A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms

Matthias Korch Thomas Rauber

Martin-Luther-Universität Halle-Wittenberg
Institut für Informatik

E-mail: {korch, rauber}@informatik.uni-halle.de

Correspondig author:

Matthias Korch

Martin-Luther-Universität Halle-Wittenberg
Fachbereich Mathematik und Informatik
Institut für Informatik
D-06099 Halle (Saale), Germany

E-mail: korch@informatik.uni-halle.de
Phone: ++49 345 5524719
Fax: ++49 345 5527033

Abstract

Since a static data distribution does not give satisfactory results for parallel irregular algorithms, there is need for a dynamic distribution of data that can be adapted to the current runtime behavior of the algorithm. *Task pools* are data structures which can distribute data dynamically to different processors.

This paper discusses the characteristics of *task-based algorithms* and describes the implementation of selected types of task pools for shared-memory multiprocessors. Several task pools have been implemented in C with POSIX threads and in Java. Results of these implementations measured on three different shared-memory systems are shown for a synthetic algorithm and the parallel hierarchical radiosity method.

Key Words: Task Pools, Dynamic Task Scheduling, Irregular Algorithms, Hierarchical Radiosity, Performance Evaluation, Threads

A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms

Matthias Korch Thomas Rauber

Martin-Luther-Universität Halle-Wittenberg
Institut für Informatik

{korch, rauber}@informatik.uni-halle.de

26th February 2002

Abstract

Since a static data distribution does not give satisfactory results for parallel irregular algorithms, there is need for a dynamic distribution of data that can be adapted to the current runtime behavior of the algorithm. Task pools *are data structures which can distribute data dynamically to different processors.*

This paper discusses the characteristics of task-based algorithms *and describes the implementation of selected types of task pools for shared-memory multiprocessors. Several task pools have been implemented in C with POSIX threads and in Java. Results of these implementations measured on three different shared-memory systems are shown for a synthetic algorithm and the parallel hierarchical radiosity method.*

1. Introduction

Designing parallel algorithms for irregular problems is difficult, because it is not possible to predict the amount of work connected to a given part of the input data. Therefore there is no good strategy available to create an initial data distribution that minimizes the communication during the execution of the algorithm. To use all available processors efficiently, irregular algorithms must either allow data to be transferred between processors at runtime or assign computations to a processor only when the processor becomes idle and makes a request.

One way to design irregular algorithms for shared-memory multiprocessors is to split the algorithm into several types of *tasks* which are used as the minimum unit of parallelism. Every task is associated with a quantity of data and the work needed to process these data. Tasks are stored in a common data structure, which is called *task pool*. Some tasks for data from the input set are initially stored in the task pool, and then every processor removes tasks from the pool and processes them until all task have been executed. During the execution of a task, new child tasks may be created and inserted into the task pool for a later execution.

Task pools offer an easy and reasonable way to design parallel algorithms for irregular problems. They can be used as a universal approach to these problems. But since they hardly take advantage of locality, often methods which exploit special properties of the problem give better results. For example, methods performing iterative steps could use cost estimates extracted from earlier steps to readapt the data distribution after each interation [30].

Parallel algorithms that use task pools can be described by an abstract model. Using this model, the runtime behavior of these algorithms can be characterized by *task graphs*, and it is possible to use *task grammars* for the description of the algorithm itself. Thus it can be seen that executing a task-based algorithm leads to the problem of scheduling a directed acyclic graph (DAG) to multiple processors dynamically. This problem is $\mathcal{NP}$-hard [4].

The designer of a task pool can attack this problem by introducing heuristic methods that try to reduce the latency of the schedule. Another possibility is to simply ignore this problem. This increases idle time but allows the task pool operations to be implemented very efficiently. This paper follows the second approach by minimizing the number of instructions of the task pool operations and choosing data structures which allow to reduce contention on shared data.

Implementations of task pools usually use central or distributed queues to store tasks. If distributed queues are used, there should also be some mechanism to transfer tasks between these queues, so that the work load can be balanced.

If parts of the task pool data structures are shared by several processors, synchronization must be used to avoid race conditions. This paper describes some possible ways to re-

duce the number of calls to synchronization operations and the time processors spend waiting to acquire locks.

Since allocating and freeing objects in main memory is expensive, one should always try to reduce the number of such system calls. This can be done by re-using memory blocks or by allocating larger blocks which hold several objects. Because task pools use dynamic objects to represent task instances and typically large numbers of task instances with short execution times are used to achieve a good distribution of work load, saving system calls strongly improves the performance.

This paper describes several types of task pools which have been implemented in C with POSIX threads and in Java. These implementations have been evaluated on three different shared-memory systems: a Linux PC, a Sun Enterprise 420R and two Sun Fire (3800 and 6800). Results are shown for a synthetic algorithm and the *radiosity* application from the SPLASH-2 application suite [38].

The results for both algorithms show that from the task pools implemented *dynamic task stealing* provides the best scalability. Synchronization overhead and waiting times of such task pools can be reduced by using private and public queues. Memory managers can significantly improve the performance. As a consequence of optimizing Java Virtual Machines, the results for the Java implementations are more disputable than the results for the implementations in C. While the synthetic algorithm gives repeatable results, the *radiosity* application is harder to evaluate due to its nondeterministic character.

The rest of the paper is organized as follows: In Section 2 an overview of task based algorithms and their representation by graphs and grammars is given. Section 3 then describes the types of task pools that were chosen to be implemented, and Section 4 handles general implementation issues. After this, Section 5 briefly compares the potential of the two programming languages C and Java that have been used in our work. The task pools we have implemented are introduced in Section 6, and the machines and algorithms investigated are described in Section 7. Section 8 presents results for the synthetic algorithm and the radiosity application. Related work is presented in Section 9. Finally, Section 10 concludes.

2. Task-based algorithms

To formally describe algorithms that use task pools, a model named *task-based algorithm* can be used. In this model the algorithm provides several *task types*. Every task type consists of a set of instructions from the programming language that is used, and it can have several parameters that specify the data to be processed. When the algorithm is executed, *instances* of these task types (which we will often call *tasks* in the following for simplicity) are created and stored in a common data structure which is called *task pool*. Task instances take arguments which correspond to the parameters of the underlying task type.

Task-based algorithms work in two phases. During the first phase an initial set of tasks is created from the input set and stored in the task pool. This phase is therefore called *initialization phase*. The initialization phase can be done sequentially by a single processor or in parallel by multiple processors. Its execution time is usually very short compared to the over-all execution time of the algorithm. For that reason it may often be appropriate to execute this phase on a single processor. The operation to insert initial tasks into the pool is named `init()`.

The second phase is called *working phase*, because it computes the solution of the algorithm from the initial task set and usually takes nearly all of the total computation time. To achieve good performance, it is important that all available processors take part in this phase and that the idle time of the processors is minimized. The working phase is organized as a loop that all processors execute in parallel. In this loop each processor requests a task from the task pool by executing the operation `get()`, which the task pool provides for this purpose. If a task is returned, the processor will execute it. Otherwise the processor will exit from the loop. Since this process is common to all task-based algorithms, the task pool may provide an operation that implements the complete working phase.

When a task is executed, it can create new child tasks by executing the task pool operation `put()`. Thus every task that is executed in a task-based algorithm and that is not an initial task has exactly one parent task and can have several child tasks. Initial tasks differ from that in not having a parent.

2.1. Representation by graphs

These hierarchical dependences can be described by a *task graph*. It contains a node for every task, and a directed edge is drawn between two nodes if the target node is a child of the source node. The resulting graph is a forest of trees, the roots of which are the initial tasks. Particularly, this graph is directed and acyclic.

If there are data dependences between tasks, they can be visualized by introducing *dependence edges*. These directed edges connect two nodes if the source node provides data needed by the target node. The resulting graph is called *dependence graph*. In this simple case dependences between two tasks must not introduce circles into the graph. Otherwise deadlock will occur.

In practice there might be more complex dependences between tasks when tasks are waiting for certain conditions on shared variables at arbitrary times of their execution. In this case dependence edges must be labeled with the associ-

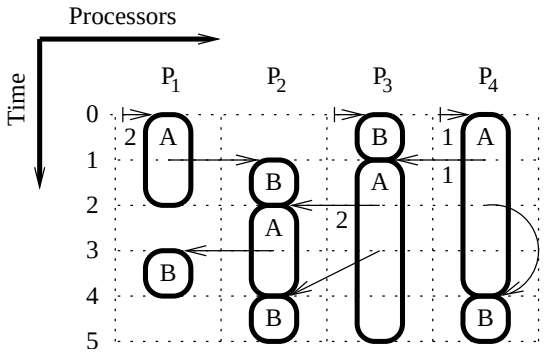

Figure 1. Example for the visualization of a schedule.

ated conditions, and the order in which these conditions are generated and checked must be visible. In a very complex algorithm it will be necessary to include the data flow inside of the tasks into the graph. Checking such a graph for deadlocks leads to the *halting problem* that cannot be decided [34].

Specific runs of a given task-based algorithm may produce different task graphs even if the same input is used. Such algorithms are called *nondeterministic*. In contrast to this, *deterministic* task-based algorithms always produce the same task graph when the same input is used.

