-5 task.

The mechanism of work stealing helps to balance the load evenly among helper threads, while minimizing the overhead of the task runtime. Blumofe and Leiserson [2] have proven a number of theoretical bounds on this algorithm's use of time and memory space.

3 Prior Work

3.1 Cilk

Cilk [1, 6] is a research system designed to integrate task-based parallelism into a C-like programming language. It was originally developed for use in SMP systems; early versions strove to minimize thread communication due to its expense on these systems. The project has evolved through several generations, and is now in its fifth major release.

Cilk is both a compiler frontend and runtime library. The compiler frontend transforms code written for Cilk into GNU C, then passes the transformed code to the standard gcc compiler to build an executable. Though earlier versions of the system had complicated programming models, the syntax of Cilk 5 is designed to be programmer friendly.

Cilk tasks are written as C functions with a special `cilk` keyword decorating the function declaration. A task is invoked as a procedure call annotated with the `spawn` keyword. An example Cilk program is shown in Figure 1, which is a task-parallel implementation of a Fibanocci number calculation. When using Cilk, it is the responsibility of the programmer to call `sync` before using the return value of a spawned task. The `sync` keyword acts as a barrier which ensures that all child tasks visible in the current scope

have completed. The return value of any running task is undefined until the sync instruction is executed.

The Cilk programming model (at least in its current form) provides a simple interface to the programmer, one that integrates comfortably with sequential code written in C. As such, it is an elegant mechanism for expressing task parallelism. However, Cilk has some practical drawbacks. Subtle (and nondeterministic) bugs can occur if the programmer forgets to use the `sync` keyword before accesses to task return values. Because Cilk is based on the C language, it does not support C++ constructs. Also, the Cilk compiler's dependence on nested functions[5] means that Cilk can only be used in conjunction with compilers that support the GNU flavor of C. Finally, Cilk's custom compiler and runtime are both nontrivial applications, and may suffer portability and installation woes.

3.2 Threading Building Blocks (TBB)

In early 2007, Intel released the Threading Building Blocks (TBB) library an open source project. It is a C++ library that provides support for task-based programing and loop level parallelism, as well as a variety of support classes for multithreaded programming, such as thread-safe containers. The library is quite flexible, allowing for extensive performance tuning, and the associated runtime has been highly optimized to run well on Intel processors.

The TBB task model is object oriented; a task is represented as a class, and is invoked by passing an instance of the class to the runtime scheduler. Figure 2 shows how the previous Fibanocci example would be written as a TBB task. The task's code is placed inside of the `execute()` member function, which is a virtual method of the parent `task` class. Any local data needed by the task is declared as a class member variable, to be initialized through the task's constructor. Tasks in TBB must be allocated using placement-`new`, as shown in the example.

A function that creates tasks invokes them using one of the varieties of the `spawn()` function. The `spawn_and_wait_for_all()` function is analogous to a Cilk `spawn` followed immediately by a `sync`: it prevents the parent function from proceeding any further until all children have completed. TBB requires the programmer to explicitly state how many children a task will spawn by calling the `set_ref_count()` function. Depending on the variety of `spawn()` used, this reference may or may not include the parent. In Figure 2, the reference count is set to three: two for the children and one for the parent.

TBB also has support for more advanced task programming. Explicit continuation passing can be done in a TBB task by specifying a function to be run when the task completes. Additionally, TBB enables the programmer to bypass the scheduler by returning a new task from the

```
class FibTask : public task {
public:
  const int n;
  long* const sum;
  FibTask( long n_, long* sum_ )
    : n(n_), sum(sum_) {}

  task* execute() {
  if( n < 2 ) {
    *sum = n;
    return NULL;
  } else {
    int x,y;
    FibTask& a = *new( allocate_child() )
                  FibTask(n-1, &x);
    FibTask& a = *new( allocate_child() )
                  FibTask(n-2, &y);

    set_ref_count(3);

    spawn( b );
    spawn_and_wait_for_all( a );
    *sum = x+y;
    return NULL;
  }
}
```

Figure 2. An example of TBB task programming.

`execute()` function rather than `NULL`. The returned task will be executed immediately without consulting the scheduler. TBB also has wrapper classes to support conversion of parallel loops into tasks. Like the TBB `task` class, these wrappers are object-oriented.

