A Comparison of some recent Task-based Parallel Programming Models

Artur Podobas[1], Mats Brorsson[1,2], and Karl-Filip Faxén[2]

[1] KTH Royal Institute of Technology
[2] Swedish Insitute of Computer Science (SICS)

Abstract. The need for parallel programming models that are simple to use and at the same time efficient for current ant future parallel platforms has led to recent attention to task-based models such as Cilk++, Intel TBB and the task concept in OpenMP version 3.0. The choice of model and implementation can have a major impact on the final performance and in order to understand some of the trade-offs we have made a quantitative study comparing four implementations of OpenMP (gcc, Intel icc, Sun studio and the research compiler Mercurium/nanos mcc), Cilk++ and Wool, a high-performance task-based library developed at SICS.

We use microbenchmarks to characterize costs for task-creation and stealing and the Barcelona OpenMP Tasks Suite for characterizing application performance. By far Wool and Cilk++ have the lowest overhead in both spawning and stealing tasks. This is reflected in application performance when many tasks with small granularity are spawned where Cilk++ and, in particular, has the highest performance. For coarse granularity applications, the OpenMP implementations have quite similar performance as the more light-weight Cilk++ and Wool except for one application where mcc is superior thanks to a superior task scheduler.

The OpenMP implemenations are generally not yet ready for use when the task granularity becomes very small. There is no inherent reason for this, so we expect future implementations of OpenMP to focus on this issue.

1 Introduction

Now that parallelism is the only way forward to be able translate Moore's law into performance, it has become all the more important to find parallel programming models that are suitable for future manycore architectures from today's 4-64 cores to 100s in five years and 1000s in ten years and beyond [11,6].

We will soon have access to more cores than we will expect to utilize effectively at any given time. We may not even be able to run all at full speed for power reasons. Even so, scalability of application performance is of utmost importance to be able to deliver continued total system performance improvements. Some current approaches to parallel software development require the programmer to handle the complexity of performance scalability in addition to

exposing the parallelism in the underlying algorithms. We believe that this is a dead end to leverage the manycore technology at a wide scale. The vast majority of programmers must be able to focus on exposing potential parallelism with the aim for high-quality and high productivity. For portability and efficiency reasons, it should be left to the system software layers to deal with assigning work to available resources dynamically in run-time, although there is a need to expose it to programmers in special cases.

All of these considerations favour programming in high level programming models which provide *abundant fine-grained parallelism*, obviating the need for developers or compilers to explicitly map computations to the underlying hardware, which in any case is a moot approach in the face of dynamic workloads and heterogeneous hardware. This mapping is instead done by a run-time system that takes care of scheduling and resource management of the parallel activities. Such programming models are characterized by large numbers of dynamically created concurrent computations (tasks).

An important development are the task-parallel programming models such as exemplified by OpenMP [3,15], Cilk++ [5,13], and the Intel TBB framework [16]. This style is characterized by fine-grained parallelism that follows closely the structure of the application. For instance, a parallel loop can be implemented as a set of tasks corresponding to one or more loop iterations. The main form of synchronization is waiting for the completion of child tasks. Data parallelism can be realized as a higher level abstraction on top of task parallelism. Thus, efficient implementation of task parallelism also aids this model. A vital property of task-level parallelism is its ability to cope with heterogeneous and dynamically varying numbers of processing cores which is an inevitable result of future manycore development as we approach physical limits. While most implementations of this parallel programming model is done entirely in user-level libraries, there is at least one implementation where the model is integrated in the operating system [1].

This study is a performance comparison between six different implementations of task-parallel programming models. The models looked at are four implementations of OpenMP, Cilk++ and *Wool*, a new high-performance task library [9]. The four OpenMP implementations are: Gcc (v 4.4) [14], Intel Icc (v 11.0), Sun Studio 12 (update 1) and Mcc (Mercurium version 1.3.1 with Nanos run-time system version 4.1.3), a research compilation framework and run-time system from Barcelona Supercomputer Center [2,4]. For the study we have used a set of microbenchmarks developed by ourselves and applications from the Barcelona OpenMP Tasks Suite [8]. Although different schedulers and implementations have been compared before, this is, to the best of our knowledge the first to include Wool and to explicitly investigate the effects of fine-grained task-based parallelism [2,12].

