Original published: M. Hofmann and G. Rünger. Towards an adaptive task pool implementation. In *Proceedings of the IPDPS-08 Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08)*, pages 1–8. IEEE, 2008. Online available at http://dx.doi.org/10.1109/IPDPS.2008.4536477.

Towards an Adaptive Task Pool Implementation

M. Hofmann* and G. Rünger
Department of Computer Science
Chemnitz University of Technology, Germany

E-mail: {mhofma,ruenger}@informatik.tu-chemnitz.de

Abstract

Task pools can be used to achieve the dynamic load balancing that is required for an efficient parallel implementation of irregular applications. However, the performance strongly depends on a task pool implementation that is well suited for the specific application. This paper introduces an adaptive task pool implementation that enables a stepwise transition between the common strategies of central and distributed task pools. The influence of the task size on the parallel performance is investigated and it is shown that the adaptive implementation provides the flexibility to adapt to different situations. Performance results from benchmark programs and from an irregular application for anomalous diffusion simulation are presented to demonstrate the need for an adaptive strategy. It is shown that profiling information about the overhead of the task pool implementation can be used to determine an optimal task pool strategy.

Keywords: adaptive software, irregular algorithms, multithreading, parallel computing, profiling, task pools.

1 Introduction

The dynamic behavior of irregular algorithms poses the challenge for parallel programming to achieve efficient parallel implementations. In contrast to regular applications, where a static work distribution is often sufficient, irregular applications require a dynamic distribution of work and data at runtime. Especially for multithreaded shared memory programming, the task pool concept can be used to implement the necessary dynamic load balancing. An application is decomposed into several tasks, that are managed by a task pool data structure and executed in parallel by an arbitrary set of processors. The number of working threads can vary from only a few on small multicore systems to tens or hundreds on large SMP systems. Due to the progress in multicore technology, parallel platforms with high numbers of working threads (up to 32) become increasingly available.

Implementation details have a significant influence on the performance of task pools and specific properties like the task granularity or the number of working threads may require different implementation strategies. However, these specific properties strongly depend on the actual application, the current problem to be solved, and the parallel platform to be used. Adaptive software can provide the flexibility to adapt to the actual situation in advance or at runtime.

This article presents an adaptive task pool implementation that covers common task pool strategies and provides the flexibility to adapt to the needs of the actual application. The adaptivity is achieved by a flexible mapping of task queues to threads executing the program. The strategies of central and distributed task queues are special cases of this flexible mapping. With this flexible strategy, the major performance critical routine of a task pool is implemented in a new task pool implementation. Important information about the overhead of the task pool implementation caused by locking mechanisms and load imbalances are monitored.

The behavior of the new task pool implementation is investigated using benchmark programs with different workloads and an irregular application, that uses random fractal structures to simulate anomalous diffusion processes [3]. Performance measurements are presented using up to 32 working threads on an IBM p690+ 32-way SMP system. The results show that an adaptive strategy can provide optimal performance under the varying conditions of the different applications. The results also show that profiling information about the task pool implementation can be used to determine an optimal strategy with respect to the current workload.

The rest of this article is organized as follows. Section 2 lists related work and Section 3 introduces the task pool concept. Section 4 presents the adaptive task pool implementation, followed by performance results in Section 5. Section 6 concludes the paper.

*Supported by Deutsche Forschungsgemeinschaft (DFG).

2 Related Work

The concept of task pools is closely related to processing schemes using work or job queues. These and similar concepts share a lot functionality, but their specific implementations differ due to different requirements of the application area. Furthermore, a lot of work from the broad field of dynamic scheduling and load balancing algorithms is applicable to task pools.

In [11] different task pool strategies are presented, together with a discussion of implementation details like memory managers and queue organization mechanisms. The influences of synchronization mechanisms based on hardware support are investigated in [7]. The *Carbon* technique [12] aims to provide hardware support for fine-grained parallelism on chip multiprocessors by introducing additional processor instructions that implement task pool functionalities. Work stealing concepts are used to overcome load imbalance problems when having multiple task queues [1]. The work stealing algorithm of *Cilk* [4] makes use of private double ended queues, where the queue owner can operate on one side while other threads steal work from the other side. Profiling of task-based applications based on task pools was proposed to detect performance critical program behavior and for estimating parallel execution times [8]. Task pool concepts were also used for implementing irregular algorithms with OpenMP [14, 16]. Additionally, the task construct is one of the biggest additions to the upcoming OpenMP 3.0 specification [13] and enables the creation of explicit tasks that can be executed in parallel.