Dependences give another way to classify task-based algorithms. First, there are algorithms with no dependences between tasks at all. One example is the Barnes-Hut method for n-body simulations [30]. Such algorithms create a large number of initial tasks that depend only on data that was provided before the initialization phase, and no child tasks are created. This class of algorithms is called $\mathfrak{D}_0$. Some other algorithms only use hierarchical dependences between tasks that are caused by the creation of child tasks. The according class of algorithms is $\mathfrak{D}_H$. The hierarchical radiosity method is an example [13, 30]. All other algorithms may have arbitrary dependences. This class will be named $\mathfrak{D}_\infty$. Obviously, $\mathfrak{D}_0$ is a subset of $\mathfrak{D}_H$ which itself is a subset of $\mathfrak{D}_\infty$.

When a task graph is drawn in the plane, the geometric representations of nodes and edges can be used to visualize the temporal progress or schedule of the algorithm. The *schedule* assigns a processor, creation time, starting time and termination time to every task. For visualization, the coordinates of the plane are used to represent processors and time. The geometric extent of a node then determines starting and termination times of the node as well as the processor assigned. The creation time of a node is determined by the source coordinates of the edge that leads from the parent to that node. Figure 1 shows an example.

Repeated runs of a task-based algorithm that use the same input will usually produce different schedules. The reason for this is that the order in which tasks are executed and the assignment of tasks to processors depend on the execution times of the tasks. But these execution times are influenced by the current scheduling decisions of the operating system. There are also other sources of noise that influence the execution times of tasks, like caches or concurrent accesses to limited resources, for instance main memory or other I/O hardware.

2.2. Task grammars

Grammars can be used to describe not only specific runs of a task-based algorithm but the algorithm itself. A *simple task grammar* consists of three sets that contain the available task types, the task types that can be used to create initial tasks and the productions that indicate which task instances each task type can create and in which order this would be done. The productions can be derived from the sequence of operations that each task type identifies.

More complex grammars may be used to include runtime aspects. Such *runtime grammars* consist of a set of available task types, a subset of task types that can be used to create initial tasks, a set of possible arguments for tasks, a set of possible values of shared variables and, finally, a set of productions. The productions are more complex than those of simple task grammars. There may be different productions for different arguments or values of shared variables, and the productions may additionally specify running times between two events.

The grammar

$$G = [\{A, B\}, \{A\}, P, \{1, 2\}, \emptyset]$$

with productions

$$\begin{aligned} P = \{&B \rightarrow \{1\}, \\ &A(1) \rightarrow \{1\}\ [A(1)|A(2)]\ \{1\}\ B\ \{2\}, \\ &A(2) \rightarrow \{1\}\ B\ \{1\}\} \end{aligned}$$

is a simple example for a runtime grammar. The corresponding algorithm uses two task types A and B, but only tasks of type A can be created as initial tasks. The integers 1 and 2 may be used as arguments to some tasks, and no shared variables are used. Tasks of type B simply terminate after one time step. The behavior of tasks of type A depends on their argument. If 1 is passed as the argument, one task of type A will be created after one time step. The argument of this child task is chosen from 1 or 2. After a second time step, tasks of type A with the argument 1 always create one task of type B and finally terminate after they have proceeded for two further time steps. For argument 2, a task of type A creates a task of type B after one time step and terminates after another time step.

As we have seen before, in practice it is impossible to specify the execution time of a task exactly, because noise is introduced by the operating system, other user processes and hardware. However, to describe an algorithm, it is often sufficient to use exact values of any needed time basis. If the grammar is used for scheduling decisions by the task pool, then modeling the execution times by probability distributions [31] or fuzzy sets [7] might do better.

2.3. Usable parallelism and running time of task-based algorithms

Given a task or dependence graph, it is hard to tell how many processors should be employed. It can be shown by constructing an example that algorithms exist which need as many processors as the graph contains tasks to achieve optimal running time. Such algorithms need only twice the execution time of the biggest task. Furthermore, it is possible to show that for any given dependence graph there is a schedule that utilizes as many processors as the task graph has leaves (see [18] for details).

If a task-based algorithm is executed on a single processor, the running time of the working phase is given by the sum of the execution times of all tasks. Thereby the execution time of a task includes the time to request this task from the task pool. Since all tasks must be executed, the algorithm cannot run faster, and the processor will never be idle, because after one task has completed there must either be at least one runnable task or no task in the task pool. If there were only tasks in the pool that cannot be executed because some dependence is not met, deadlock has occurred. When the task pool is empty, the algorithm has completed.

When there are no dependences between tasks – which is the case for algorithms from $\mathfrak{D}_0$ –, only the initially available tasks have to be executed. This implies that all tasks that are stored in the task pool are always runnable. If the tasks can be ideally balanced, no idle time needs to occur. Thus, if there are p processors, a task-based algorithm from $\mathfrak{D}_0$ can be executed in time

$$O\left(\frac{r_{seq}}{p} + \max_{v \in V}\{r(v)\}\right),$$

where r_{seq} is the running time of this algorithm on a single processor, V is the set of all created tasks, and $r(v)$ is the execution time of task v.

An optimal schedule for a given dependence graph with hierarchical or even arbitrary dependences and an unlimited number of processors can be obtained by applying the ASAP (*as soon as possible*) method. Here a task will be executed right after it has been created. This is possible, since the number of processors is unlimited and therefore there always is a processor available if needed.

The *static* scheduling of a given task graph to a limited number of processors is $\mathcal{NP}$-hard. But there are many efficient approximation algorithms [4]. However, since a task pool does usually not know the resulting task graph during the execution of a task-based algorithm, it has to solve a *dynamic* scheduling problem. This lack of information causes that for a limited number of processors the optimal schedule can only be approximated by heuristics.

In practice the number of processors available is usually very small compared to the size of the task graph. Actually, task-based algorithms are designed to create large task graphs consisting of small tasks in order to achieve a good balancing of the work load.

3. Types of task pools

This section describes several types of task pools, variants of which have been implemented in C and Java. While implementation details are the subject of Sections 4, 5 and 6, this section concentrates on the high-level description of some types of task pools, some of which were chosen to be implemented.

The objective for the design of our task pool implementations has been to provide universal data structures that can be used with any task-based algorithm. This implies that no knowledge about the algorithm may be presumed.

The main goal of our implementations has been to reduce the total execution time of the task pool operations. To achieve this, we have tried to reduce the number of instructions that these operations consist of and to avoid expensive function calls. We also have attempted to reduce the number of concurrent accesses to shared data structures in order to reduce the conflict rate and implicated waiting times.

The implementations presented in this paper cover *central*, *randomized*, *distributed*, and *combined central and distributed* task pools, and also task pools with *dynamic task stealing*. Many other implementations may be thought of, a few of which are outlined in [18].

Since shared-memory multiprocessors were selected as target systems, the thread model has been employed to implement the task pools. It provides multiple threads of control that share a common address space. Threads can be used in C with, for example, the POSIX thread library [5], which was used in this study. The other programming language we have used is Java. Java was designed as a multithreaded language [24]. It provides all necessary classes and mechanisms to develop shared-memory applications. Additional libraries are not required.

3.1. Central task pools

Central task pools use a single central queue to store tasks. This queue is accessed by all processors concurrently.

When a processor accesses the central queue, it must use *mutual exclusion* to protect the queue in order to avoid race conditions.

Mutual exclusion is available with all POSIX thread libraries, which provide *mutex variables* for this purpose. Java programmers can use `synchronized` blocks or methods.

However, waiting times occur if two queue operations are issued simultaneously. The number of access conflicts increases with the number of processors.

3.2. Randomized task pools

A way to improve the performance of central task pools is to introduce additional central queues. Since the queues are not assigned to processors, all accesses to these queues must use mutual exclusion.

When a new task is created, it is inserted into a randomly chosen queue. To remove a task, all queues are queried in randomly chosen order until a task has been found or all queues have been visited.

In the case that there are more queues than processors, there is a chance that even if all processors are performing an access simultaneously, no two processors choose the same queue. For example, the number of queues could be bound to the number of processors by a constant factor ≥ 1. But even in this case the probability that no two processors choose the same queue decreases when the number of processors is increased (see [18]). Besides, the `get()` operation becomes very expensive in this case, because the number of queues that must be queried in this operation increases.

3.3. Distributed task pools

Distributed task pools avoid access conflicts by not sharing any data between processors. Each processor uses its own queue to store tasks and performs only accesses to its local queue. Therefore each processor can only process those tasks that were assigned to it in the initialization phase. This corresponds to a static data distribution.

Without knowledge of the algorithm and the task types used, the task pool cannot estimate the cost of tasks. Thus the initial task distribution will be imbalanced in most cases. But this approach has the advantage of not needing any synchronization operations, since no shared variables are used, and it allows to evaluate the performance improvements achieved by a dynamic data distribution.

3.4. Combined central and distributed task pools

To overcome the problems of the implementations mentioned above, a combination of these methods can be used. To do so, each processor is assigned a local queue that it can use exclusively. In addition, a central queue is used for load balancing. A processor then adds tasks to the central queue when the size of its local queue exceeds a specified threshold. Whenever a processor runs out of tasks, it transfers tasks from the central to its local queue.