We have found that the studied OpenMP implementations are not yet ready for fine-grained task parallelism. The associated overheads are by far to high. Cilk++ and Wool, on the other hand perform comparably well with a sligth advantage for Wool.

2 Task-based parallel programming models

The task-parallel programming models represented by the implementations studied here were pioneered in the mid-90s by Blumofe at al. at MIT [5]. It was early recognized that the work-sharing constructs of OpenMP are not sufficient to express the potential parallelism in programs dominated by pointer-based data structures [17], but it took almost ten years to enter the OpenMP specification [3]. Wool was developed in order to further investigate the overheads associated with the task-based model and we have found that it indeed is possible to further push the limits.

Below is a short introduction to how each of the three models: OpenMP, Cilk++ and Wool implement task-based parallelism exemplified on a recursive calculation of the Fibonacci sequence.

2.1 OpenMP

OpenMP is a programming model that was created by a group that was representing several major vendors of high-performance computing [7]. It uses compiler directives and library routines to express and control parallelism. By adding these compiler directives to a sequential program, the users specify what parts are to be executed in parallel and how. As of version 3.0, OpenMP supports constructs for task-based parallelism while previous versions focused on loop based parallelism [15]. Figure 1 shows a code snippet of the finbonacci calculation. Parallelism is created by the "`#pragma omp parallel`" construct which creates a "team of threads". The statement following the `single` directive is executed by the first encountering thread and kicks-off the recursive computation.

The compund statement following a "`#pragma omp task`" construct constitutes a computation which can be scheduled to be executed by any of the participating threads in a team of threads. A task may be executed immediately at the task creation or deferred to later execution by some other thread. By default, tasks are tied to the thread that starts executing it, so once a task has begun execution, it is then always executed by the same thread. The algorithm by which tasks are scheduled to available threads is not specified by the OpenMP specification but rather left to the implementation.

The `taskwait` construct suspends execution of the current task until all tasks created within this task has finished. In the example of Figure 1 this means that we are guaranteed to have valid values of variables `x` and `y`.

2.2 Cilk++

Cilk++ is model created and maintained by Cilk Arts, based on the original cilk model developed at MIT [13,5]. Through a small number of keywords, which are used to define possible parallel areas of a serial code, efficient parallel execution is realized. Removing these keywords from a cilk program creates a so-called "serial elision" of the program, which basically is a serial version of the programmed that can be used for debugging purposes. The Cilk++ scheduler is a work-first (also

```
    ...
#pragma omp parallel   /* Parallel region, a team of threads is created */
#pragma omp single
       {
           /* Executed by the first thread */
           fib_result= fib(n);
       }
    } /* End of parallel region */
    ...

int fib(int n) {
    int x, y;
    if (n < 2)
        return n;
    else {
        #pragma omp task shared(x)
           x = fib(n-1);            /* A new task */
        #pragma omp task shared(y)
           y = fib(n-2);            /* A new task */
        #pragma omp taskwait        /* Wait for the two tasks above to complete */
           return x + y;
    }
}
```

Fig. 1. OpenMP Fibonacci

called depth-first) scheduler with a work-stealing mechanism where different idle workers can steal from other workers task pools. Figure 2 shows the example function written using Cilk++. `cilk_spawn` is the keyword for spawning a task, and `cilk_sync` will synchronize all spawned tasks with their parent.

The `cilk_spawn` and `cilk_sync` constructs are direct counterparts to the `task` and `taskwait` constructs of OpenMP. In contrast to OpenMP, the worker threads are completely implicit in Cilk++ and only the tasks are explicit. Also, the scheduling of tasks is predefined and not open for different implementations.

