A Comparison of Task Pool Variants in OpenMP and a Proposal for a Solution to the Busy Waiting Problem

Alexander Wirz, Michael Süß, and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies,
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
`wirz@student.uni-kassel.de; {msuess, leopold}@uni-kassel.de`

Abstract. Irregular algorithms are difficult to parallelize using existing OpenMP constructs. This paper concentrates on algorithms that deploy task pools, i.e., data structures for dynamic load balancing. We present several task pool variants that we have implemented in OpenMP, and compare their performance. Due to the lack of a mechanism in OpenMP to put a thread to sleep, we had to use busy waiting in our implementations. To eliminate this need, we suggest an extension to OpenMP that allows to put a thread to sleep on demand.

1 Introduction

OpenMP [1] provides powerful constructs to parallelize regular programs, i.e., programs that execute a similar set of operations on different elements of a regular data structure such as an array. Irregular applications, in contrast, are difficult to parallelize using the existing OpenMP constructs. For irregular applications, the units of work can usually not be distributed statically among a fixed number of threads, because they are created dynamically at runtime and their number depends on the given input. Moreover, it is often not possible to predict the amount of work to be done in a unit for any particular input data.

One approach to achieve dynamic load balancing is the use of task pools. A task pool is a data structure that stores dynamically created work units (tasks) to support distribution to a certain number of threads. Section 2 gives an overview about some task pool variants that we have implemented in OpenMP, and presents the results of our runtime experiments with three irregular applications using task pools: Quicksort, Labyrinth-Search and Sparse Cholesky Factorization. Performance numbers gathered with the workqueuing model proposed by Shah et al. [2] are included for comparison in this section as well.

One problem we have been confronted with during the implementation of our task pools is the lack of a suitable mechanism in OpenMP to put a thread to sleep while waiting for a condition to become true. The programmer therefore has to resort to busy waiting, which may be wasteful on the available computing resources. Fig. 1 sketches a solution to this problem by proposing an extension to OpenMP, which is spelled out in Sect. 3. Sect. 4 surveys related work, and Sect. 5 summarizes our results.

– **# pragma omp yield**:
  release the processor so that another thread can run on it
– **#pragma omp sleepuntil (scalar_expression)**:
  sleep until scalar_expression becomes true

Fig. 1. Scheduling in a nutshell

2 Task Pools

Task pools are used to achieve dynamic load balancing in irregular applications. A task pool stores tasks that are created dynamically at runtime. It also provides a set of operations that allow threads to insert and extract tasks concurrently in a threadsafe manner. The remainder of this section is organized as follows: Sect. 2.1 introduces the high level interface for the programmer used by all our task pool variants. In Sect. 2.2, the different task pool variants are described, while Sect. 2.3 highlights the most severe implementation problem we had with all variants: lack of a suitable mechanism to put threads to sleep while waiting for a condition to become true. Finally, in Sect. 2.4, we introduce three example applications that are used in Sect. 2.5 to assess the performance of the different task pool variants: Quicksort, Labyrinth-Search and Sparse Cholesky Factorization.

2.1 Application Programming Interface

All implemented task pools use the same application programming interface. This API provides functions to initialize and destroy the task pool structure, as well as to insert and extract tasks concurrently. Listing 1.1 shows an example of the relevant part of an OpenMP program that uses our API.

```
task_data_t *task_data;
tpool_t *pool = tpool_init(num_threads, sizeof(task_data_t));
task_data = generate_initial_task();
tpool_put(pool, 0, task_data);
#pragma omp parallel shared(pool)
{
  task_data_t *my_task_data;
  int me = omp_get_thread_num();
  while(TPOOL_EMPTY != tpool_get(pool, me, &my_task_data);
    do_work(my_task_data);
}
tpool_destroy(pool);
```

Listing 1.1. OpenMP program using task pools

First, a task pool must be initialized by using the *tpool_init()* function. This function must only be called once, and only by a single thread. Afterwards, the task pool can be used to store (*tpool_put()*) and extract (*tpool_get()*) tasks. The latter function blocks until it either successfully extracts a task from the pool, or discovers that the task pool is empty and all threads using the pool are idle.