TBB has the benefit of being a highly optimized platform for task parallelism. It exposes configuration hooks so that programs may be tuned for different architectures. Because of its industry support, it seems likely that TBB will continue to grow and improve as time goes on.

However, TBB does have drawbacks, mostly from an ease-of-programming standpoint. The class encapsulation model leads to code which is scattered and difficult to read; to move the functionality of a task into a class, the programmer must take the code out of its natural place in the flow of the program. The additional burdens of placement-`new` and manual reference counting multiply chances for programmers to make mistakes.

3.3 Task Parallel Library (TPL)

The Task Parallel Library (TPL)[9] is part of the Microsoft C# language. Syntactically, it is somewhat similar to TaskMan. TPL also supports loop-level parallelism in a manner similar to OpenMP[4]. Because TPL is a proprietary library and has only recently become available for public demonstration, we have not investigated it closely.

4 TaskMan: A Simple C++ Task Manager

> *Let me tell you how it will be*
> *There's one for you, nineteen for me*
> – from the Beatles song "Taxman"

We have designed and implemented TaskMan, an environment for task-based parallel programming. This system was designed with the following goals:

- The task system should be expressed as a C++ library, with no changes to the base language.
- The programmer's interface to the library should be as simple as possible.

An example of a TaskMan function (again, for computing Fibonacci numbers) appears in figure 3. The library's primary interface, made up primarily of the `task()` function and the `result` class, is summarized in figure 4.

The programmer defines a task by calling the `task()` function, which accepts any number of arguments[2]. The first argument must be a function pointer, and the remaining arguments are used as parameters to this function. The return value of `task(f,...)` is an object of type `result<T>`, where `T` is the return type of the function `f`.

The `result` object represents a *future*[10]. The value of the future may be accessed through the `result` class's `*` operator. This operator may be used multiple times; the task will execute only once.

The TaskMan library may run tasks in parallel, and evaluate them in any order. Consider the following simple example:

```
result<int> r = task( f, 1 );
...
cout << (*r);
```

The library guarantees that some thread will execute the call `f(1)`, at some time between the call to `task()` and the return from `(*r)`. Note that no programmer-visible variables ever have indeterminate state in this example (unlike in Cilk, where task return values are undefined before a `sync` keyword).

Class member functions may serve as the first argument of a call to `task()`. In this case, a pointer to an object of the relevant class must occupy the second argument. For example, given a class `C` with member function `func`, the following is a valid task definition.

```
C obj;
result<...> res;
res = task( &C::func, &obj, ... );
```

Because tasks may run in parallel and in any order, there is ample opportunity for race conditions, especially on global data. It is the programmer's responsibility to ensure that tasks are independent and thread-safe. If necessary, the programmer may use locks or other synchronization mechanisms to guard access to global structures within tasks. (However, accessing globals during tasks may lead to serialization and poor performance. Tasks are best suited to truly independent work.)

5 Implementation of TaskMan

5.1 `task()` and `result`

A call to the `task()` function creates a tuple representing the task definition and pushes that tuple to the head of the thread's task queue. The calling function then continues. This behavior differs from that of the abstract task-stealing system described by Blumofe and Leiserson. Their theoretical system begins executing a freshly defined task immediately, pushing the remainder of the calling function onto the queue as a stealable task. However, this approach requires

[2] In reality, there is a static limit to the number of arguments accepted by `task()`. However, this limit is an implementation detail and can be increased at need.

```
int fib( int n ) {
  if( n < 2 )
    return n;
  else{
    result<int> x, y;
    x = task( fib, n - 1 );
    y = task( fib, n - 2 );
    return ( *x + *y );
  }
}
```

Figure 3. An example of TaskMan programming.

```
void taskman_init( int num_threads );
void taskman_shutdown();

template <typename T>
class result{
  T operator*();
};

// for any function with type
// signature T f(A1, ..., An)
result< T >
task( T (*func_p)(A1, ..., An),
      A1 arg1, ..., An argn );
```

Figure 4. TaskMan API

a level of support for continuations and closures [1] that is not available in standard C++,. Thus, we have chosen to take the simpler approach.[3]

The `*` operator in the `result` class does not immediately launch the associated task. Instead, this operator extracts the future's values using the following algorithm:

```
1  get_result_value( result<...> r )
2    let t = the task associated with r
3    if( t has finished executing )
4      return t's return value
5    else
6      do
7        get a task and run it
8      until (t has finished executing)
9      return t's return value
```

Line 7 of this algorithm first check the thread's local task queue for available work. If it finds the local queue empty, it attempts to steal from other threads. If no tasks are available anywhere, line 7 takes no action, and the loop iterates.