2.3 Wool

Wool is a library supporting the nested independent task parallel programming model [10]. It provides constructs for defining, spawning, and joining with tasks as well as for defining and invoking parallel `for` loops. Joining is accomplished using the `SYNC` operation which blocks until evaluation of the corresponding task is completed, providing for a direct, as opposed to continuation passing, program structure. Wool is designed to test the limits of low overhead task management. Figure 3 shows the example function written using Wool. `SPAWN` creates a task and `SYNC` synchronizes with the task, and fetches the return value off it. `CALL` is basically a faster version of a merged `SPAWN` and `SYNC`.

```
int fib(int n) {
  int x, y;
  if (n < 2)
    return n;
  else {
    x = cilk_spawn fib(n-1);
    y = cilk_spawn fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Fig. 2. Cilk++ Fibonacci

```
TASK_1 (int, fib, int, n) {
        if (n < 2)
                return n;
        else {
                int x, y;
                SPAWN( fib, n-1 );
                y = CALL( fib, n-2 );
                x = SYNC( fib );
                return x + y;
        }
}
```

Fig. 3. Wool Fibonacci

Wool is implemented using work stealing, that is, each processor (core) has a private task dequeue; `SPAWN` pushes a task on to the *owner* end of the dequeue while a `SYNC` pops a task from the same end. When a processor is out of work it steals a task from the *thief* end of the task dequeue of a randomly selected *victim*.

The stacklike behaviour of `SPAWN` and `SYNC` is visible in the API since a `SYNC` always joins with the most recently spawned, unjoined task. `SPAWN` operations do not return a result; if the task is stolen, the result is stored in the task queue when ready and extracted by the corresponding `SYNC`. Tasks that are not stolen are invoked by the `SYNC` using the arguments stored in the dequeue, this is known as *inlining* the task. A `SYNC` operation takes as argument the name of the task definition of the task it expects to join with, so that the code to invoke when inling the task is known and can be called directly, exposing it to compiler optimization.

Synchronization between thief and victim is based on individual task descriptors in the task dequeue rather than on the pointers into the queue, so that

the local processor's spawning and joining can take place independently of other processors looking for work.

Wool is implemented in C using macros, inline functions and a small amount of inline assembly (on x86 and x86-64 only to emit the `exchg` instruction). The SPARC V9, x86, x86-64 and IA64 architectures are currently supported. Wool is being developed at SICS.

3 Microbenchmark characterization

Most schedulers are built around a set of queues. The activities to be scheduled are stored in queues when not running on a processor. A simple scheme is to have a single queue to where all processors store work not presently running, but this scheme suffers from several performance problems. First, if work is fine grained relative to the number of procesors, there will be significant contention for access to the queue. Second, work tends to be distributed over the processors in a random fashion, while it would be advantageous to keep related work running on the same processor to maximize the effectiveness of caches. Hence most task schedulers use distributed queues, where each processor manages their own queue(s) with occasional coordination between processors for the purpose of load balancing.

In this section we present measurements of the basic operations of the different task schedulers using a small set of microbenchmarks. Specifically, we measure the overhead of using tasks as compared to running the same computation using procedure calls.