The *Task Pool Teams* approach [5] extends a library based task pool implementation to distributed address space programming by introducing special communication operations and separate communication threads. Various other parallel environments for shared and distributed memory programming make use of concepts similar to that of task pools. In [10] a hierarchical framework with language and runtime support is introduced to provide independent load balancing policies for subsets of threads. This approach can be used to apply different load balancing strategies to different computational tasks. The *LilyTask* system [15] is based on a distributed task pool and introduces task groups and task relations to allow an easy mapping of computational problems to tasks. The *PMESC* library [2] is designed for implementing task-parallel programs on distributed memory systems. Processors use local queues to manage tasks and dynamic load balancing is achieved by redistributing tasks on request. The C++ based parallel programming language *Charm++* [9] represents an object oriented approach with a message driven execution model and user selectable load balancing strategies.

3 Task Pools

The concept of task pools can be used to achieve dynamic load balancing in parallel programming, especially for multithreaded shared memory systems. The computational work is divided into tasks which are inserted and stored in a task pool data structure. A fixed number of threads is responsible for extracting and executing the tasks. During the execution, new tasks can be created and inserted into the task pool. Dependencies between tasks are usually handled by inserting a task not until all its dependencies are fulfilled. The task pool approach is especially appropriate for implementing irregular algorithms and applications, in which computational work is dynamically created at runtime and strongly depends on the input data.

3.1 A General Task Pool API

An application programming interface (API) for a library based task pool implementation includes the following operations:

- `tpool_init`: initialize the task pool
- `tpool_destroy`: destroy the task pool
- `tpool_put`: insert a task into the task pool
- `tpool_get`: extract a task from the task pool

Independent from the actual implementation of the task pool, this API is capable of being used directly with POSIX-threads as well as with OpenMP parallel directives. Listing 1 shows pseudocode for implementing a parallel computation using task pools.

Listing 1. Using the task pool API with OpenMP

```
1  tpool *tp = tpool_init(...);
2
3  tpool_put(tp, ...); /* insert initial task(s) */
4
5  #pragma omp parallel shared(tp)
6  {
7    tpool_task tpt;
8    int th_id = omp_get_thread_num();
9
10   while (tpool_get(tp, th_id, &tpt) != TPOOL_EMPTY)
11     tpt.task_routine(tpt.argument);
12 }
13
14 tpool_destroy(tp);
```

In conjunction with thread libraries like POSIX-threads, lines 5-12 have to be replaced with appropriate thread creation and termination mechanisms (e.g.: `pthread_create` and `pthread_join`). The `tpool_get` routine is repeatedly

called by all working threads in parallel (line 10). If tasks are available, the routine extracts a task from the pool and the task is executed (line 11). Otherwise, the calling thread is blocked until a task is available for extraction. The task pool is empty, when all threads are waiting in `tpool_get`. In this case, all threads are unblocked (`tpool_get` returns `TPOOL_EMPTY`) and the parallel computation is finished.

3.2 Task Pool Strategies

Several strategies for implementing task pools are available [4, 11, 16]. Depending on the organization of the tasks in queues, one can coarsely distinguish between *central* and *distributed* task pools. A central task pool stores all tasks in one public queue to which all threads have access. This results in almost perfect load balancing, since idle times for threads can only occur if the public queue runs out of tasks. The major disadvantages are waiting times that occur during extract and insert operations, because access to the queue is synchronized to prevent access conflicts. The opposite strategy is the distributed task pool, where every thread owns a private task queue. This enables concurrent accesses to the different queues and reduces the waiting times, e.g. for extract operations. However, having many queues naturally leads to load imbalances and introduces the problem of selecting appropriate queues for the insert operations.