In this approach, mutual exclusion is only needed for the central queue. But the central queue may become a bottleneck when the number of processors increases. Therefore the threshold must be carefully chosen to find a good tradeoff between synchronization and load imbalance.

3.5. Dynamic task stealing

The most promising approach is dynamic task stealing. It uses local queues for each processor, but allows processors to access foreign queues. A processor then uses its local queue until it gets empty. When this happens, it tries to *steal* tasks from another processor by removing tasks from this processor's queue.

To avoid race conditions, mutual exclusion must be used for all queue accesses if there is only one queue per processor. This increases the number of instructions executed in every task pool operation, which implicates longer execution times. But since stealing is only done when the queue of a processor is empty, there are very few simultaneous accesses to a particular queue, and the over-all waiting time is small.

Randomized local pools

The number of simultaneous accesses to a particular queue can be reduced by introducing additional queues per processor. These may be accessed similarly to central randomized task pools. When a processor is performing an operation on any local pool, it selects one of the queues of the local pool at random. To avoid race conditions in the stealing process, mutual exclusion must be used with every access. Waiting times can be reduced further by altering the queue when acquiring a lock for a certain queue fails.

However, increasing the number of queues per processor not only reduces waiting times but also increases the overhead needed to administrate these queues. In this particular case, selecting a queue at random may be expensive compared to the execution times of the task pool operations of standard dynamic task stealing.

Private and public queues

If two queues per processor are used, one can reduce simultaneous accesses to a queue, waiting times, and synchronization overhead by applying different access rights to both queues.

In order to achieve this, one of the queues is used as a *private* queue. Only the local processor is allowed to access it. Therefore no mutual exclusion is needed for the private queue and all accesses to it will be very fast. For the purpose of stealing, each processor is assigned a *public* queue. Following some strategy, the owner transfers tasks from its private to its public queue. When a processor runs out of tasks, it can steal some from another processor's public queue. Since there may be simultaneous accesses, there must be mutual exclusion for the public queue.

If the processors mainly work on their private queues, synchronization operations rarely have to be executed. Only when occasionally a public queue is accessed, the overhead for locking is needed, and only then waiting times may occur.

Principally, two oppositional strategies for filling the public queues can be thought of. A *greedy* processor that uses the first strategy would try to keep most tasks for itself as long as possible. Tasks are transferred to the public queue only when the public queue is (nearly) empty. Using the second strategy, a *generous* processor would hold nearly all of its tasks in the public queue and only remove a few tasks from it if the private queue gets empty.

Both strategies are comparable concerning the costs of the task pool operations. While the greedy processor must often check if the public queue is empty, the generous processor very often has to move tasks between its local queues. Since this task transfer can be implemented by a few pointer operations, it does not increase the access costs considerably. On the other hand, the greedy strategy might lead to additional idle time, since stealing processors may find some queues empty even though there are many tasks stored in the according private queues.

Heuristics for stealing

Waiting times can be reduced by decreasing the number of simultaneous accesses to queues. One way to achieve this is to keep the number of stealing operations small. Therefore using a simple heuristics to steal a large amount of work may be worth the costs.

Such heuristics can be used, for example, when there are many hierarchical dependences between tasks and the owner processes its local queues in *last in first out* (LIFO) order. That means that this processor always uses the same ends of its queues to enqueue and dequeue tasks, respectively. As a result, the corresponding task graph is processed in depth-first order. If now tasks are stolen from the opposite end of the queue that is not used by the owner, there is a good chance that the stolen task will create a large subtree. In the best case this task is one of those that were created in the initialization phase.

Other approaches may attempt to steal several tasks at once in order to gather a large amount of work. Then the number of tasks to be stolen may be determined by a constant number, a constant factor, or even the number of processors.

4. General implementation issues

4.1. Ending the working phase

The working phase ends when all tasks have been executed. To verify this condition, it is not sufficient to check if the task pool contains no tasks. It is still possible that a task that is in execution at the time of the query creates new child tasks, which themselves can create large subtrees of tasks. If then a processor had already decided to leave the working phase, its processing power would be lost for further computations.

Because of this, there must be some mechanism to decide if an idle processor may leave the working phase or if it should wait for new tasks to be created. Which mechanism is appropriate depends on the task pool implementation used.

The most simple case is given if a distributed task pool is present that does not use dynamic task stealing. Since all queues are private, newly created tasks can never be accessed by other processors than their producers. Therefore a processor may leave the working phase as soon as its private queue is empty.

Most other implementations must use a different approach. When there are central queues, the idle processors must check all these central queues. In case of dynamic task stealing it is sufficient to visit only a few neighboring queues. If now all processors agree that there are no more tasks left, the working phase is completed. Thus a processor which could not find any task to execute keeps waiting until either new tasks are created or all other processors reach the same state.

This mechanism can be put into practice with the help of *conditional waiting*, which the POSIX thread libraries provide by the concept of *condition variables*, and which is included in Java as the `wait()-notify()` mechanism.

4.2. Implementation of queues

From the task pools' point of view, queues are data structures that store a set of objects. They must provide operations to insert a given object and to extract an arbitrarily chosen object.

The order in which the objects are removed is not important for the functionality of the task pool, but it becomes more interesting when heuristics shall be used to dynamically optimize the schedule. For nondeterministic algorithms the order in which tasks are executed can also influence the resulting task graph.

Another important effect of the execution order on all task-based algorithms is the impact on the maximum memory required. When a task graph is executed, the tasks that are currently being stored in the task pool define a border line through the task graph that divides the tasks whose execution has already been started or even finished from those that have not been created yet. Now, if the task graph is processed depth-first, the maximum number of tasks to store is equal to the depth of the task graph. If instead a breadth first search is used, the breadth of the task graph determines the space needed.

Since the execution times of the task pool operations will increase with the complexity of the underlying data structures, it is important to find simple and efficient queue implementations. Usually single- or double-linked lists, or arrays with one or more index pointers are appropriate. Also lists of arrays are conceivable.

Compared to lists, arrays have the advantage that memory for all items of the array is allocated with a single system call when the array is created. Furthermore, manipulation of the index pointers can be done with fewer instructions than are necessary to insert or delete elements of a list. Another point is locality, which will be discussed in Section 4.5.

On the other hand, arrays only have a limited size. If a queue is implemented by a single array, the size of the array must be increased when the number of objects to be stored exceeds the size of the array. In some cases this implicates that the data of the array must be copied to a second, larger array. Otherwise, if the array was not enlarged, the task pool would set limitations to the task-based algorithm which would not be acceptable in practice.

A good trade-off is found in a list of arrays. Instead of enlarging the array that has become too small, in this case a new array is linked to the preceeding one, and new objects are stored in this new array.

In order to avoid race conditions, often mutual exclusion is used to protect queues as a whole. In these cases usually no simultaneous manipulations are allowed. But it is possible to implement queues that can be accessed simultaneously by multiple processors when multiple locks per queue are used. A simple and efficient LIFO queue that only needs two locks and allows an enqueue and a dequeue operation to be executed in parallel is presented in [23].

If lock granularity is reduced further to the level of single tasks, queues can be implemented which allow two simultaneous dequeue operations. This is of importance for task pools with dynamic task stealing. But since for these queues a dequeue operation needs to acquire a total of three locks, the task pool operations become expensive even if no waiting times occur.

The problem of concurrent accesses to a shared data structure is well known. Therefore attempts have been made to develop data structures that are *lock-free* (e. g. [35]) or *non-blocking* (e. g. [14, 23, 27]). In our investigations such data structures have not been implemented, because they rely on machine-dependent primitives like COMPARE & SWAP.

4.3. Mutual exclusion

Besides conditional waiting, which was used to detect the end of the working phase, mutual exclusion is the most important synchronization mechanism for task pools. It must be used to protect queues or even single tasks when concurrent operations are performed.

Usually, mutual exclusion is realized by *locks*. A processor that wishes to access a protected object must acquire the attached lock before it is allowed to read or manipulate the object. After the processor has finished the manipulation, it must release the lock. The implementation of the lock must ensure that no two processors can possess the lock at the same time.

Since some task pools need to lock objects in every `put()` or `get()` operation, the overhead required for mutual exclusion may significantly influence the total execution time of the algorithm.

Because locks are commonly used in multi-threaded programs, all thread libraries or multi-threaded languages provide mechanisms for locking. The POSIX thread libraries provide locks as special objects that are called *mutex variables*. Separate `lock()` and `unlock()` operations can be performed on these variables. In the Java programming language, implicit locks are attached to each class or object. There are no separate operations for acquiring and releasing these locks. Instead `synchronized` blocks and methods are used to implicitly create a correct pair of these operations.

These locking mechanisms provided by libraries or languages often are based on complex data structures. This is necessary in a multiprogrammed environment to save the processing time of waiting threads for other threads or processes, respectively. Synchronization mechanisms for such environments are discussed in [37].

When a task-based algorithm is the only process running on a selected target system, other approaches can be considered. Since using more threads than processors increases scheduling overhead because more context switches must be performed, often exactly as many threads as there are processors are used. In this case, the operating system can assign a separate processor to every thread. But in this situation, suspending waiting threads does not improve the performance. Therefore, if there are the same numbers of threads and processors, threads can wait actively by polling a shared variable in a loop. Locks on this basis are called *spin locks* [21].