Finally, function *tpool_destroy()* frees the memory used by the task pool. All task pool variants and test applications were implemented in C.

2.2 Variants of Task Pools

We implemented several variants of task pools. Some of them (*sq1*, *sdq1* and *dq8*) were ported to OpenMP from existing POSIX threads and Java implementations described by Korch and Rauber [3]. Others (*dq9* and *dq9-1*) have been developed by the authors as enhancements of the *dq8* variant. The remainder of this section explains the variants.

Central Task Queue: The simplest way to design a task pool, called *sq1*, is to use a single shared task queue. Each thread is allowed to access this queue with functions *tpool_put()* and *tpool_get()*. We used OpenMP lock variables to ensure that only one thread can access the task queue at a time. The variant has the drawback that when two or more threads are trying to access the task pool simultaneously, they have to wait for each other. Therefore, the task pool can become a bottleneck for applications that use a large number of threads or access the pool frequently. Nevertheless, this variant offers good load balancing capabilities and performs well for applications that create only few tasks or access the task pool rarely.

Combined Central and Distributed Task Queues: To reduce waiting times caused by access conflicts, the task pool variant *sdq1* uses distributed task queues. It manages a private task queue for each thread and permits only the owner thread to access the queue. Therefore no synchronization operations are needed for the private queues. An extra central queue is maintained for load balancing. Whenever a private queue is empty, the owner thread tries to fetch a task from the central queue. To ensure the exchange of tasks among the threads, the size of the private queues is limited. If a thread tries to enqueue a new task and discovers that its private queue is full, it will move the new task to the central queue.

Distributed Queues with Dynamic Task Stealing: In contrast to *sdq1*, the task pool variants *dq8*, *dq9* and *dq9-1* use multiple shared queues to reduce the possibility of access conflicts: each thread has its own private and its own shared queue. If a thread runs out of tasks in its private queue, it will take a task from its shared queue. If the shared queue is also empty, the thread will try to steal a task from the shared queue of another thread, and then return it from *tpool_get()*.

Although the task pool variants *dq8*, *dq9* and *dq9-1* are conceptually similar, they use different strategies for filling the shared queues. Like *sdq1*, *dq8* uses private queues with a limited size. If a private task queue is full, the new tasks are moved to the shared queue.

Unlike *dq8*, variants *dq9* and *dq9-1* adjust the size of the private queues dynamically, based on the state of the shared queue. The size of a private queue in *dq9* and *dq9-1* is not limited to a certain value. The private queues of these task pool variants can contain an arbitrary number of tasks. The reason is that *dq9* and *dq9-1* try to keep most tasks in the private queues to reduce the number of operations on shared queues. A thread will move a task into its shared queue in *tpool_put()* only if the shared queue is running empty. If both the private and the shared queue are empty, a new task will be inserted into the shared queue in *dq9*, but into the private queue in *dq9-1*.

Another difference between *dq9-1* and *dq9* is the point in time, when task stealing is started. While *dq8* and *dq9* do not attempt to steal tasks before a private queue is empty, *dq9-1* initiates task stealing as soon as the number of tasks in the private queue drops below a predefined threshold value. This is done to prevent the private queue from running empty.

All of these different variants are summarized in Tab. 1.

Name	**Num. Q.**	**Num. Shared Q.**	**Size of Priv. Q.**	**Task-stealing Time**
sq1	1	1	-	-
sdq1	$num\_threads + 1$	1	limited (2)	-
dq8	$num\_threads * 2$	$num\_threads$	limited (2)	priv. queue empty
dq9	$num\_threads * 2$	$num\_threads$	unlimited	priv. queue empty
dq9-1	$num\_threads * 2$	$num\_threads$	unlimited	priv. queue low (2)

Table 1. Comparison of implemented task pool variants (Q. stands for Queue, the number in braces stands for the actual number of tasks used in our tests)

2.3 Implementation Problem: Busy Waiting

The implementation of the task pools sketched in the previous section was relatively straightforward with OpenMP, but we encountered a problem for which OpenMP does not provide an adequate solution: Each time a thread tries to extract a task but detects an empty task pool, it has to wait until another thread inserts a new task. Korch and Rauber [3] solved the problem for their implementations with condition variables in POSIX threads, and the *wait()-notify()* mechanism in Java, respectively. Unfortunately, there is no mechanism in OpenMP to put a thread to sleep until an event occurs or a condition becomes true. In our task pool implementations, we therefore had to fall back on busy waiting, which results in unnecessary idle cycles. For this reason, Sect. 3 suggests OpenMP extensions to solve the problem. These simple extensions can be used to avoid busy waiting in a task pool, and are also helpful in other contexts.