The use of distributed queues makes creation of parallelism a two stage process: First, new tasks are spawned to the local queue and later some (typically few) of the tasks move to execute on other processors. This enables very low overhead task creation since the first step typically does not entail communiction between processors.

All experiments in this and the next sections were run on a Dell PowerEdge SC1435 dual quadcore Opteron server with 16GB of memory running Ubuntu Linux 9.04, kernel 2.28-15.

3.1 Experimental metodology

Measuring the cost of inlined tasks We have measured the cost of creating, spawning and joining with a task *on a single processor* by comparing the timings of the `fib` programs (given in figures 1, 2 and 3) when running on a single processor with that of a serial C program and dividing by the number of tasks created by the task parallel program. We have used different inputs for the different systems to arrive at execution times of a few seconds. This measures the marginal cost of a task over the cost of a procedure call in the case where the task end up being executed by the same processor that spawned it. Such tasks are referred to as *inlined* tasks in the work stealing context, and we extent the terminology to all of the systems investigated regardless of the scheduler used.

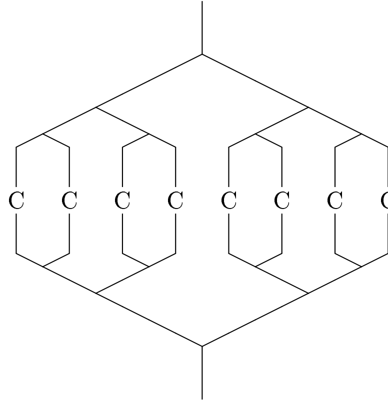

Fig. 4. The steal cost microbenchmark for 8 processors; C is an simple loop that makes no memory references

Inlining is the most common fate of a small task, especially in a work stealing implementation such as that of Cilk++ and Wool. The cost covers allocating, initializing and making the task available to the scheduler (spawning) as well as the cost of later joining with the not yet executed task (including the cost of synchronizaion and the cost of resuming its execution). We have attempted to disable optimizations that are unsafe when running on more than one processor, hence these timings include the cost of atomic instructions or memory barriers for the joining operation (sync or taskwait) although these rae not strictly necessary for a single processor execution.

Measuring the cost of stolen tasks We measure the cost of stealing or migrating a task to a different processor (core) using a microbenchmark that repeatedly spawns and joins a balanced binary tree of tasks (see Figure 4), each of which executes a simple loop C making no memory references. The size of the tree is equal to the number p of processors used; consequently, the depth d of the tree is $\log_2 p$. We have measured the scaling behavior by varying d from 1 to 3 for two, four and eight processors, respectively. To compute the cost of spawning and joining the tasks in the tree, we compare the time to execute a depth d tree on 2^d processors (the parallel run) with the timing for a depth 0 tree (that is, just the loop C) on a single processor (the sequential run); the difference is the spawn/join cost. The number of iterations of the C loop is chosen individually for each system and value of d so that the difference in time becomes 10-20%, ensuring that we really get parallel execution of the tasks. Given that, the number of repetitions R is chosen to make the running time a few seconds. C and R are of course identical in the corresponding seqential sequential and parallel runs for a single system.

3.2 Performance results from microbenchmark study

As we can see from Table 1, Wool have significantly lower spawn/join cost than the other systems, and the consequences of that can be seen in the timings for the application programs. Since these measurements are from sequential runs,

Table 1. Costs of task creation/spawn/join on a single processor and across two, four and eight processors. All costs are measured in clock cycles.