4.4. Memory management

Since tasks are the minimum unit of parallelism, typical task-based algorithms create plenty of small tasks that must be stored in the shared memory. Thus lots of small memory blocks must be allocated and freed during the working phase.

If a system call is executed for each such action, optimal performance can never be reached. This is because system calls are usually more expensive than calls to user functions. In addition to that, system calls may create a bottleneck in the case that the operating system executes system calls sequentially or uses central free-lists for all threads or processes, respectively.

Therefore task pools should always be accompanied by a memory manager that reduces the number of system calls. The task pool then requests memory blocks from this memory manager and returns used memory blocks back to it for a later re-use.

The memory manager can use various approaches to reduce the number of system calls. The most important of all is to re-use memory blocks. Another important strategy is to allocate several objects in advance by a single system call.

To re-use memory blocks, free blocks are collected in free-lists. If a processor later requests a block of the same type, it can be removed from one of the free-lists.

Since free-lists are nothing but queues, the memory manager can be organized similar to the task pools. The free-lists can be either *central*, *randomized*, *distributed*, *combined central and distributed*, or use *stealing*.

The free-lists are usually implemented as linked lists. To link the memory blocks, special items can be used that store a pointer to a free memory block and a pointer to the next item. A solution with much less complexity links the memory blocks directly by storing a single pointer inside of each block. In this case the memory manager must ensure that all memory blocks are large enough to store this pointer.

4.5. Locality

All of the current high-performance machines are equipped with caches. But the fast access times for cache data can only be exploited if the application programs are designed to show *locality*.

If a program features *spatial locality*, accesses to adjacent memory cells by the same processor will be speeded up. On a multiprocessing system locality in space may also have negative effects if multiple processors share data in their caches (*false sharing* [11, 32]).

A task pool may either increase or decrease spatial locality explicitly. If the task queues or free-lists are implemented by arrays, their elements will occupy adjacent memory cells. When linked lists are used, their elements will be arbitrarily located in memory. In order to avoid spatial locality, task blocks may be surrounded with dummy elements which are never accessed.

Temporal locality will improve the performance of both single- and multiprocessing systems as it accelerates repeated accesses to identical memory cells by the same processor.

The edges of the dependence graph of a task-based algorithm show where tasks re-use data that has been provided by other tasks. If we assume that tasks perform complex computations which supersede the arguments given as input, after a task has completed always one of its child tasks should be executed. Thereby the output of the task is re-used as the input of the child. This strategy leads to a depth-first search of the task graph. On the other hand, if the child tasks do not significantly change their initial cache state, the output of a task could be re-used several times if more than one of its child tasks are processed right after its completion.

But in general a depth first search implemented by a LIFO queue will usually take better advantage of locality compared to a FIFO queue realizing a breadth first search. If tasks are executed in FIFO order, a child inserted into the queue will not be executed before all tasks have been processed that were being stored in the queue when the child was created. All these tasks potentially overwrite the cache data that have been moved into the cache by the parent of the child. Using LIFO order, at least one of the children created by a task is executed directly after its parent. If a FIFO queue is used, a child can only be executed immediately after its parent if at the time of the creation of the child the queue was empty.

5. Potential of C and Java

C is a programming language that is close to hardware. C programs may be written very compact, but they tend to become unreadable and error-prone then. Provided that appropriate libraries are used, “everything” can be done with C. The extension C++ also provides very complex but flexible classes. Multithreaded programming is not part of the language but is available with, for example, the POSIX thread library. C programs are compiled to native code and run very fast on the dedicated target machine.

In contrast, Java has been designed to be a platform independent programming language that allows the development of interactive applications for the Internet. The syntax of Java is close to C, but elements that often caused mistakes were omitted. Great importance was attached to keep the language simple and easy to learn. Java is fully class-based and multithreaded. To be platform independent, Java programs are compiled to intermediate code, which can be

interpreted on any target machine on which a *Java Virtual Machine* is available.

Though by now there are very fast virtual machines, they cannot meet the performance of native C code. But besides, there are differences in the structure of the programming languages which give C some advantages over Java.

First of all, in Java all variables either have a basic type, like `int` or `char`, an array type, or are *object references*. These object references can be compared to pointers to class instances in C++. No compound type, like `struct` in C, exists. Instead instances of classes must be used. This makes efficient use of memory more complicated.

An example for this are local variables. In C the memory of a local variable that has a compound type is allocated on the stack by incrementing the stack pointer when the scope of the variable is entered. Since in Java class variables are only object references, memory on the stack is allocated only for a pointer. The object must be created explicitly by calling the `new` operator, which then allocates the memory needed.

Apart from that, in Java it is not possible to allocate memory for several objects at once. While in C it is easy to allocate an array for a compound type, an array of an object type only provides memory for the references and each object must be allocated separately in Java.

Another important point is the recycling of task instances. If all task types use the same compound type in C, used task instances can be collected in free-lists and can be re-used later easily. Yet in Java, different task types would use incompatible classes. This makes it difficult for a universal task pool that has no knowledge about task types to recycle them, because there must be separate free-lists for each task type in order to be able to efficiently implement the allocation operations.

Except for memory management, the synchronization primitives of Java are much less flexible than those of POSIX threads. At first, mutual exclusion is done in C by two separate function calls for locking and unlocking that can be placed at any desired position in the source code. In Java `synchronized` blocks must be used that insert a pair of similar instructions implicitly into the intermediate code.

Conditional waiting also sets limitations to the Java programmer. POSIX threads only requires that the function call to suspend a waiting thread is executed when the associated mutex variable is locked. The call to wake up one or several threads can be done at any place in the source code. In contrast to POSIX threads, Java additionally requires the wake-up calls (`notify()`) to be placed inside a `synchronized` block on the object to wait for. This introduces an unnecessary bottleneck if the thread that generates the condition does not need synchronization on this object for its manipulations.

But even though the synchronization primitives of Java are less flexible than those of C, they allow to develop synchronization classes which can be used just like the C mechanisms [24]. But using such classes creates higher overhead.

6. Implementation

This section gives a brief overview of the task pools and memory managers we have implemented in C and Java. A complete list of all memory managers implemented in C is displayed in Table 1. Because of the difficulties outlined in Section 5, no memory managers have been implemented in Java. The task pools implemented in C and Java, respectively, are contrasted in Tables 2 and 3.

6.1. Interface of the libraries

The task pools and memory managers implemented in C comply with the layout displayed in Figure 2. Task types are implemented as C functions, and task instances are represented by instances of a compound type that stores a pointer to the function that implements the type of the task instance and a pointer to a fixed-size memory block that contains the arguments of the task instance.

The initialization of the task pools is done by the function `tpool_create()`. It can be called by multiple threads, but a single call by one thread is sufficient. The number of threads, `n`, and the maximum space needed to store arguments for task instances are taken as arguments. The function `tpool_finalize()` destroys the task pool. It takes no arguments and a single call is sufficient.

To get along with only one function call, the initialization and the finalization must be executed sequentially. The disadvantage of this approach is that these sequential phases can decrease the possible speed-up if their execution time is long.

During the initialization phase of the task-based algorithm, task instances can be created by the function `tpool_init()`. `tpool_put()` is used by task instances to create new child tasks during the working phase. Both functions identify the calling thread by an integer argument, supposing that all threads using the task pool are numbered from 0 to $n-1$. Additionally, they need to know the task type and the arguments of the task instance, which are given in form of a pointer to a function and a pointer to a data structure, respectively.

The working phase is executed by a call to `tpool_run()` with the thread number as argument. This function is identically implemented for all task pools. It is realized by a loop in which task instances are removed from the task pool with `tpool_get()`. If the call to `tpool_get()` was successful, the task instance is

```
// interface of the task pools

typedef void (*tpool_function_t)(int, void *);

void tpool_create(int n, int arg_size);
void tpool_finalize();

void tpool_init(int tid, tpool_function_t f, void *arg);
void tpool_put(int tid, tpool_function_t f, void *arg);

void tpool_run(int tid);

// task pool internal data structures and functions

typedef struct
{
        tpool_function_t function;
        void *argument;
        ⋮   // pool specific data
} tpool_task_t;

⋮   // pool specific data

int tpool_get(int tid, tpool_function_t *f, void **arg);

// interface of the memory managers

void tpool_init_memory(int task_size, int arg_size);
void tpool_finalize_memory();

void *tpool_alloc_arg(int tid);
void tpool_free_arg(int tid, void *arg);

void *tpool_alloc_task(int tid);
void tpool_free_task(int tid, void *task);
```

Figure 2. Layout of the C task pools.

```
class TaskPool
{
        public static class Task
        {
                ⋮   // pool specific data
                public void run(int tid) {}
        }

        ⋮   // pool specific data

        public TaskPool(int n) {}

        public void init(int tid, Task T) {}
        public void put(int tid, Task T) {}

        private Task get(int tid) {}

        public void run(int tid) {}
}
```

Figure 3. Layout of the Java task pools.

executed and the loop is continued. Otherwise the function returns. Care must only be taken to decouple the working phase from the initialization phase. This is necessary to avoid corruption of the data structures if `tpool_init()` is allowed to write to unprotected non-local queues.