2.4 Benchmarks

To compare the performance of our task pool variants, we have implemented three irregular applications: Quicksort, Labyrinth-Search and Sparse Cholesky

Factorization. Quicksort is a popular sorting algorithm, initially invented and described by Hoare [4]. The Labyrinth-Search application finds the shortest path through a labyrinth using the breadth-first search algorithm. To ensure that all labyrinth cells with the same distance from the entry cell are visited before any other cells are processed, we use two task pools. The tasks in the first pool correspond to cells with distance d from the entry cell. The second task pool is used to collect tasks (cells) with distance $d + 1$.

Cholesky Factorization is an algorithm to solve systems of linear equations $Ax = b$. It exploits the fact that a symmetric positive definite matrix A can be decomposed into $A = LL^T$, where L is a lower triangular matrix with positive diagonal elements. Using this decomposition, the original equation can be solved more efficiently. Information on Cholesky Factorization can be found, for instance, in the book by George and Liu [5]. To test our task pool variants, we have implemented only the most expensive part of Cholesky Factorization: numerical factorization. Numerical factorization computes the nonzero elements of the result matrix L. We have implemented a so-called right-looking factorization scheme. Each task computes one column of the result matrix, dividing all elements of this column by the square root of its diagonal. Then, all columns which depend on the recently computed column are updated by adding a multiple of the computed column to them.

2.5 Results

Performance measurements were carried out on an AMD Opteron 848 class computer with four processors at 2.2 GHz, and on a Sun Fire E6900 with 24 dual-core Ultra Sparc IV processors at 1.2 GHz. On the AMD system, a maximum of four threads was used, while on the Sun system, a maximum of eight threads was used. Although more threads would have been possible on the latter machine, eight processors is the maximum number that this machine supports without encountering NUMA-effects (as it consists of multiple mainboards with 4 dual-core processors each). On the AMD system, the benchmarks were compiled with the Intel C++ Compiler 9.0 using options `-O2` and `-openmp`. On the Sun Fire E6900, the Guide compiler with options `-fast --backend -xchip=ultra3cu --backend -xcache=64/32/4:8192/512/2 --backend -xarch=v8plusb` was used. We have not used the native SUN compiler, because it does not support the workqueuing extension (see next paragraph).

For comparison, we implemented Quicksort and the Cholesky factorization using Intel's proposed workqueuing model. It was first introduced by Shah et al. [2], as an integrated approach to achieve dynamic load balancing for irregular applications. Those authors suggest an OpenMP extension which allows to split the work into units (tasks) that are distributed dynamically to the threads of a program using a task queue. Since both the Intel C++ and the Guide compilers already support the workqueueing model, we implemented two benchmarks using this proposed OpenMP extension on the same set of machines. Unfortunately, we could not implement the Labyrinth-Search algorithm with this model, because we did not find a way to use two different queues and ensure that all tasks from one queue are executed before the program starts to execute tasks from the second queue.

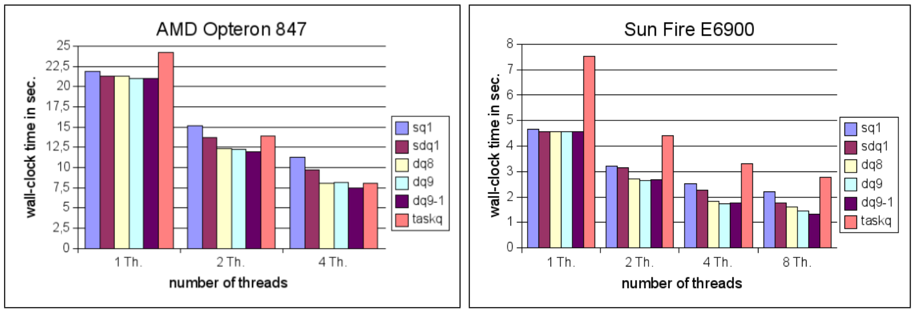

Fig. 2. Wall–clock times for Quicksort in seconds. Each time shown is the average of three runs.