Choosing an optimal strategy strongly depends on the actual situation. Large numbers of short tasks and high numbers of threads can lead to increased serialization due to the synchronized queue access. Load imbalances may occur, e.g. when having only a few number of tasks or varying task sizes. Fixed task sizes or additional information about the tasks can help to achieve a balanced distribution of tasks to queues. Solutions proposed for the problem of imbalanced workloads include combinations of central and distributed task pools as well as support for dynamic task stealing [1, 11].

4 An Adaptive Task Pool Implementation

In addition to the task pool strategies mentioned above, we propose a general implementation that is able to cover strategies like central and distributed task pools and provides the flexibility to adapt to the needs of the actual situation.

4.1 Adaptive Functionalities

The adaptive task pool implementation consists of a set of queues and a set of threads and uses a mapping that assigns threads to queues. This results in a variable relation between threads and queues; central and distributed task pools are included as special cases. The number of queues can be changed and threads can be reassigned to other queues. Together with the ability to transfer tasks from one queue to another, the implementation provides the functionality to adapt the task pool strategy to the actual situation. Profiling is used to gather information about the behavior of the task pool at runtime and helps to detect the actual times spent in locks for queue accesses or waiting times due to empty queues.

4.2 Implementation Details

The task pool implementation is based on POSIX-threads and uses mutex and condition variables to implement locks and barriers. Besides the necessary data structures, one of the most important parts of a task pool implementation is the `tpool_get` routine. This routine is called frequently (at least once for every task) and therefore it is critical to performance when there is a large number of tasks. Also, this routine is responsible for implementing the task pool strategy and for taking control of the overall processing of the task pool (e.g. exit if the task pool is empty, handle empty queues, etc.). The pseudocode in Listing 2 outlines the general functionality of the `tpool_get` routine.

Listing 2. General functionality of the `tpool_get` routine.

```
1  int tpool_get(tpool *tp, int th_id, tpool_task *tpt)
2  {
3    lock_queue(...);
4
5    while (queue_is_empty)
6    {
7      if (all_queues_empty) perform_task_pool_exit();
8
9      if (switch_queue) restart_tpool_get();
10
11     conditional_wait_with_timeout();
12   }
13
14   *tpt = ...  /* extract task from the queue */
15
16   unlock_queue(...);
17 }
```

Access to the queue is protected by using a locking mechanism (line 3 and 16). While the actual queue is empty, the processing of `tpool_get` is blocked until the whole task pool is empty (line 7) or a queue switch is requested (line 9). By using a waiting mechanism with timeout (e.g., `pthread_cond_timedwait`), the calling thread is repeatedly activated while waiting for new tasks. This enables a monitoring of the times spent waiting on an empty queue (line 11) and provides the ability to react on increased waiting times (e.g. perform a queue switch). The times spent on acquiring access to a queue (line 3) can also be monitored

(across consecutive calls to `tpool_get`) and provide useful information about serialization effects due to queue accesses. Similar information about times spent on acquiring access to a queue can be gathered in the `tpool_put` routine. Furthermore, special strategies for choosing a queue for inserting new tasks can also be implemented in `tpool_put`.

5 Performance Results

To investigate the influences on the performance of the task pool implementation, we use benchmarks that allow us to control properties like the size of the tasks and the number of tasks. A basic task of the benchmark computes the value of the number Pi up to a fixed precision. The size of the task is adjusted by repeating this computation several times. Additionally, results are shown for an irregular application, that uses random fractal structures for simulating diffusion processes. Performance measurements are performed on a single 32-way SMP node with 32 Power4+ processors and 128 GB main memory. Every working thread is executed by a single dedicated processor.

5.1 Static Workload

For measurements with a static workload, a certain number of tasks (with fixed task size) is inserted into the task pool before the execution starts. To achieve a balanced workload, the total number of tasks is always chosen to be a multiple of the number of threads and at least 10 tasks per thread are provided.