The memory managers are initialized by the function `tpool_init_memory()` with the maximum sizes of the data structures for task instances and arguments as arguments. The finalization is done by calling the function `tpool_finalize_memory()`. Both functions must always be paired. Successive or even concurrent calls to one of these functions are not allowed. But it is permitted to use multiple initialization/finalization pairs if necessary.

Memory for task instances and arguments can be allocated by the functions `tpool_alloc_task()` and `tpool_alloc_arg()`, respectively. To free task instances or arguments, `tpool_free_task()` or `tpool_free_arg()` must be called, respectively.

The layout of the Java task pools is shown in Figure 3. They use classes derived from the class `Task` to represent task types. These derived classes must overwrite the method `run()` in order to implement the desired functionality of the task type. Task instances are represented by instances of the classes derived from `Task`. Arguments are stored in fields of the derived classes by an appropriate constructor that every derived class that takes arguments must implement.

The task pools are initialized by creating an instance of the corresponding `TaskPool` class. In doing so, the number of threads using the task pool must be given as an argument to the constructor. No explicit finalization is

necessary, because the garbage collector will free all allocated memory automatically when the task pool object is not longer referenced.

Similar to the C implementations, there are two methods `init()` and `put()` to insert task instances into the task pool during the initialization and working phase, respectively. The working phase is executed by the method `run()`, which removes tasks from the pool with the help of the method `get()`.

6.2. Memory managers implemented in C

Various memory managers have been implemented in C to investigate the performance improvements that can be achieved by optimizing the memory operations of task pools. A complete list of all memory managers that have been implemented is shown in Table 1.

In order to measure the variation in performance caused by the memory managers compared to the case that no memory manager is used, a dummy memory manager called `bs` has been implemented, which capsulates the operating system functions `malloc()` and `free()` by the memory manager interface.

Besides recycling of freed objects, allocating several objects at once is another important strategy to improve the performance. All memory managers that use this strategy are named "`bk`". The memory managers which only rely on recycling are named "`l`".

On the basis of the types of free-lists used, the memory managers implemented can be classified into three types which all use free-lists to recycle objects and thus try to improve the performance. The first of these types only uses central free-lists. The names of these memory managers do not contain any prefixes. The second type only uses distributed free-list, which is indicated by the prefix "`d`" in their names. The last type implemented uses distributed free-lists and additional central free-lists for balancing. The memory managers of this type are named with the prefix "`c`".

Two ways of linking the objects in the free-lists have been implemented. The first uses special items to link objects, and the second approach uses the memory space inside of freed objects to store a pointer to the next object. The names of the memory managers which do not use items are tagged with the suffix "`f`".

The memory managers which allocate several objects at once either allocate large fixed-size arrays during the initialization (no additional prefix), or allocate blocks for a small number of objects when required (prefix "`g`"). The elements of these arrays or blocks are either directly returned to the application by successive `tpool_alloc_*()` function calls (no additional suffix), or all of the elements are inserted into the corresponding free-lists right after the block or array has been allocated (suffix "`o`").

Except for `clf`, all memory managers that use both central and distributed free lists link the objects stored in the free-lists by additional items. But not all of these memory managers use a central free-list for the items. Therefore all memory managers which do use a central free-list for items have the suffix "`b`" added to their name.

In the case that central and distributed free-lists are used, there are two different implementations of the "`gbk`" approach. When the application requests a memory block from the memory manager and the local free-list for this type of memory blocks is found empty, the memory manager can either first check the central free-list before it checks if some element is left from the last call to `malloc()`, or vice versa. The versions which do not check the central free-list first, are indicated by the suffix "`1`" in their names.

6.3. Task pools implemented in C and Java

We have implemented a number of task pools in C and in Java. The names of the task pools allow to distinguish between four basic types of task pools:

(a) central task pools with a single queue (`sq*`),

(b) distributed task pools with or without dynamic task stealing (`dq*`),

(c) combined task pools which use one central and several distributed queues (`sdq*`), and

(d) randomized task pools (`rq*`).

In most cases, there are several implementation variants of each basic type. These variants are distinguished by numbers which are added to the names of the implementations as a suffix. Tables 2 and 3 list all the task pools we have implemented.

Four variants of central task pools, `sq1` to `sq4`, have been implemented in C and one in Java (`sq1`). `sq1` and `sq2` both use LIFO queues, and both lock the entire queue for every access. They differ in the implementation of the queues as `sq1` uses a single-linked list while `sq2` uses an array of a fixed size. `sq3` and `sq4` are both FIFO queues with reduced lock granularity. Both allow simultaneous enqueue and dequeue operations. `sq3` locks single tasks, but `sq4` is based on the two-lock queue from [23] and only needs to lock head and tail of the queue.

The two distributed task pools without dynamic task stealing implemented in C are named `dq1` and `dq3`. In Java only `dq1` has been implemented. They both use LIFO queues, but `dq1` implements them by single-linked lists while `dq3` uses fixed-sized arrays.

Free Lists	Items	Memory Allocation	Block Handling	Locking	Name
—	—	elements	—	—	`bs`
central	yes	elements	—	central lists	`l`
central	yes	blocks for whole lists	elements are removed when needed	central lists	`bk`
distributed	yes	elements	—	—	`dl`
distributed	no	elements	—	—	`dlf`
distributed	yes	blocks for whole lists	elements are removed when needed	—	`dbk`
distributed	yes	blocks for whole lists	elements are removed after allocation	—	`dbko`
distributed	yes	blocks for some elements	elements are removed when needed	—	`dgbk`
distributed	yes	blocks for some elements	elements are removed after allocation	—	`dgbko`
distributed	no	blocks for some elements	elements are removed after allocation	—	`dgbkof`
central[a] + distributed	yes	elements	—	central lists	`cl`
central + distributed	no	elements	—	central lists	`clf`
central + distributed	yes	elements	—	central lists	`clb`
central[a] + distributed	yes	blocks for some elements	elements are removed when needed if central free list is empty	central lists	`cgbk`
central + distributed	yes	blocks for some elements	elements are removed when needed if central free list is empty	central lists	`cgbkb`
central[a] + distributed	yes	blocks for some elements	elements are removed after allocation	central lists	`cgbko`
central + distributed	yes	blocks for some elements	elements are removed after allocation	central lists	`cgbkob`
central[a] + distributed	yes	blocks for some elements	elements are removed when needed; blocks are allocated when central free list is empty	central lists	`cgbk1`
central + distributed	yes	blocks for some elements	elements are removed when needed; blocks are allocated when central free list is empty	central lists	`cgbk1b`

[a] no central free lists for items

Table 1. Memory managers implemented in C.

Type	Details	Queues	Access	Memory Allocation	Locking	Name
central		single-linked list	LIFO	when needed	queue	`sq1`
central		array	LIFO	during initialization	queue	`sq2`
central		double-linked list	FIFO	when needed	tasks	`sq3`
central		single-linked list	FIFO	when needed	head + tail	`sq4`
distributed		single-linked lists	LIFO	when needed	—	`dq1`
stealing	from working end	single-linked lists	LIFO	when needed	all queues	`dq2`
distributed		arrays	LIFO	during initialization	—	`dq3`
stealing	randomized local pools	single-linked lists	LIFO	when needed	all queues	`dq4`
stealing	from opposite end	double-linked list	LIFO	when needed	all queues	`dq5`
stealing	from opposite end	double-linked list	LIFO	when needed	tasks	`dq6`
stealing	from working end	arrays	LIFO	during initialization	all queues	`dq7`
stealing	public and private queues	single-linked lists, arrays	LIFO	when needed	public queues	`dq8`
central + distributed		single-linked list, arrays	LIFO	when needed	central queue	`sdq1`
randomized	rand_r, floating point	single-linked lists	LIFO	when needed	all queues	`rq1`
randomized	Mersenne Twister, integer	single-linked lists	LIFO	when needed	all queues	`rq2`
randomized	Mersenne Twister, floating point	single-linked lists	LIFO	when needed	all queues	`rq3`

Table 2. Task pools implemented in C.

Type	Details	Queues	Access	Locking	Name
central	`notify()` in every `put()`	single-linked list	LIFO	queue	`sq1`
distributed		single-linked list	LIFO	—	`dq1`
stealing	from working end; `notify()` in every `put()`	single-linked lists	LIFO	all queues	`dq2`
stealing	from opposite end; `notify()` in every `put()`	single-linked lists	LIFO	all queues	`dq5`
stealing	public and private queues; `notify()` in every `put()`	single-linked lists, arrays	LIFO	public queues	`dq8`
stealing	from working end; `notify()` only if queue is empty	single-linked lists	LIFO	all queues	`dq9`
stealing	from working end; busy-wait instead of `wait()` and `notify()`	single-linked lists	LIFO	all queues	`dq10`
central + distributed	`notify()` in every `put()`	single-linked lists, arrays	LIFO	central queue	`sdq1`
randomized	Mersenne Twister, floating point; `notify()` in every `put()`	single-linked lists	LIFO	all queues	`rq3`

Table 3. Task pools implemented in Java.

Six task pools with dynamic task stealing, `dq2` and `dq4` to `dq8`, have been implemented in C. In Java, five such task pools, `dq2`, `dq5`, and `dq8` to `dq10`, have been implemented. `dq2` is used as a reference implementation. It uses single-linked lists as LIFO queues, and task are stolen from the working end. All other task pools with dynamic task stealing differ from `dq2` in usually only one implementation detail.