System	Spawn/join (inlined)	Spawn/join (2 cores)	Spawn/join (4 cores)	Spawn/join (8 cores)
Wool	19	2 200	5 600	10 400
Cilk++	134	31 050	73 600	110 400
Gcc	415	5 200	16 800	52 800
Icc	878	4 830	9 200	20 240
Mcc	1 005	25 760	253 920	706 560
Sun cc	915	45 000	780 000	552 000 000

the overheads are only due to book keeping and not to effects like false sharing. This also explains why some systems are very dependent on limiting task depth.

When it comes to the parallel spawn/join benchmarks, the results show how well the synchronization and communication works. The ideal case here is low absolute time together with scaling with depth of the tree rather than the size (number of spawns) since the steals/migrations closer to the roots of the tree could in principle be performed in parallel. Of the tested systems, Cilk++ appears to be closest to scaling with the depth of the tree, although the costs are among the highest in absolute terms. Both mcc, gcc and the Sun cc compiler scales worse than linearly in the number of tasks spawned while Wool scales approximately linearly. The Intel compiler scales slightly better than linearly.

The Sun compiler stands apart in this comparison. Not only are the absolute costs higher than those of the other systems, but the scaling behaviour is excessive. Further studies are needed to understand this behaviour, but a preliminary analysis is as follows: We have observed that total CPU time in the eight processor case is less than eight times the elapsed time. Thus the processors have significant idle time. This phomenon occurs when the number of avalable tasks is not much larger than the number of processors. For instance, running the benchmark with $d = 4 + \log_2 p$ brings the overhead to about five million cycles. Thus these results are probably related to the synchronization mechanisms used when accessing the task queue(s).

The microbenchmarks gives us some indications on the relative performance, but how is the effect on real application perfomance? This is investigated in the next section.

4 Application study

In this section we study and compare the performance of the six different models for five benchmarks from an early version of the Barcelona OpenMP task suite (BOTS) [8]. We ported the programs to Cilk++ and Wool, respectively, which was an easy task as these programs only used the functionality of the subset of OpenMP that is implemented in both Cilk++ and Wool. The five programs chosen are mostly taken from the original Cilk distribution (except SparseLU).

Table 2. The programs chosen to drive the performance comparison.

Name	Domain	Summary	Task size modifier
FFT	Spectral method	Calculates a Fast Fourier Transform	Size of vector before going serial. From 64k down to 16.
NQueens	Search	Finds solutions of the NQueens problem	Permitted task depth: 4, 8, 12 and 16.
Multisort	Integer sorting	Uses a mixture of sorting algorithms to sort a vector	Size of list before starting serial sort/merge. From 512k/256k down to 16/8.
SparseLU	Sparse linear algebra	Computes the LU factorization of a sparse matrix	N/A.
Strassen	Dense linear algebra	Computes a matrix multiply with Strassen's method	Size of sub-matrix before going serial. From 256 down to 16.

When this project started, we also had the programs Alignment and Floorplan from BOTS available but they either used OpenMP threadprivate variables or had other issues that made porting to Cilk++ and Wool not straightforward. Table 2summarizes the five programs used. The default workloads of each program as implemented in BOTS have been used.

In the next sections we show the relative performance of the six different task-based models/implementations. The programs are configured so that the task-depth in recursive calls can be controlled. SparseLU does not have recursive generation of tasks, so this program naturally, does not have this. Table 3 shows the compiler flags used in each case.