Figure 1 shows speedups depending on the task size for different numbers of threads extracting and executing tasks from a central task pool (one queue). For task sizes down to a value of 800 (number of times Pi is computed per task) the achieved speedup corresponds to the number of utilized threads. Only for the case with 32 threads, the speedup remains slightly below the ideal value of 32. A further decrease of the task size leads to a constant decrease of the speedup values. This effect is most significant when the number of threads is large. For very small task sizes (< 10), all speedups fall below a value of 2. The inefficiency of this case is caused by synchronized accesses to the public queue and illustrates the main disadvantage of the central task pool.

Figure 2 shows speedup values depending on the task size with 32 threads and for different numbers of queues. Using one queue corresponds to a central task pool, while 32 queues represent a distributed task pool with a separate queue for each thread. Other numbers of queues represent a stepwise transition between these two strategies. The results clearly show that the strong dependency between performance and task size can be eliminated by increasing the number of queues. Almost constant performance

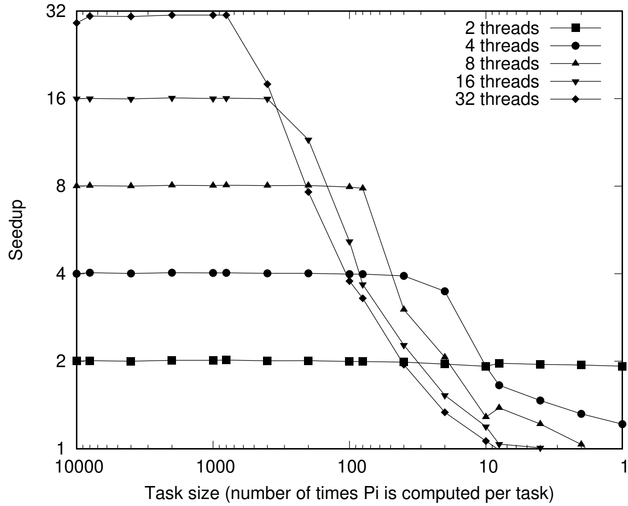

Figure 1. Speedups depending on the task size, using a static workload and a central task pool (one queue) with different numbers of threads.

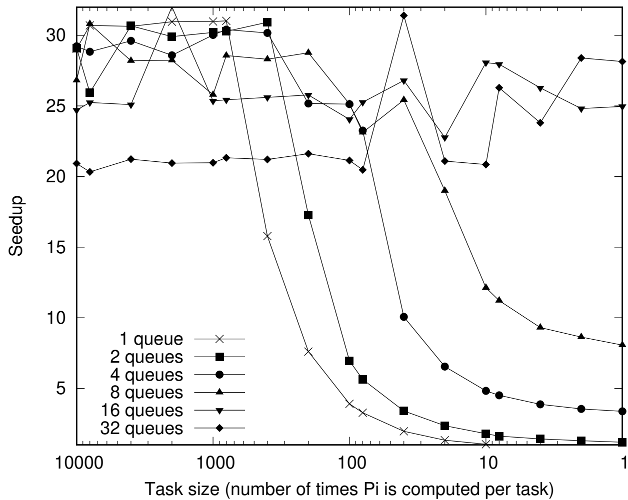

Figure 2. Speedups depending on the task size, using a static workload and 32 threads with different numbers of queues.

is achieved when using at least 16 queues. But, for big task sizes and a high number of queues the performance is not optimal. Using low numbers of queues (up to 8) leads to better speedup values for big task sizes, but results in a significant loss of performance when the task size becomes too small. The measurements show that a fixed strategy will not provide an optimal solution for the different situations and an adaption to the actual conditions is needed.

5.2 Dynamic Workload

For investigating the behavior of our task pool implementation with dynamic workload, we use the synthetic benchmark proposed in [11]. The work structure of a task is illustrated with pseudocode in Listing 3.