- `dq4` uses an additional queue for every thread and organizes its queues as randomized local pools.
- `dq5` implements the heuristics introduced in 3.5 by stealing tasks from the end opposite to the working end.
- `dq7` implements its queues by fixed-sized arrays rather than lists.
- `dq8` uses a private and a public queue for each thread. The private queues are implemented by small, fixed-sized arrays, and the public queues are single-linked lists of such arrays.

A slight exception to this rule is `dq6`. It uses double-linked lists and reduces the lock granularity to the level of tasks to allow two concurrent dequeue operations on the two ends of a queue. Therefore it is more similar to `dq5` than to `dq2` and, for this reason, should be compared with `dq5`.

The C implementations all execute wake-up calls every time a task is inserted into a queue. This can be done because the wake-up calls to `pthread_cond_signal()` are quite fast, and thus a processor which missed a wake-up call because it was busy trying to steal a task from another processor will only sleep until the next task is inserted. Otherwise unnecessary idle time would be introduced. However, this approach is not optimal in Java, because `notify()` calls must always be enclosed into a `synchronized` block, and thus a bottleneck is created when `notify()` is executed in every `put()` operation. But to be comparable with the C implementations, the Java implementations execute the `notify()` operation in every `put()` operation by default. To investigate the impact of this bottleneck, `dq9` and `dq10` have been implemented. `dq9` executes `notify()` only when a task is inserted into an empty queue, and `dq10` eliminates the problem by using a flag combined with a busy wait instead of the `wait()`-`notify()` mechanism.

One task pool with one central and several distributed queues has been implemented in C and in Java (sdq1). Its implementation is very similar to `dq8`, since its local queues are fixed-sized arrays and the central queue is a list of arrays.

Three very similar randomized task pools, `rq1`, `rq2`, and `rq3`, have been implemented in C, but only `rq3` has been implemented in Java. All of them use four times as many central queues as there are processors. They differ only in the way of random number computation. `rq1` uses the standard library function `rand_r()` combined with floating point arithmetic to select queues. The other two rely on the Mersenne Twister [22], which was supposed to run faster than the standard library function. `rq2` uses integer arithmetic, and `rq3` uses floating point arithmetic.

7. Target machines and algorithms

To study the task pools that we have implemented, three different shared-memory multiprocessors have been used. All of them have been used to investigate the performance of the task pools for a synthetic algorithm and the hierarchical radiosity method in C with POSIX threads and in Java.

7.1. Machines

The first of the three shared-memory systems is an Linux PC with two Pentium III processors at 600 MHz. The processors have two first level caches of 16 KB for data and instructions, respectively, and a 256 KB second level cache for both data and instructions. They are interconnected by the *AGTL+ Frontside Bus*.

The operating system on this machine is Linux. All C programs have been compiled with the default optimization level under `gcc-2.95.2`. The Java programs have been executed in a Java 1.3 HotSpot Virtual Machine.

The second multiprocessor is a Sun Enterprise 420R with four UltraSPARC II processors at 450 MHz. It also has two separate 16 KB L1 caches for instructions and for data, respectively, but is equipped with a much larger L2 cache of 4 MB. The interconnection system between processors and other system components is the *Ultra Port Architecture* (UPA). Efficient access to main memory is provided by crossbar switches.

The E420R runs Solaris 8, and the C compiler is part of the Sun WorkShop 6. The default optimization level has been used as on the Linux PC. For our experiments on this machine, an implementation of Java 1.2 has been used, which included a just-in-time (JIT) compiler.

The largest machines available have been two Sun Fire with 8 (Sun Fire 3800) and 24 (Sun Fire 6800) UltraSPARC III processors at 750 MHz, which communicate by the Sun Fireplane Interconnect. Each processor has 32 KB L1 cache for instructions, 64 KB L1 cache for data and 8 MB L2 cache for both instructions and data.

The operating system of both machines is Solaris 8. As on the E420R, the C compiler from Sun WorkShop 6 and Java 1.2 have been used.

k	$P(k)$
15	5 149
20	57 290
25	635 593
30	7 049 122
31	11 405 739
32	18 454 894
33	29 860 667
34	48 315 596

Table 4. Total number of tasks created by the synthetic algorithm $P(k)$ for some values of k.

7.2. Algorithms

To investigate the different task pool variants in detail, a synthetic algorithm has been implemented which is deterministic and provides user control of its runtime behavior. It uses only one task type A that can be described by the following productions:

$$A(i) \rightarrow \begin{cases} \{100f\} & \text{for } i \leq 0, \\ \{10f\}\ A(i-2)\ \{50f\} & \\ \quad A(i-1)\ \{100f\} & \text{for } i > 0. \end{cases}$$

The values in curly braces determine simulated computation times. The unit of time is the number of iterations of a loop that is used to simulate computations. The factor f can be used to adjust the computation time of the tasks. Shared variables are not used. Though no locality in the computations can be exploited, this approach has the benefit that no synchronization operations are necessary to protect shared variables and thus performance limits that are due to synchronization are only set by the task pools themselves.

When called with argument k, in the initialization phase the algorithm sequentially creates k tasks with the arguments $k-1$ to 0. It can be shown by induction that the total number of tasks processed by the algorithm grows exponentially with k as fast as the sequel of Fibonacci numbers [18]. The total number of tasks that the synthetic algorithm creates for selected values of k is displayed in Table 4. All the results we present for the synthetic algorithm have been measured with $k = 32$ and $f = 40$. With these parameters the algorithm has to process a total of 18 454 894 tasks.

The synthetic algorithm is well suited to investigate the scalability of the task pools. It provides a deterministic task creation scheme that produces a highly unbalanced task graph. The parameters of the algorithm can be controlled by the user, which allows to investigate the influence of these parameters on the performance of the task pools. For instance, it is possible to decrease the computation time of the tasks to increase the share of time needed to execute the task pool operations. This allows to uncover bottlenecks in the task pool operations, which could not be observed if the computation time of the tasks is orders of magnitudes higher than that of the task pool operations. Furthermore, since the absolute difference of the execution times of the task pool operations can be very small, increasing the number of tasks executed allows to measure a multiple of this difference as the difference of the overall execution times necessary to execute all tasks.

To see how the task pools implemented perform with a realistic application, the *radiosity* application from the SPLASH-2 benchmark suite [38] was chosen to be investigated. The SPLASH-2 suite was developed to provide a set of shared-memory applications which facilitate the evaluation of different hardware architectures.

The *radiosity* application is an implementation of the hierarchical radiosity method [13, 30]. This global illumination algorithm computes radiosity values for a given geometric scene by an adaptive hierarchical subdivision of the object surfaces. For each of the resulting surface elements, a radiosity value is computed by taking all interacting surface elements of other object surfaces into consideration.

As a C implementation of this method, the *radiosity* application from the SPLASH-2 suite could be used directly by applying only minor modifications. To investigate the task pools written in Java, the application had to be ported.

The *radiosity* application can be considered as a nondeterministic algorithm, because different execution orders of the tasks can be observed to produce a marginally different number of surface elements and, as a consequence, slightly different numbers of interactions are processed.

Since the task pools have been designed as universal data structures that can be used with any task-based algorithm, they cannot make assumptions about the application which exceed the model of task-based algorithms. Therefore different task pools can use different execution orders and induce different execution times of the *radiosity* application. Furthermore, in parallel runs, the order of task execution is affected by the concurrent behavior of the processors.

A first approach to minimize the effects of nondeterminism is averaging the execution times of multiple runs. Another approach is to use statistical information combined with the execution time to assess a particular run of the algorithm. For example, the total number of interactions processed per second or the number of visible interactions processed per second allow a good comparison. This approach has the advantage that it measures the performance of the task pools as the ratio of work per unit time. Thus it is nearly independent of the execution order used by specific task pools.

Three input scenes, “test”, “room” and “largeroom”, are included in the SPLASH-2 application. For our experiments

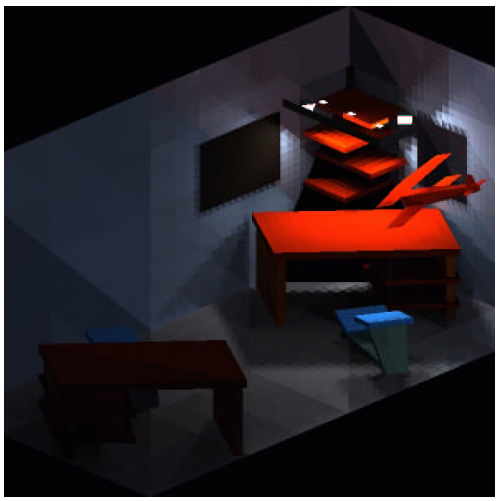

Figure 4. Szene “largeroom”.

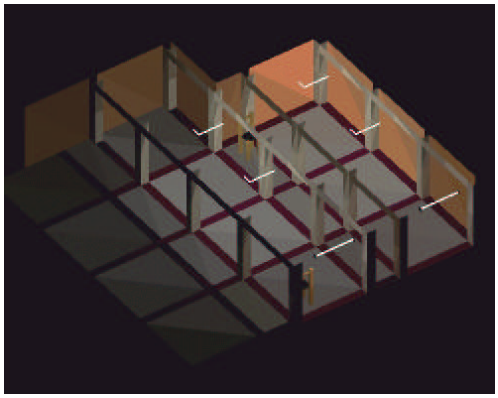

Figure 5. Szene “Halle”.