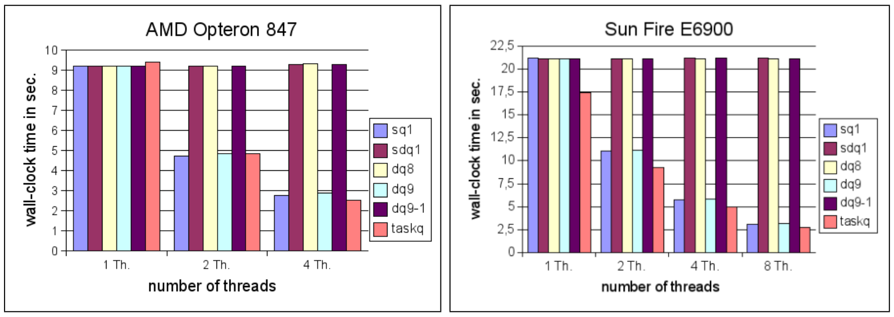

Fig. 3. Wall–clock times for the Cholesky factorization. Each time shown is the average of three runs.

Fig. 2 shows the wall-clock times in seconds for the Quicksort benchmark application with different task pool variants and the Intel taskq implementation. We used an array with 100.000.000 elements as input data on the AMD Opteron system and an array with 10.000.000 elements on the Sun Fire E6900. The results for the Cholesky factorization are shown in Fig. 3. For the Cholesky factorization, a 500x500 matrix was used as input. Fig. 4 shows the results for the Labyrinth-Search benchmark application with different task pool variants.

Our experiments indicate that the performance of different task pool variants depends on the type of application. Quicksort and Labyrinth-Search, which create a large number of tasks, achieve better performance using task pools with distributed task queues. Cholesky factorization, in contrast, generates only a few

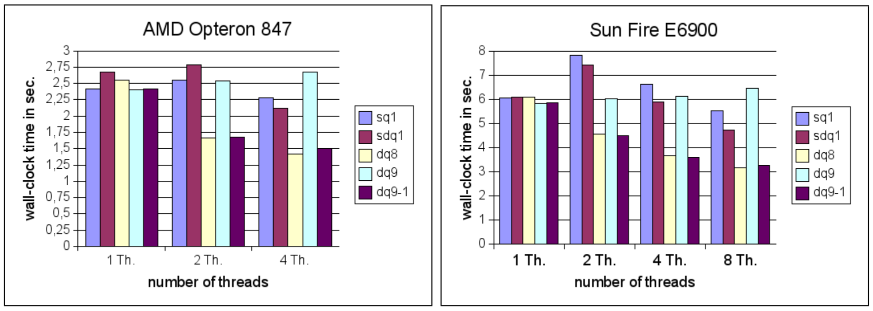

Fig. 4. Wall–clock times for Labyrinth-Search. Each time shown is the average of three runs.

tasks, and therefore good load balancing is crucial. The use of private queues turns out to be a drawback in this case, because all tasks remain in the private queues and the idle threads have no chance to fetch them. The performance of *dq9* is good, though, because this variant makes the distribution of tasks among the queues dependent on the number of tasks in the pool. If there are only a few tasks in the pool (shared queues are empty and at least one thread is idle), a new task will be inserted into a shared queue (and not, like e. g. for *dq9-1* into a private queue). If there are enough tasks in the pool, however, *dq9* will insert a new task into a private queue to avoid synchronization operations. Using this technique, *dq9* achieves much better performance than the other task pool variants with distributed queues.