This algorithm for forcing a future has several implications:

- The `*` operator may return very quickly (if the associated task has already run), or may run for some time. In particularly, it may recursively execute any number of tasks.
- The `*` operator does not necessarily run the task associated with the `result` it is called on. (That task may have been stolen by another thread.)
- Any task, once launched, remains at a fixed location in the runtime stack until it completes. Partially completed tasks are never put into a work queue.
- If another thread is executing the task associated with a `result`, and there is no other work available, the thread accessing the `result` may spin on the task's "finished" flag. This is a potential source of overhead, although this situation should be rare in practice.

It should be noted that the approach we have taken can lead to very deep stacks, particularly when tasks are spawned recursively. The stack depth itself is a potential source of overhead.

5.2 Work stealing

Like Cilk, TaskMan assigns a work queue to each thread. The queue interface is designed to work polymorphically, so that the work queue implementation can be interchanged.

[3] We do not know what effect, if any, this modified approach has on the performance bounds proven by Blumofe and Leiserson, though in practice the impact appears to be minimal.

Thus TaskMan could be used as a test platform to compare implementations of the work-stealing algorithm.

To date, we have only had time to bring up one implementation. In this implementation, each queue is a wrapper around a `std::deque` object, with access to this object synchronized with a per-queue mutex lock. A thread uses blocking lock operations to add or remove a task from its own queue, and uses a non-blocking `trylock()` operation when stealing from another queue. If a lock is not immediately available upon a steal operation, the steal is considered a failure. Since no thread ever blocks on a lock other than its own, deadlock is avoided.

We have two partially complete task queue implementations, which could be analyzed in future work. One is a transactional memory implementation, in which access to the work queues is synchronized with atomic blocks. The second is an implementation that relies on hardware task queue support as described by Kumar et. al. [8]

5.3 Tasks in C++

Futures can be thought of as a form of lazy evaluation [10]. Lazy evaluation is not directly supported in C++, and the syntactic machinery that makes it possible is fairly complicated. Though we have sought to hide this complication within the library, away from the programmer, it leads to additional function calls that add to already considerable stack pressure.

The static type discipline of C++ is also a limitation. In order to make the `task()` function both flexible and type-safe, extensive use of templates is required. There are two primary disadvantages of this. First, templated functions and classes are compiled to separate object code for each template signature. Thus, in a program that uses many different functions as tasks, many variants of `task` will be compiled, increasing the program's size and contributing to bloat. Second, in the neighborhood of templated code, syntax and type errors are notoriously verbose and confusing. The `task()` function is type safe, but mistakes in using it may produce unreadable error messages.

TaskMan's `result` class relies on the C++ feature of operator overloading. We chose to use the $\star$ operator as the means of forcing a future because of its syntactic similarity to pointer dereference.

6 Methodology

We evaluate the performance of TaskMan by comparing it against two systems that represent the current state of the art in task-based programming, namely TBB and Cilk. Our evaluations are run on two different CMP systems with considerably different architectures. The first is an 8 core machine consisting of two 2.33GHz Intel Core2 Quad "Clovertown" chips with 4GB of memory. Processors in this system are dynamically scheduled superscalar cores, with each core containing a dedicated floating point unit. The second system contains a single Sun Ultrasparc T1 "Niagara" with 4GB of main memory. The T1 consists of 8 cores, each of which is 4-way multithreaded, for a total of 32 in-order hardware contexts. All contexts share a single floating point unit that has an access latency similar to an L2 cache bank [7].