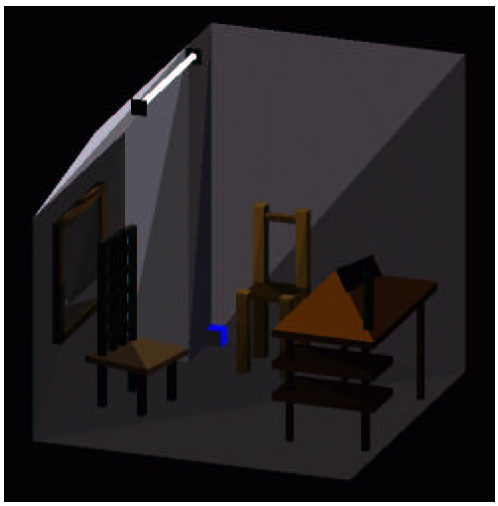

Figure 6. Szene “Raum11w”.

the largest input scene, “largeroom” (Figure 4), was chosen to be investigated. After the sequential creation of the BSP tree, this scene consists of 532 patches, and 5 to 8 iterations are performed to compute about 240 000 interactions. This number may vary by 0.9 %. The number of surface elements created for this scene is 21 000 and may vary by 1.4 %. The average number of visible interaction is 89 000 and varies by maximum 1.2 %.

Since in our first experiments on the smaller machine the results of the task pools for the *radiosity* application showed only minor differences, on the larger machines we have also measured results for two more-detailed scenes, “Halle” and “Raum11w”. For this purpose, the extensions implemented in [26] have been used to enable the *radiosity* application to read the input scene from a file. Scene “Halle” (Figure 5) consists of 1 157 patches, and about 440 000 interactions, 184 000 of which are visible, are processed in usually 4 iterations. The scene “Raum11w” (Figure 6) is even larger. It contains 2 979 patches, which lead to about 553 000 visible interactions from a total of 1 700 000 that are processed in about 4 iterations. For both scenes, the number of surface elements and the total number of interactions do not change if the program execution is repeated. But the number of visible interactions still varies by maximum 1.2 %.

8. Results

8.1. C versions of the synthetic algorithm

To assess the task pools implemented in C, the execution times for all memory managers have been measured. Then, the minimum execution time has been used as basis for the assessment, arguing that this value is the best execution time that can be achieved with a specific task pool variant provided that the most suitable memory manager is used.

In order to investigate the ability of the task pools to exploit locality, we have used the PCL library [3] to measure the misses of both L1 and L2 caches.

Linux PC

The minimum execution times of the C versions of the synthetic algorithm measured on the Linux PC are shown in Table 5. Table 7 shows the corresponding speed-ups. The cache misses measured are displayed in Table 6.

As expected, the two distributed task pools, `dq1` and `dq3`, do best if only one thread is used, because all their queues are private and they therefore do not need to execute any synchronization operations to protect the queues. When run in parallel, they usually take more time than all other task pools, since the initialization phase of the synthetic algorithm was designed to create an unbalanced data distribution and no transfer of tasks is taking place.

Algorithm	Test, $k = 32$, $f = 40$				Radiosity, "largeroom"			
Language	C		Java		C		Java	
Threads	1	2	1	2	1	2	1	2
sq1	0.96	1.76	0.92	1.41	1.00	1.92	0.99	1.14
sq2	0.96	1.79	–	–	1.00	1.91	–	–
sq3	0.90	1.63	–	–	0.99	1.90	–	–
sq4	0.95	1.73	–	–	0.99	1.84	–	–
dq1	1.00	1.59	0.98	1.54	1.00	1.78	0.99	1.14
dq2	0.95	1.85	0.96	1.67	1.00	1.95	1.00	1.14
dq3	1.00	1.60	–	–	1.00	1.76	–	–
dq4	0.95	1.82	–	–	1.00	1.95	–	–
dq5	0.95	1.86	0.89	1.71	1.00	1.95	0.99	1.19
dq6	0.91	1.77	–	–	0.99	1.95	–	–
dq7	0.95	1.86	–	–	1.00	1.94	–	–
dq8	1.00	1.96	0.93	1.82	1.00	1.78	1.00	1.18
dq9	–	–	0.92	1.75	–	–	1.00	1.19
dq10	–	–	0.95	1.83	–	–	1.00	1.18
sdq1	1.00	1.96	1.00	1.90	1.00	1.95	0.99	1.19
rq1	0.94	1.74	–	–	0.99	1.94	–	–
rq2	0.93	1.74	–	–	0.99	1.93	–	–
rq3	0.93	1.73	0.85	1.41	0.99	1.93	0.98	1.14

Table 7. Speed-ups on the Linux PC.

	Number of Threads			
	1		2	
1	dq3	314.9 s	dq8	160.7 s
2	dq1	+0.2 %	sdq1	+0.0 %
3	sdq1	+0.4 %	dq7	+5.4 %
4	dq8	+0.4 %	dq5	+5.6 %
5	sq2	+4.6 %	dq2	+5.7 %
6	sq1	+4.7 %	dq4	+8.0 %
7	dq2	+4.9 %	sq2	+9.7 %
8	dq7	+5.1 %	dq6	+10.9 %
9	dq5	+5.1 %	sq1	+11.3 %
10	dq4	+5.5 %	rq2	+12.3 %
11	sq4	+5.6 %	rq1	+12.4 %
12	rq1	+6.8 %	rq3	+12.9 %
13	rq3	+7.0 %	sq4	+13.1 %
14	rq2	+7.3 %	sq3	+20.1 %
15	dq6	+10.4 %	dq3	+22.4 %
16	sq3	+11.2 %	dq1	+23.3 %

Table 5. Comparison of the minimum execution times of the C versions of the synthetic algorithm on the Linux PC.

Cache	L1		L2	
Threads	1	2	1	2
sq1	11	148327	3	77750
sq2	8	121847	2	66222
sq3	46843	174486	38712	95179
sq4	23881	207935	19210	107188
dq1	10	45598	3	24150
dq2	14	46648	3	25340
dq3	8	39404	2	20353
dq4	1527	103861	45	52361
dq5	16	45155	3	24189
dq6	19	46240	4	24791
dq7	10	38158	3	19815
dq8	12	27896	3	15176
sdq1	12	32246	3	17927
rq1	2871	146912	90	77257
rq2	3886	164280	97	82909
rq3	3954	173149	92	86549

Table 6. Average number of cache misses (in thousands) of the C versions of the synthetic algorithm on the Linux PC.

To compare dq1 and dq3, the array implementation of the queues in dq3 seems to be slightly faster than the list implementation of the queues in dq1. This can be explained by the lower complexity and the higher locality of the array implementation. Since the data of the queues is not shared, false sharing can only occur if threads are assigned to different processors during their lifetime.

The best parallel results are achieved by the two task pools dq8 and sdq1. dq8 implements dynamic task stealing with private and public queues, while sdq1 is a combined central and distributed task pool. They both have in common that synchronization operations do not have to be executed in every task pool operation. This reduces synchronization overhead as well as the number of simultaneous accesses to queues. Additionally, since tasks are organized in small blocks of four tasks, they show very good locality. But this block organization also has the drawback to coarsen the available parallelism, since now the minimum unit of parallelism is a block of four tasks.

The central queue of sdq1 does not impose a bottleneck since the number of processors is extremely small. Because of the reduced synchronization overhead, dq8 and sdq1 also obtain the best sequential execution times of all task pools that allow the balancing of tasks. As a result, the best speed-up for the C versions of the synthetic algorithm on the Linux PC of 1.96 has been measured with dq8 and two threads. Because their results are nearly identical, it is not possible to compare dq8 and sdq1 on this machine.

Central and randomized task pools obtain about the same performance. Even though sq3 and sq4 allow two simultaneous queue operations they cannot outperform sq1 because of their higher complexity and worse locality (see Table 6). As it was the case for the two distributed task pools, the array implementation of the central queue of sq2 is faster than the list implementation of sq1. Due to their very similar implementations, the execution times of the three randomized task pools show hardly any differences.

In general, best performance is provided by dynamic task stealing. Since those task pools do not use central data structures, the task pool operations are not sequenced. Only synchronization overhead and waiting times in the stealing process slow them down. Therefore dq8, which uses private and public queues, is the fastest task pool with dynamic task stealing, because it only needs to synchronize the public queues.

To compare the other task-stealing pools, it can be seen that implementing queues by arrays, like it was done for dq7, achieves better performance than the list implementation of dq2. The stealing heuristics used for dq5 also outperforms the standard implementation dq2.

Randomized local pools have been implemented as dq4. They decrease the performance due to the higher complexity of the data structure and reduced locality. Similarly, reducing the lock granularity to tasks increases the synchronization overhead so much that the results of dq6 are the worst of all task pools with dynamic task stealing.