Table 3. Compiler flags used in the experiments.

Compiler	Flags
Serial (gcc)	-O3 -m64 -static
Wool	-O3 -m64 -lpthread -lm
Cilk++	-O3 -m64 -static
Gcc	-O3 -m64 -fopenmp -static
Icc	-O3 -m64 -openmp -openmp-link static
Mcc	-O3 -m32 -v -k
Sun cc	-O3 -m64 -xopenmp=parallel

Most of these programs, except SparseLU, are recursive in nature and one of the main objectives of this study has been to investigate as to how OpenMP and the other models fare when we allow deep recursions as it is either difficult to make automatically in the run-time system or cumbersome for programmers to do themselves. This is modified for the different programs according to the task size modifier specified in Table 2.

4.1 Performance results

Parallelism overhead The first experiment reports on the overhead created by the parallel constructs in Wool, Cilk++ and OpenMP, respectively in sequential program compared to the parallel programs with one thread. Table 4 shows the result of this measurement normalized to the sequential execution of each program. In these experiments, as in all others in this section, the programs have been executed ten times on an otherwise unloaded machine and the mean execution time taken.

Table 4. Overhead of parallel constructs for the five programs and six models.

Compiler	FFT	NQueens	Multisort	SparseLU	Strassen
Wool	0.96	0.98	0.96	0.97	0.76
Cilk++	0.93	0.78	0.93	0.96	0.76
Gcc	0.95	0.91	0.97	0.88	0.72
Icc	0.98	0.93	0.94	0.98	0.84
Mcc	0.91	0.95	0.98	0.99	0.55
Sun cc	0.73	0.89	0.92	0.99	0.94

The results are somewhat inconclusive. None of the compilers is consistently best, although Icc seems to have a robust performance in this respect. All others have really poor performance for at least some benchmark. Future studies will investigate the sources of this in more details.

Overall performance Figures 5 to 9 show the speedups relative the serial execution for all compilers using 1-8 processors. To the left in each figure is shown the default coarse grain task granularity where a cutoff in the generation of recursive tasks is set quite early. To the right (except for SparseLU) is shown a fine-grained generation of tasks for which we discuss the results in section 4.1.

When task-granularity is coarse, all compilers perform relatively well. The difference in performance can be attributed more to difference in compiler optimizations rather than in the implementation of parallelism. For Multisort, Mcc clearly outperforms all other compilers and the full reason for this remains to be investigated. Profilers show that the programs spend most of their time in the run-time system for multisort so the implementation of task scheduling is crucial here. Sun CC outperforms all other compilers for the Strassen benchmark. The reason here is superior code quality and in particular much lower branch miss prediction penalty as indicated by experiments using AMD CodeAnalyst.

Importance of task-depth cut-off The real interesting results are shown when we allow the program to generate many fine-grained tasks. In all cases, the OpenMP compilers really perform poorly when the task granularity becomes small. One of the main advantages of using a task-based model is that the user

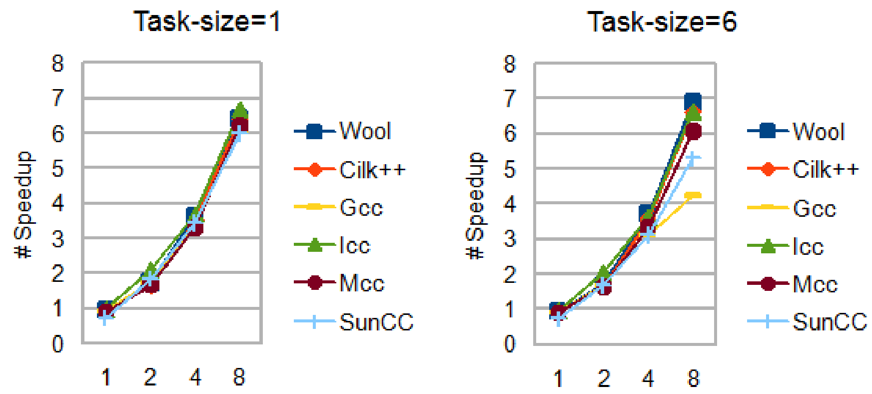

Fig. 5. FFT speedup with coarse and fine grained task granularity.

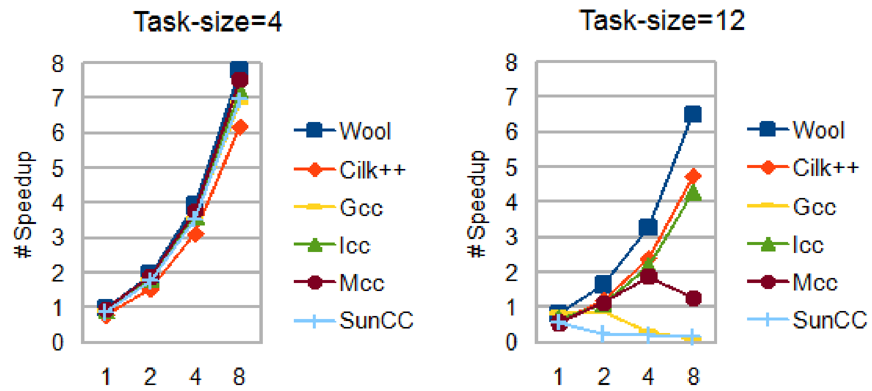

Fig. 6. NQueens speedup with coarse and fine grained task granularity.

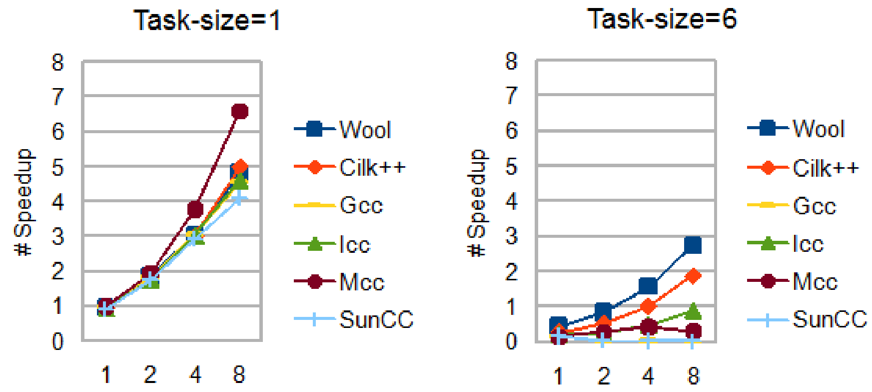

Fig. 7. Multisort speedup with coarse and fine grained task granularity.

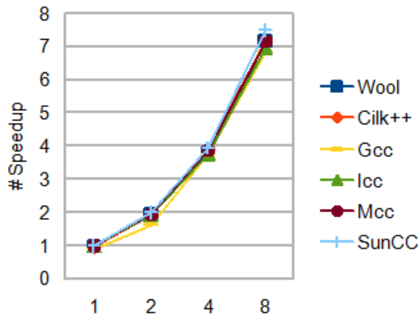

Fig. 8. Sparse LU Speedup.

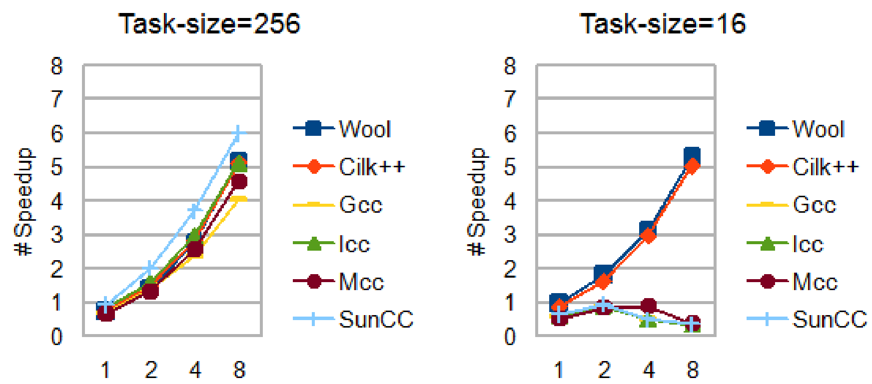

Fig. 9. Strassen speedup with coarse and fine grained task granularity.