Listing 3. Task routine of the synthetic benchmark.

```
1  void synthetic_task(int i)
2  {
3    if (i > 0)
4    {
5      compute_Pi(10 * f);
6      tpool_put(synthetic_task, i - 2);
7      compute_Pi(50 * f);
8      tpool_put(synthetic_task, i - 1);
9      compute_Pi(100 * f);
10
11   } else compute_Pi(100 * f);
12 }
```

Depending on the parameter i, a task either performs a number of computations and creates two additional tasks or performs only computations. The computational parts of the task are represented by the `compute_Pi` routine and consist of repeated computations of the value of Pi. The parameter of the `compute_Pi` routine specifies the number of times Pi is computed per call. The parameter f is used to control the amount of computations and determines the size of the task. Initially, a number of $k = 12$ tasks with parameters $i = k - 1 \ldots 0$ are inserted into the task pool. With $k = 12$, the synthetic benchmark creates a total number of 1204 tasks.

Figure 3 shows speedup values depending on the task size factor f and the number of queues, using 32 threads. The results show the influences of a dynamic workload on the parallel performance. Similar effects of dynamic workload are typical for irregular applications. The overall speedup values vary in the range of 2 to 32. But, considering only the optimal number of queues for a given task size, the speedup increases to about 19 in the worst case (small task sizes).

In Figure 4 the amount of time spent in acquiring the lock for queue accesses is shown. Figure 5 shows the amount of waiting time while a queue is empty. The values represent average times of all threads measured in `tpool_get` as

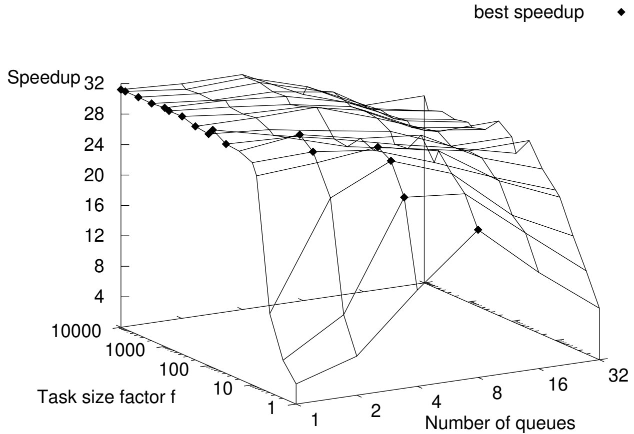

Figure 3. Speedups depending on the task size factor f and the number of queues, using a dynamic workload and 32 threads. The symbol ♦ denotes the best speedup for a specific task size factor f.

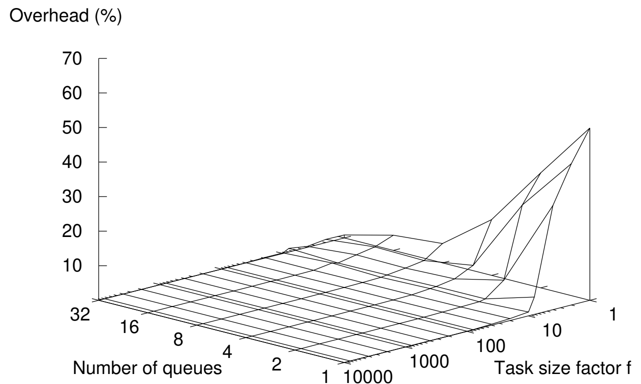

Figure 4. Amount of time (percentage of total time) spent in acquiring locks depending on the task size factor f and the number of queues, using a dynamic workload and 32 threads.

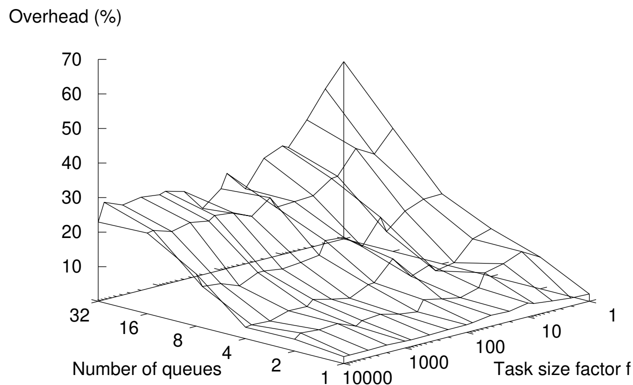

Figure 5. Amount of time (percentage of total time) spent waiting while a queue is empty depending on the task size factor f and the number of queues, using a dynamic workload and 32 threads.