Fig. 3 shows that the only task pool variant that uses one central queue (*sq1*) achieves the best performance for Cholesky Factorization. The reason is the good load balancing offered by *sq1*: all tasks are kept in one central queue, where all threads can access them. Due to the small number of tasks generated by the algorithm, a central queue does not slow down the program because the application accesses the task pool only rarely.

The bottom line from our experiments is that there is no clear winning taskpool implementation. It depends on the application, which task pool variant is suited best.

As can be seen, the performance of the task pools implemented inside the two compilers using Intels taskq is comparable to (and in some cases even better than) our implementations for the Cholesky example. When many tasks are generated and stored in the pools (as is the case for Quicksort), our optimized task pools are able to outperform the Intel implementations, though.

3 Solving the Problem of Busy Waiting

As has already been stated in Sect. 2.3, there is a problem regarding busy waiting and OpenMP. The problem is shortly rehashed on a broader scale in Sect. 3.1. Afterwards, Sect. 3.2 specifies our proposed solution, and Sect. 3.3 gives our reasons for the design. Finally, in Sect. 3.4, the specification is applied to our examples, and some ways to use the new functionality are shown. A reference implementation of the suggested changes to the OpenMP functionality can be found in a special release of the OMPi Compiler [6] that is available from the authors on request.

3.1 Problem Description

The problem of busy waiting has already been discussed by the same authors [7]. It manifests if a thread has to wait for a condition to become true before it can continue. In the case of our task pools, for instance, function *tpool_get()* is supposed to return an element from the pool, but if there is no element left, it has to wait for work to become available. The most sparing way for the computing resources to implement this waiting is to put the thread to sleep until the condition becomes true. Unfortunately, there is no functionality available in OpenMP to support the waiting, though.

As a valid workaround, the programmer may poll a condition repeatedly, thereby wasting processor time. This approach is known as *busy waiting*. To give another example, busy waiting is also required for pipelined algorithms, where a stage has to wait until a previous stage has completed its work. Busy waiting is best avoided, especially when other threads are waiting for the processor to become available, or when power consumption is an issue, e.g. in embedded systems.

Novice OpenMP programmers may resort to using locks to solve the problem. In their approach, the waiting thread tries to set an already set lock, and is put on hold as a result. As soon as work is available, a different thread will unset the lock, thereby enabling the waiting thread to continue. Although this approach often works, it is not compliant with the OpenMP specification, because the lock is unset by a different thread than the owner thread, which leads to unspecified behaviour. Furthermore, there is no guarantee that a thread waiting on a lock is put to sleep at all (busy waiting is also allowed), and therefore this approach is even more flawed.

The problem described above cannot be solved in OpenMP satisfactory as of now, since there are no directives for scheduling available. Therefore, Sect. 3.2 suggests a possible addition to the OpenMP specification that makes the suggested workarounds (busy waiting or non-compliant use of locks) obsolete. The problem has already been noticed by Lu et al. [8], who suggested the introduction of condition variables (as found in POSIX threads) in 1998. Our solution tries to combine the power of condition variables with the ease of use of OpenMP.

Let us make one more fact perfectly clear: The newly proposed functionality is not useful for the common case in computing centers today, where one processor is exclusivly available for each thread. It is intended for the more general case that multiple threads are competing for the available processors. With the advent of multi-core CPUs in common desktop systems and the expected shift to multi-threaded applications, we soon expect this case to be the dominant one.

3.2 Specification

We suggest two new directives:

#pragma omp yield
Similar to the POSIX function sched_yield (), this function tells the scheduler to pick a new thread to run on the current processor. If no new thread is available, it returns immediately. The directive provides a simple way to pass on knowledge on what is important and what not from the programmer to the runtime system and operating system scheduler. As a second new directive, we propose:

#pragma omp sleepuntil (scalar_expression)
This directive puts the current thread to sleep until the specified scalar expression becomes true (non-zero). The expression is occasionally tested by the runtime system in the background. Before each test, a flush is carried out automatically, to keep the temporary view of memory consistent with memory. An implementation of the directive is not required to wake up the sleeping thread immediately after the expression becomes true, nor does it have to wake it up if the expression becomes true and becomes false again shortly afterwards. Not all threads waiting on the same expression have to wake up at the same time either. It is unspecified, how many times any side-effects in the evaluation of the scalar expression occur.