should be able to concentrate on exposing the parallelism inherent in the application rather than trying to figure out how it would fit on this particular machine.

Furthermore Wool consistently outperforms Cilk++ for all fine-grained applications. In some cases significantly. We have paid great detail in spefically making the spawn process cheap and also to provide for a low-overhead and scalable work-stealing algorithm which pays off for programs such as NQueens and Multisort.

5 Conclusions

We have reported on performance results for a number of task-parallel programs for six different implementations of task-based parallel programming models. It is clear that the OpenMP implementations studied work well for coarse grained applications but equally clear that they fail miserably for fine-grained tasks. One of the main advantages of task-based parallelism is that the programmer can concentrate on exposing available parallelism instead of worrying on scheduling the computations on threads manually which is what you have to do with normal thread-centric models as pthreads and also to some extent with OpenMP ≤ 2.5. There is no inherent reason that OpenMP should perform this badly for fine-grained applications so we expect that this will be an area of focus for future implementations.

This study is, to the best of our knowledge, the first to compare Wool with the other major models and also to look at finer granularity of tasks. We are happy to note that the comparison puts Wool into a good position, but there are also a number of questions that we need to further study in order to understand what is happening. Why is Sun cc so increadibly bad in the micr-benchmark but reasonably effective for the coarse grained applications? Why is mcc considerably better for the multisort application? Why are the parallelism overheads varying so much for the different programs? All of these questions as well as scalability issues for larger core counts will be the focus of future studies.

References

1. Grand central dispatch. Technology Brief, 2009. Retrieved Sept 29, 2009 from http://www.apple.com/macosx/technology/.
2. E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An experimental evaluation of the new openmp tasking model. *Lecture Notes In Computer Science*, 5234:63–77, 2008.
3. Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
4. J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 2004, 2004.

5. R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216. ACM New York, NY, USA, 1995.
6. S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual conference on Design automation*, pages 746–749. ACM New York, NY, USA, 2007.
7. L. Dagum, R. Menon, and S.G. Inc. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.
8. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP '09)*, page 124–131, Vienna, Austria, September 2009. IEEE Computer Society, IEEE Computer Society.
9. Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, 2008.
10. Karl-Filip Faxén. Wool user's guide. Technical report, Swedish Institute of Computer Science, 2009.
11. International technology roadmap for semiconductors. http://www.itrs.net, 2007. Retrieved on Sept 28, 2009.
12. Olivier Stephen L. and Prins Jan F. Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs. In *Fifth International Workshop on OpenMP*, June 2009.
13. C.E. Leiserson. The Cilk++ concurrency platform. In *46th Design Automation Conference, San Francisco, CA*, 2009.
14. D. Novillo. OpenMP and automatic parallelization in GCC. In *GCC developers summit*, 2006.
15. Openmp v. 3.0 specification. http://www.openmp.org, 2008. Retrieved on Sept 28, 2009.
16. J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.
17. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible Control Structures for Parallel C/C++. In *First European Workshop on OpenMP, September*, September 1999.