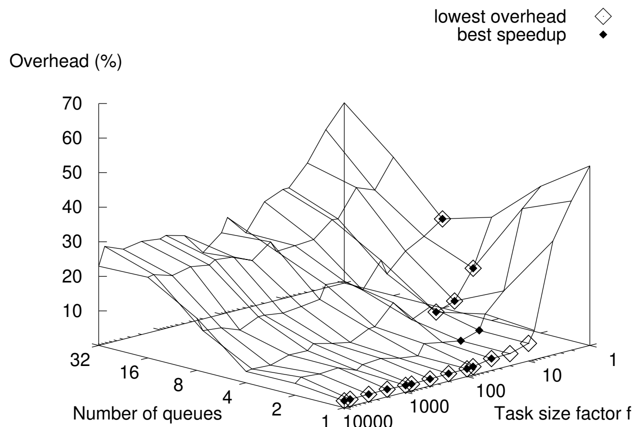

Figure 6. Total amount of time (percentage of total time) spent in acquiring locks and waiting while a queue is empty depending on the task size and the number of queues, using a dynamic workload and 32 threads. The symbols ◇ and ♦ denote the number of queues with the lowest overhead and the best speedup for a specific task size factor f.

shown in Listing 2. The overhead caused by locks increases when having small tasks ($f < 10$) and few queues (< 8). Otherwise, this overhead is negligible. The time spent waiting while queues are empty increases when using higher numbers of queues. A decision for an optimal number of queues has to pay attention to both effects.

In Figure 6 the total overhead together with the lowest result for each task size is shown. The number of queues that leads to a minimum overhead decreases from 8 to 1 for an increasing task size. This corresponds to the results of the best speedup values as shown in Figure 3. The information about the time spent in locks or waiting at empty queues can be used to adapt the strategy of a task pool implementation to the actual situation. The resulting adaptive task pool implementation provides more flexibility than other fixed strategies and releases the user from choosing a suitable strategy in advance. Furthermore, there exists the possibility to adapt to changing conditions even at runtime.

5.3 Master Equation Application

As an example for an irregular application, an implementation of the master equation approach for random Sierpinski carpets is used. The application simulates diffusion processes and uses random fractal structures to model the structural properties of real materials such as aerogels, porous rocks, or cements. The master equation approach calculates the probability distribution for the location of a test particle (*random walker*) on a two-dimensional random fractal structure (random Sierpinski carpet) at different time steps. The carpet consists of basic work units called *iterator*. An iterator is recursively constructed using a set of predefined *generator* patterns. The iterator level specifies the number of recursion steps used for the construction and determines the size of the iterator. Figure 7 shows an example with three different generators of size 3×3 (left) and a randomly created iterator of level 2 (middle). The black tiles of the iterator represent valid locations for the walker.

In every time step, the walker is allowed to move from one tile to a neighboring tile. The probability distribution over all valid tiles at time step t is calculated from the probability distribution at the previous time step $t-1$ using the following master equation:

$$p_i(t) = \sum_{j \in <i>} G_{ij}\, p_j(t-1) + (1 - L_i)\, p_i(t-1)$$

The probability $p_i(t)$ of tile i in time step t is calculated by accounting for the gain and loss of probability caused by the movement of the walker. The sum is over the set of all neighbors $< i >$ of tile i. The gain factors G_{ij} describe the probability for the walker to arrive at tile i coming from tile j and $L_i = \sum_{j \in <i>} G_{ji}$ is the overall loss of tile i. The gain and loss factors are calculated once depending on the structural properties of the iterators and the behavior of the walker. The simulation starts with a carpet consisting of a single iterator and a delta distribution for the probabilities of the location of the walker (probability 1 at the starting position and 0 otherwise). New iterators are appended to the boundaries of the carpet as the simulation proceeds and the probability distribution spreads over the carpet. A more detailed introduction to the parallel application can be found