6.1 Benchmarks

6.1.1 Statistical Evaluator

To gain insight into the the overheads of TaskMan, we created a microbenchmark that executes null tasks, the durations of which are determined by a random sampling from a chosen probability distribution. We use this benchmark to determine two things: 1) how TaskMan performs when executing tasks of different average duration and 2) how TaskMan is affected by burstiness in task arrivals (i.e. enqueues). We ran tests on samples from the Exponential distribution to determine the former and the Pareto distribution for the latter.

The exponential distribution is characterized by by a coefficient of variation equal to one, such that the mean and standard deviation are equivalent. This leads to a meaningful mean value and limited burstiness in task arrivals. For the Pareto distribution, we chose the distribution parameters so that samples had more variation than those from the exponential distribution while still maintaining the same mean value. This increased variability leads to more burstiness, which can be used to determine how TaskMan performs when activity is unbalanced.

We ran two different versions of the statistical benchmark to simulate loop-level parallelism and task-level parallelism. In the loop-level version, tasks are spawned in bulk at the beginning of the run, at which point the main thread waits until all tasks have completed. For the task-level version, tasks are spawned recursively in such a manner that the task dependencies form a 4-ary tree of variable depth. We approximated both head and tail recursion by spawning children both before and after a task's alloted working time has elapsed. Each task waits on its children to complete before returning.

The tasks in this microbenchmark simulate working time by spinning on hardware tick counters until the duration specified by a random sampling has elapsed. Because the hardware counters on the Sun T1 serialize across the system, and because the counters on the Clovertowns do not, we only ran this microbenchmark on the Intel system. Also, due to time constraints, we only compare TaskMan to TBB for this particular workload.

6.1.2 Cilk Suite

We converted three benchmarks that ship with the Cilk distribution into both the TaskMan and TBB models. The first benchmark is a dense matrix multiply, the second a heat diffusion computation, and the third a parallel LU decomposition. Because TBB cannot run on an Ultrasparc architecture, and because all three are floating point intensive programs, we only evaluate these workloads on the Clovertown system.

6.1.3 Othello AI

Finally, we created an AI program for the board game Othello (also known as Reversi).[4] The AI performs recursive minimax search, up to a specified depth, then uses heuristics to evaluate the quality of board positions. We implemented two heuristics:

- A *simple* heuristic that evaluates a board based on the number of pieces each player possesses.
- A *strategic* heuristic that additionally considers the strategic value of each square on the board, preferring to capture edges and corners.

We created two parallel versions of this AI, one using TaskMan, and another using TBB's task interface. In both cases, the AI's recursive calls are redefined as task spawns. For a lookahead depth of n, each task-based AI spawns tasks for depths up to $n - 2$, and uses serial recursion to evaluate the last 2 levels of the minimax tree. Doing so reduces overheads by decreasing the number of task spawns and increasing the average size of a single task.

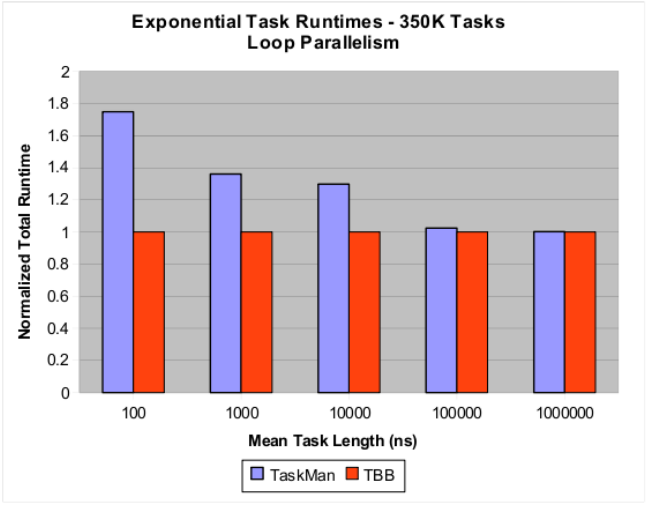
To avoid needless serialization on the Niagara architecture, the Othello benchmark explicitly avoids floating point computation within tasks.