3.3 Rationale

The *yield* directive is inspired by its POSIX counterpart, *sched_yield()*. It offers an easy to use way to influence the scheduling policies of the operating system. This can be important when computing resources are sparse and the programmer wants to optimize program throughput. An example of this would be calling the *yield* directive at the end of every pipeline step in a pipelined application, to get values through the pipeline as fast as possible.

We know of no scheduling primitive in any other parallel programming system that is as powerful and easy to use as the proposed *sleepuntil*. Thanks to the OpenMP memory model (and its ability to read variables without locking them using only *flush*, as compared to e. g. POSIX Threads), this directive is as powerful as condition variables, yet it lacks their difficult usage. The directive can be emulated by wasting time in a loop, but this would be busy waiting and wasteful to the available computing resources, as outlined in Sect. 3.1.

The proposed changes are fully backwards compatible to the existing OpenMP specification, since no behaviour of existing OpenMP functionality is altered in any way.

3.4 Application

We have emphasized in Sect. 3.1 that there is no opportunity for a parallel algorithm using taskpools to wait for a new element out of an otherwise empty pool, without constantly polling the pool. There are two approaches to solve this problem with our newly proposed directives. The first one calls the *yield* directive whenever there is no work in the pool, which will put the thread to sleep if another thread is waiting for a processor to become available. Chances are, that a different thread will produce work for the taskpool. If there is no other thread from the same application, a context switch may occur and a different application will run on the processor, allowing for a higher throughput on the machine. Finally, if there is no other thread waiting for the processor, the call to the *yield* directive will just return and no harm is done. A second possible solution is the following:

```
#pragma omp sleepuntil (!tpool_is_empty (pool))
```

This solution offers a more fine-grained control over when the thread is supposed to wake up again, as the thread will sleep until something has been put into the taskpool and not just an unspecified amount of time as with the *yield* solution. After wake-up, it is still necessary to check if the taskpool is not empty again, as no locking of any sort is involved here. The thread might have been woken up at a time when the pool was not empty, but when it tries to actually get a task from the pool, a different thread might have already popped the task.

It is difficult to measure the impact of the proposed directives, as they are most useful on fully loaded systems. We have therefore overloaded a system by starting our benchmark applications with 32 threads using the most simple taskpool sq1. The results are shown in Fig. 5.

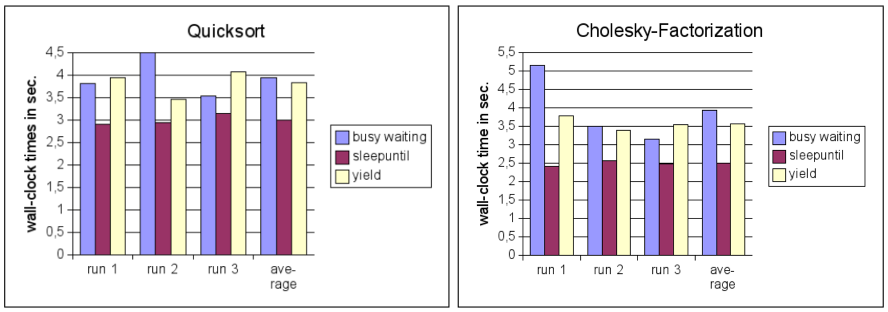

Fig. 5. The impact of the proposed directives on a fully loaded system (Sun Fire E6900 with 32 Threads running on 8 processors), measured wall-clock times in seconds over multiple runs with sq1

A different use case for both new directives is testing. When testing OpenMP compilers or performing tests for OpenMP programs, it is often useful to *force* the scheduler into certain timing behaviours that could not be tested otherwise (e. g. stalling one thread, while all other members of the team go ahead and run into a barrier). This is not possible with the present OpenMP specification (except with busy waiting again), and can be very useful to test for hard to catch errors. An example to stall execution of one thread for 100 milliseconds is shown below:

```
double now = omp_get_wtime();    /* save current time into now */
#pragma omp sleepuntil (omp_get_wtime() >= now + 0.1)
```

4 Related Work

This paper is an indirect follow-up paper to our own work on task-pools [7]. Some less advanced task pool variants were presented there, along with a first mention of the problem of busy waiting. The present paper includes more advanced task pool variants, two new example algorithms, and a proposal to solve the problem of busy waiting.