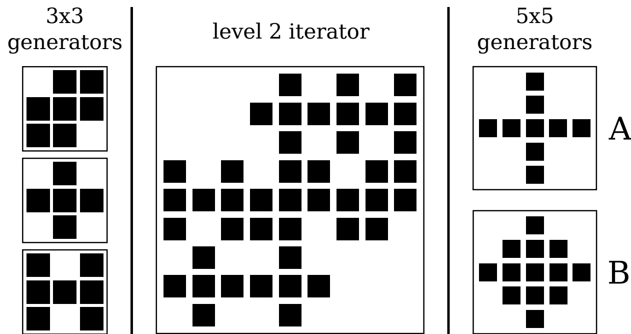

Figure 7. Three sample generators of size 3×3 (left) and a randomly created iterator of level 2 (middle). Generator sets A and B used for the performance measurements (right).

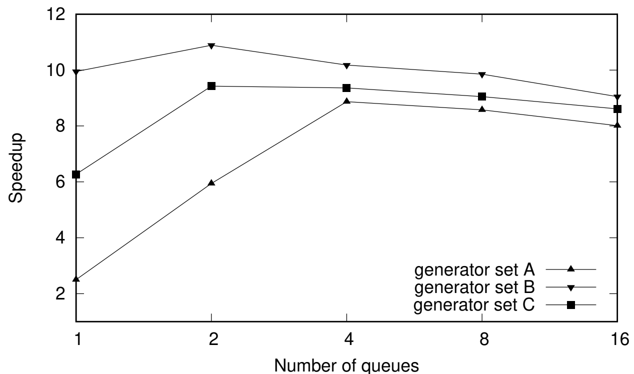

Figure 8. Speedups of the master equation application depending on the number of queues, using 16 threads and different sets of generators.

in [6]. The following results were obtained using a parallel implementation based on shared memory programming with Pthreads only.

In every time step, the probabilities of the tiles have to be updated. For every iterator, the task of calculating the new probabilities is inserted into the task pool. The tasks can be executed in parallel by several threads; only the creation of new iterators and the extension of the carpet requires mutual exclusion. The amount of work required for updating an iterator strongly depends on the chosen iterator level and the set of generator patterns. The following results were obtained for simulating about 8000 time steps using iterators of level 3 and different generators of size 5×5. The task pool implementation uses 16 threads for executing the tasks and an equal distribution of tasks to queues when multiple queues are used.

Figure 8 shows speedup values depending on the number of queues for three different sets of generators. Sets A and B consist of the single generators shown in Figure 7 (right). Set C consists of nine different generators that are randomly chosen when creating new iterators. The generators of set C are similar to the ones of sets A and B and the average number of valid tiles is close to the number of valid tiles in generator B. The different generator sets have a strong effect on the performance of the task pool implementation. The higher numbers of valid tiles in generator sets B and C increase the task sizes and lead to better speedup values in comparison to generator set A. The random structure of the iterators created with generator set C causes load imbalances due to varying task sizes and achieves lower speedups in comparison to the more regular structure with generator set B. The optimal number of queues also depends on the chosen generators. The smaller number of valid tiles in generator set A decreases the task size and requires a higher number of queues. With generator sets B and C, the best speedup values are achieved using only two queues.

For generator set C, the total overhead of the task pool implementation is shown in Figure 9. The results with the lowest overhead and the best speedup are marked for the different numbers of elapsed time steps. The very high overhead of about 60 % at the beginning of the simulation decreases during the progress of the simulation. However, the overhead for using only one queue (central task pool) remains significantly high (about 60 %). The random structure of the iterators leads to varying task sizes. For high numbers of queues, this results in an increased overhead due to load imbalances. As already shown with the benchmark program in Section 5.2, the number of queues with the lowest overhead corresponds to the number of queues with the best speedup values. After 8000 time steps, the lowest overhead of about 20 % is achieved using two queues. The measured overhead of about 20 % indicates that there is still room for further optimizations of the task pool implementation.

6 Summary

In this paper, we have investigated adaptive task pool strategies and have shown that choosing an appropriate task pool strategy is essential for achieving good performance. Fixed strategies like central or distributed task pools are suitable for specific situations only. By using a variable number of queues for managing the tasks, our task pool implementation is able to adapt to a wide range of task sizes or numbers of working threads. To ease the problem of selecting the appropriate number of queues we have shown that information about the overhead of the task pool implementation (monitored by the task pool implementation itself) can be used to determine an optimal solution.