7 Results

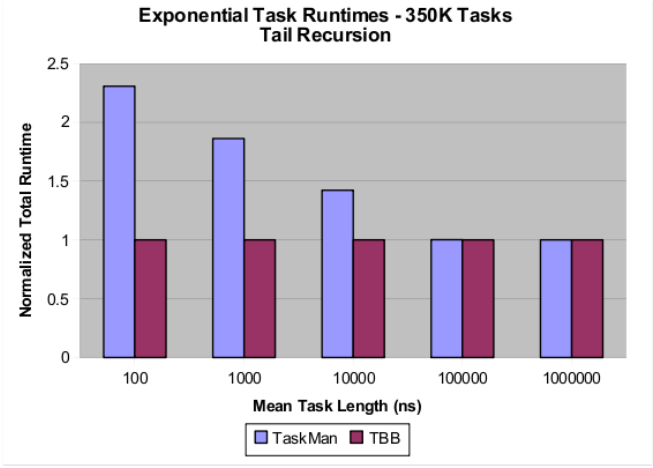
7.1 Statistical Evaluation

Figure 5 shows the results of our statistical evaluation with samples from the exponential distribution. On smaller tasks, TaskMan has a runtime overhead approximately twice that of TBB. However, when tasks are large (about .1ms), the overheads become negligible. This is because contention on the task queues is nearly eliminated when tasks run much longer than the time it takes to schedule a new task in the runtime. As can be seen from the contrast of Figures 5a and 5b, the task parallel version performed slightly worse than the loop parallel version. This is

[4]Though we wrote this benchmark from scratch, its design is based on a prototype written by Dan Gibson.



(a)



(b)

Figure 5. Results of the statistical evaluation.

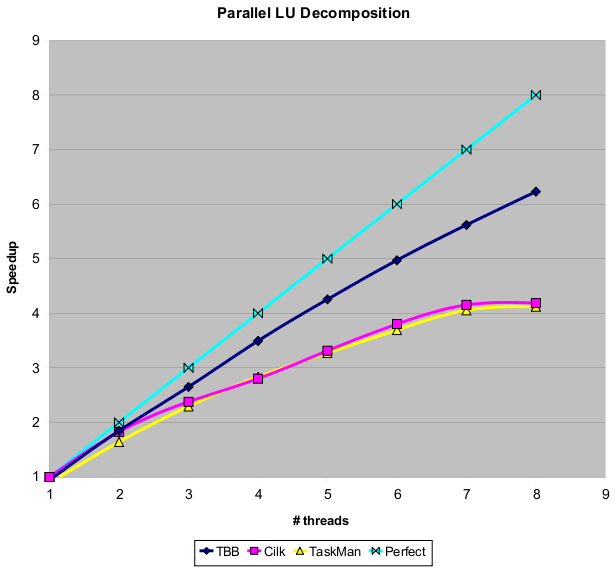


Figure 6.

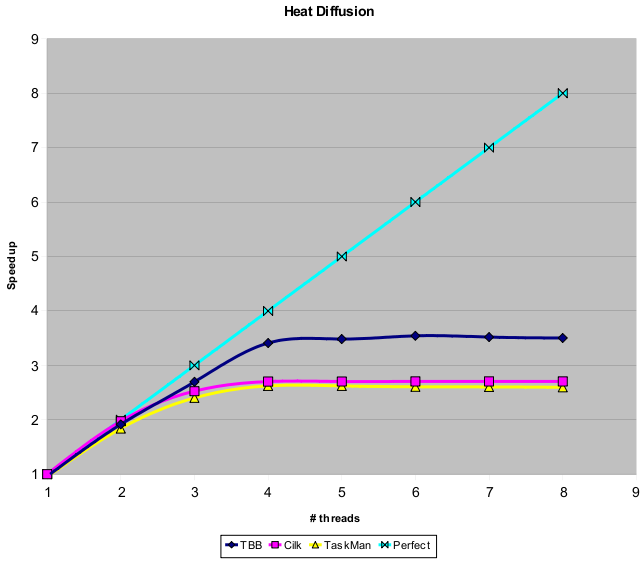


Figure 7.

likely because the current queue implementation has a single lock per thread queue that protects enqueue, dequeue, and stealing operations. Thus, the task parallel version sees more queue contention due to simultaneous enqueue and dequeue/stealing operations. A better queue implementation, such as a lock-free structure or one synchronized with memory transactions, could alleviate much of that pressure.