A detailed analysis of several task pool implementations with pthreads and Java threads can be found in the article of Korch and Rauber [3]. They conclude, that a combination of private and a public queues for each thread works best for their three benchmark applications.

An OpenMP extension that could help to deal with irregular problems, the workqueuing model, has been suggested by Shah et al. [2], and performance measurements for this extension have already been discussed in Sect. 2.5.

Another approach was proposed by Balart et al. [9]. They suggest to relax the specifications of the sections directive allowing a section to be instantiated multiple times. Additionaly they suggest to execute code outside of any section by a single thread. Each time this thread detects a section instance, it will insert this section into an internal queue. The section instances inserted into the queue are executed by a team of threads.

5 Concluding Remarks and Perspectives

Efficient parallelization of irregular algorithms is an ambitious goal that often can be tackled with task pools. We have presented several variants of task pools along with their implementation in OpenMP. To assess the performance of the variants, we have implemented three irregular algorithms: Quicksort, Labyrinth-Search and Cholesky Factorization. Results show that the correct selection of a task pool variant has a significant impact on the performance of an application. There is no universally best variant, but the suitability depends on the pattern of accesses to the task pool. Applications that generate many tasks and access the task pool frequently benefit from the usage of distributed private queues. Applications that access the task pool infrequently, in contrast, need good load balancing, and therefore gain more profit from a central shared task queue.

The second contribution of this paper has been a proposal to the OpenMP language committee. We suggested two directives: *yield* and *sleepuntil*. Both enable the programmer to influence the scheduling process, and to put threads to sleep on demand. By using these directives, the need for busy waiting is eliminated.

A reference implementation of the extended OpenMP functionality can be found in a special release of the OMPi Compiler [6] that is available from the authors on request. In the future, we plan to explore more applications with OpenMP, trying to find ways to improve the specification in the process. Our progress will be visible in the UKOMP project [10]. The project will serve as our testing ground for new functionality we discover to be useful, and also enables other developers to give feedback on how they like our changes.

Acknowledgments

We are grateful to Björn Knafla for proofreading the paper and for his insightful comments. We thank the University Computing Centers at RWTH Aachen, TU Darmstadt and University of Kassel for providing the computing facilities used to test our sample applications on different compilers and hardware.

References

1. OpenMP Architecture Review Board: OpenMP specifications. `http://www.openmp.org/specs` (2005)
2. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structures for parallelism in OpenMP. In: Proceedings of the First European Workshop on OpenMP - EWOMP'99. (1999)
3. Korch, M., Rauber, T.: A comparison of task pools for dynamic load balancing of irregular algorithms. Concurrency and Computation: Practice and Experience **16**(1) (2004) 1–47
4. Hoare, C.: Quicksort. The Computer Journal **5** (1962) 10–15
5. George, A., Liu, J.W.H.: Computer Solution of Large Sparse Positive-Definite Systems. Prentice-Hall, Englewood Cliffs, NJ (1981)
6. Dimakopoulos, V.V., Georgopoulos, A., Leontiadis, E., Tzoumas, G.: OMPi compiler homepage. `http://www.cs.uoi.gr/~ompi/` (2003)
7. Süß, M., Leopold, C.: A user's experience with parallel sorting and OpenMP. In: Proceedings of the Sixth European Workshop on OpenMP - EWOMP'04. (2004)
8. Lu, H., Hu, C., Zwaenepoel, W.: OpenMP on networks of workstations. In: Proc. of Supercomputing'98. (1998)
9. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos mercurium: a research compiler for OpenMP. In: Proceedings of the European Workshop on OpenMP 2004. (2004)
10. Süß, M.: University of Kassel OpenMP – UKOMP homepage. `http://www.plm.eecs.uni-kassel.de/plm/index.php?id=ukomp` (2005)