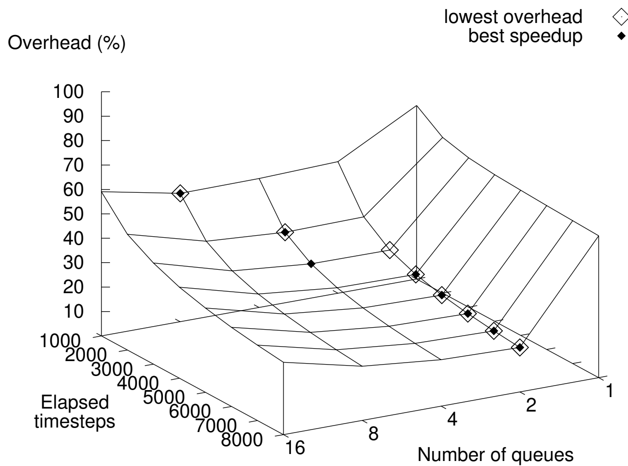

Figure 9. Total amount of overhead (locks and waits) depending on the number of elapsed time steps and the number of queues, for the master equation application using 16 threads. The symbols ◇ and ♦ denote the number of queues with the lowest overhead and the best speedup for a specific number of elapsed time steps.

Acknowledgements

All measurements are performed on the IBM Regatta p690+ System JUMP at the John von Neumann Institute for Computing, Jülich, Germany. `http://jumpdoc.fz-juelich.de/`

References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[2] S. Crivelli and E. R. Jessup. The PMESC Programming Library for Distributed-Memory MIMD Computers. *Journal of Parallel and Distributed Computing*, 57(3):295–321, 1999.

[3] Do Hoang Ngoc Anh, P. Blaudeck, K. H. Hoffmann, J. Prehl, and S. Tarafdar. Anomalous diffusion on random fractal composites. *Journal of Physics A: Mathematical and General*, 40(38):11453–11465, 2007.

[4] M. Frigo, C. E. Leiserson, and K. 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. ACM, 1998.

[5] J. Hippold and G. Rünger. Task Pool Teams: A Hybrid Programming Environment for Irregular Algorithms on SMP Clusters. *Concurrency and Computation: Practice and Experience*, 18(12):1575–1594, 2006.

[6] K. H. Hoffmann, M. Hofmann, G. Rünger, and S. Seeger. Task Pool Teams Implementation of the Master Equation Approach for Random Sierpinski Carpets. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *LNCS*, pages 1043–1052. Springer, 2006.

[7] R. Hoffmann, M. Korch, and T. Rauber. Performance Evaluation of Task Pools Based on Hardware Synchronization. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 44. IEEE Computer Society, 2004.

[8] R. Hoffmann and T. Rauber. Profiling of Task-Based Applications on Shared Memory Machines: Scalability and Bottlenecks. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference*, volume 4641 of *LNCS*, pages 118–128. Springer, 2007.

[9] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. *SIGPLAN Notices*, 28(10):91–108, 1993.

[10] V. Karamcheti and A. A. Chien. A Hierarchical Load-Balancing Framework for Dynamic Multithreaded Computations. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17. IEEE Computer Society, 1998.

[11] M. Korch and T. Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[12] S. Kumar, C. J. Hughes, and A. 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. ACM, 2007.

[13] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Draft 3.0 Public Comment*, October 2007.

[14] M. Süß and C. Leopold. Implementing Irregular Parallel Algorithms with OpenMP. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *LNCS*, pages 635–644. Springer, 2006.

[15] Y. Tang, T. Wang, and X. Li. The design and implementation of lilytask in shared memory. *SIGOPS Operating Systems Review*, 39(3):52–63, 2005.

[16] A. Wirz, M. Süß, and C. Leopold. A Comparison of Task Pool Variants in OpenMP and a Proposal for a Solution to the Busy Waiting Problem. In *Proceedings of the International Workshop on OpenMP*, 2006.