We found no noticeable difference between the head and tail recursion versions of the task-parallel statistical evaluation, and thus only show results from the tail recursion version. Likewise, we do not show the results of the statistical evaluation using a Pareto distribution because results were similar to those shown in Figure 5. This indicates that TaskMan is relatively insensitive to burstiness in task arrival times. Consequently, the mean of task durations is the main determining factor in the performance of the TaskMan runtime.

7.2 Cilk Suite

In Figures 6-8 we give the results of our converted Cilk benchmarks. In both `plu` and `heat` we perform on par with Cilk and Slightly worse than TBB. We presume this is because TBB has been highly optimized to run on Intel processors, while TaskMan has yet to be optimized for any architecture. Interestingly, we show, at least for these benchmarks, that TaskMan is able to utilize the benefits of a fast CMP interconnect to perform as well as Cilk, which has been in development for over a decade, without introducing significant complexity in either the runtime implementation or programming model. The `matmul` benchmark

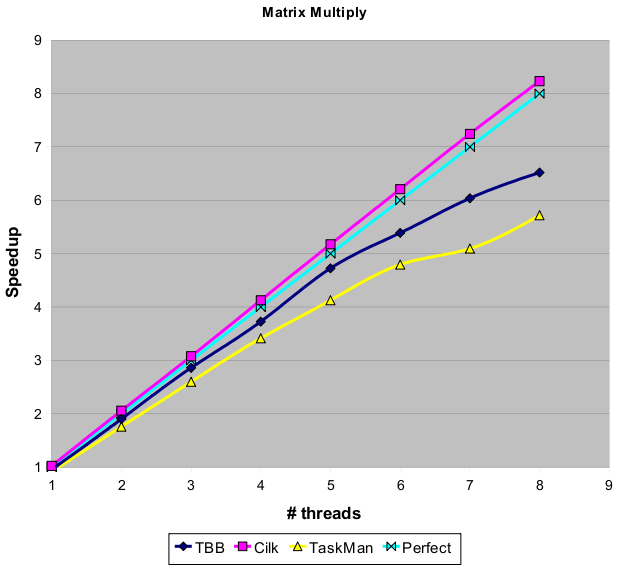


Figure 8.

again shows TaskMan performing slightly worse than TBB, though in this case Cilk outperforms both substantially. We speculate that this benchmark gains significant performance benefits from the new-task-first scheduling policy used in Cilk.

7.3 Othello

Figure 9 shows the performance of TaskMan Othello vs. TBB Othello on the 8-core Intel processor, for increasing lookahead depths. Speedups are given relative to the original, serial AI. Note that both systems have performance ceilings when the number of available tasks is relatively small (as when depth = 4). As more tasks become available, each system enjoys improved scalability.

We have noticed that, in figure 9, the speedup curves for the TaskMan AI generally see a reduction in slope when the processor count exceeds 4. We speculate that architectural effects may be at work: the Clovertown architecture has a slower interconnect between the two sets of four cores. However, this effect deserves further investigation before we make any certain claims.

In figure 10, we show the performance of TaskMan on the Sun Niagara processor. Here we can see the difference of task size on performance: the strategic board evaluator, which does more work per board and leads to longer tasks, sees a greater speedup.

Experiments on the Niagara use processor binding to distribute worker threads evenly among the Niagara's eight cores. Note the changes in speedup slope at approximately 8, 16, and 24 threads. In our experiments, maximum performance is generally attained with 24 threads, corresponding to the use of 3 out of 4 SMT threads per core.

Because of TaskMan's deep stack depths, the Niagara suffers a large number of traps to handle the spilling and filling of register windows. We speculate that this may be a source of significant overhead on this architecture.

8 Conclusions and Future Work

We have presented TaskMan, a simple C++ library for task-based parallel programming. We have found this library to be a useful tool for expressing task-parallel computation, and its similarity to Cilk has made converting existing Cilk programs very easy. The performance of TaskMan on modern CMPs is comparable to the performance of mature, highly optimized task systems, though the mature systems do perform better.

In the future, TaskMan could be extended in several ways. It could be used as a platform to compare different implementations of the work-stealing algorithm, without requiring the rewriting of benchmarks. The programming model of TaskMan could be extended to more fully match the feature sets provided by mature systems. And, of course, TaskMan can still be optimized for speed: we have yet to reach the end of the long, dark path of performance tuning.

Acknowledgments

The authors would like to thank Evan Driscoll and Ben Liblit for helpful discussions. Dan Gibson's technical assistance was invaluable.

References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[3] Intel Corporation. Intel threading building blocks 2.0 for open source. `http://threadingbuildingblocks.org/`, 2007.

[4] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[5] The Free Software Foundation. Using the gnu compiler collection – nested functions. `http://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html`, 2007.

[6] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[7] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[8] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, New York, NY, USA, 2007. ACM.

[9] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. `http://msdn.microsoft.com/msdnmag/issues/07/10/futures/default.aspx`, 2007.

[10] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

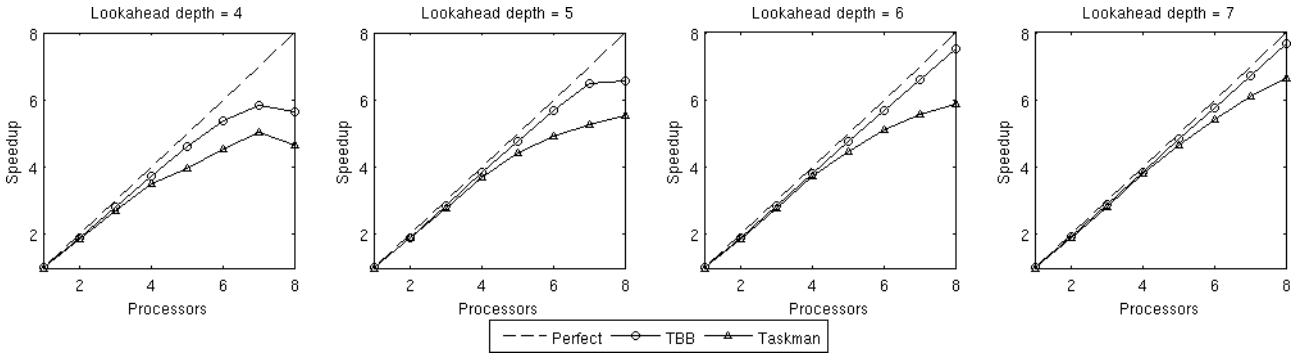


Figure 9. Othello benchmark results: TBB vs. Taskman on 8-core Intel processor

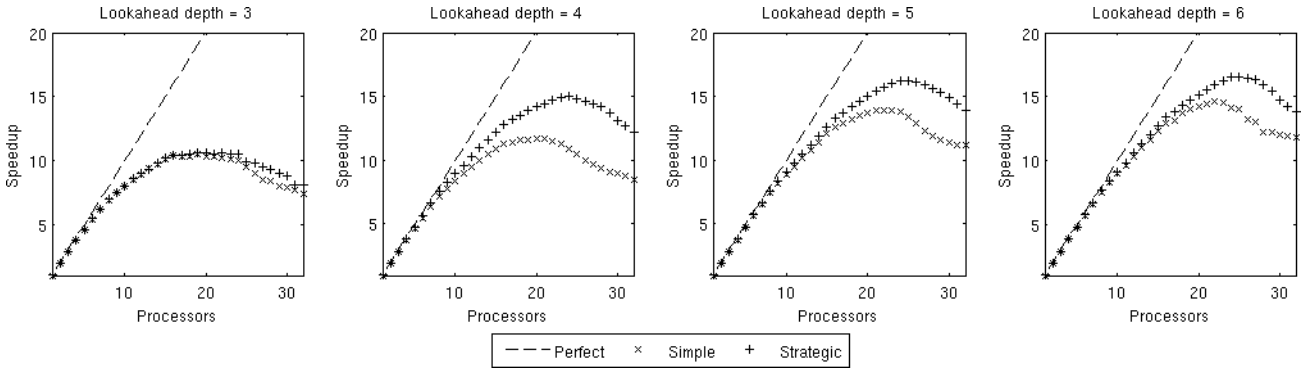


Figure 10. Othello benchmark results: Speedups on 32-thread Niagara processor, for simple board evaluator (smaller tasks) and strategic evaluator (bigger tasks)