	Number of Threads			
	1		2	
1	dlf	328.9 s	dbko	172.1 s
2	dgbkof	+0.1 %	dbk	+0.1 %
3	dl	+0.3 %	dgbko	+1.0 %
4	clf	+0.3 %	cgbk1	+1.2 %
5	dbko	+0.4 %	dgbkof	+1.5 %
6	cl	+0.5 %	cgbk1b	+1.6 %
7	cgbko	+0.5 %	cgbk	+1.7 %
8	dgbk	+0.5 %	dlf	+2.1 %
9	dgbko	+0.5 %	cgbkb	+2.1 %
10	cgbk	+0.6 %	dgbk	+2.3 %
11	cgbk1	+0.6 %	cgbko	+2.4 %
12	dbk	+0.6 %	cl	+2.7 %
13	clb	+1.2 %	cgbkob	+2.8 %
14	cgbkob	+1.2 %	clf	+3.2 %
15	cgbk1b	+1.3 %	dl	+3.3 %
16	cgbkb	+1.3 %	clb	+4.6 %
17	bs	+8.1 %	bs	+15.9 %
18	bk	+10.2 %	bk	+27.6 %
19	l	+10.3 %	l	+29.8 %

Table 8. Comparison of the averaged execution times of the memory managers for the synthetic algorithm on the Linux PC.

Using a suitable memory manager has significant impact on the execution time of a task based algorithm. On the Linux PC the best memory manager, dgbko, achieves an average speed-up of about 16 % compared to the memory manager of the operating system (bs) when the synthetic algorithm runs with two threads. A comparison of the memory managers on the Linux PC is shown in Table 8.

As expected, on the Linux PC the central free-lists of l and bk lead to even worse execution times than those of bs. The results of the other memory managers, which use distributed free-lists and sometimes also central free-lists in addition, are – in individual cases – 10 to 20 % better than those of bs.

In general, the results of the memory managers imply that it is not necessary to implement additional central free-lists for balancing. One should rather choose a simpler and faster implementation of distributed free-lists only, because the central free-lists will introduce a bottleneck if the number of processors increases, and – even without central free-lists – memory blocks are transferred between processors when there is a transfer of tasks. Allocating memory for multiple tasks at once seems to be a good idea.

On the Linux PC the best memory managers also show

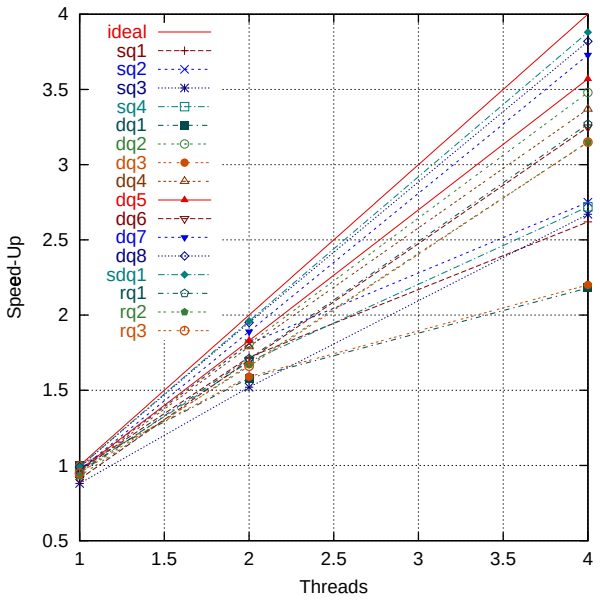

Figure 7. Speed-ups of the C versions of the synthetic algorithm on the Sun E420R.

Cache	L1		L2	
Threads	1	4	1	4
sq1	118400	30259	10	7
sq2	98907	24767	5	2
sq3	356917	169633	22397	5626
sq4	163206	40422	11663	3013
dq1	39231	10028	9	3
dq2	119473	28960	5	5
dq3	19920	5198	5	1
dq4	140185	35731	80	25
dq5	116670	28873	11	77
dq6	275189	72251	17	20
dq7	146148	26822	11	1
dq8	23029	6452	13	2
sdq1	23027	6658	11	30
rq1	133225	35147	31	25
rq2	143140	38634	87	88
rq3	152281	40418	94	34

Table 9. Average number of cache misses (in thousands) of the C versions of the synthetic algorithm on the Sun E420R.

the least cache misses on both L1 and L2 caches. So locality seems to be the reason that some memory managers that use items are even faster than the corresponding memory managers that do not.

Sun Enterprise 420R

On this machine a similar order of the execution times of the different types of task pools as on the Linux PC can be observed. Dynamic task stealing and combined central and distributed task pools provide the best results. Using four threads, the randomized task pools outperform the central task pools. The distributed task pools achieve the worst results.

As on the Linux PC, the distributed task pools, `dq1` and `dq3`, achieve the best sequential results, but are very slow when run in parallel. Their execution times improve when the number of threads slightly exceeds the number of processors. In this case the operating system can utilize an idle processor by assigning one of the additional threads to it. If the number of threads is increased even further, the overhead for thread scheduling will be more expensive than performing synchronization operations on the distributed queues. Again `dq3` is slightly faster than `dq1`. The speed-ups for `dq1` and `dq3` measured with four threads are 2.18 and 2.20, respectively.

The best parallel results are achieved by `dq8` and `sdq1` like it was the case on the Linux PC. But on this machine our results show minor advantages of `sdq1`, which reaches a speed-up of 3.88 with four threads. `dq8` only obtains 3.82. The number of processors is small enough that the central queue of `sdq1` does not become a bottleneck.

Dynamic task stealing provides the best scalability on this machine as well, and the results measured for these task pools imply the same order as on the Linux PC.

Compared to `dq2`, the stealing heuristics improves the performance of `dq5`. But this is not the case for `dq6`, because the costs to reduce the lock granularity are very high. `dq4` with randomized local pools also cannot match the performance of `dq2`. The array implementation of `dq7` speeds up even better than the heuristics of `dq5`.

All central task pools do not scale very well. If only two processors are used, the best central task pools can still compete with the worst task pools with dynamic task stealing. But if the number of processors is increased, the bottleneck of the central queue limits the performance.

Reducing the lock granularity of the central queue increases its complexity. Therefore the sequential execution times of `sq3` and `sq4` are worse than those of the comparable task pool `sq1`. Another reason for the large execution times of both `sq3` and `sq4` is that they process the tasks in FIFO order, which leads to a high number of cache misses (see Table 9). Yet, if the number of processors is increased, they do better than `sq1`. With four threads, the speed-ups measured for `sq3` and `sq4` are 2.67 and 2.72, respectively, while `sq1` only reaches 2.62.

Since the complexity of `sq4`, which is based on the two-lock queue from [23], is much lower than that of `sq3` which

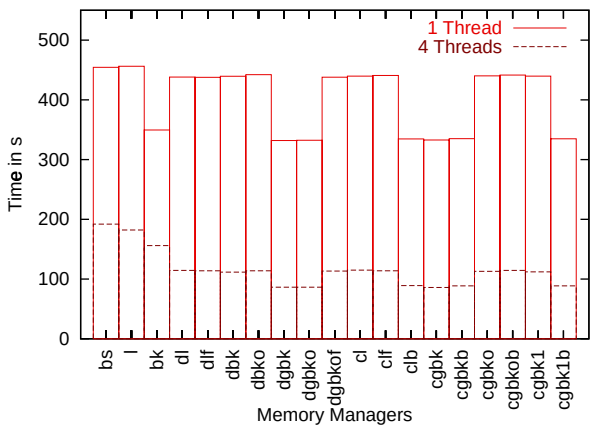

Figure 8. Runtimes of the C memory managers with `dq8` for the synthetic algorithm on the Sun E420R.

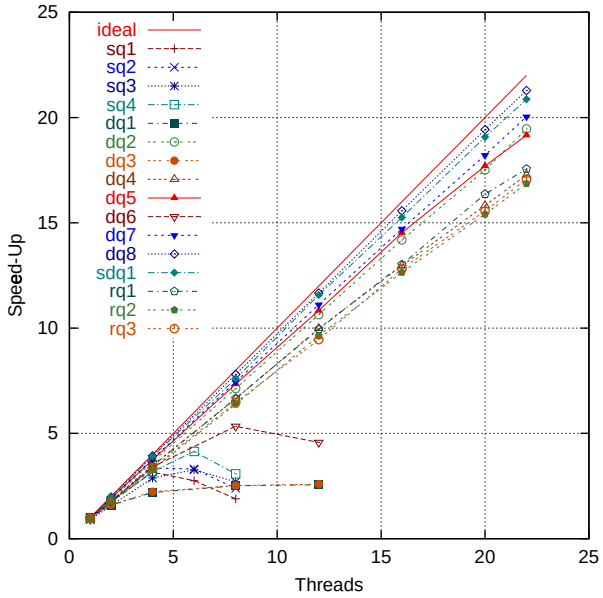

Figure 9. Speed-ups of the C versions of the synthetic algorithm on the Sun Fire.

locks single tasks, `sq4` runs faster. As for the two distributed task pools, the array implementation of the central queue of `sq2` is faster than the list implementation of `sq1`. It obtains a speed-up of 2.75.

The three randomized task pools, `rq1`, `rq2` and `rq3`, have bad sequential execution times. This is due to the high costs of random number computation and the reduced locality. However, when run on multiple processors, the randomized task pools scale better than the central task pools, but cannot compete with dynamic task stealing. The speed-ups they achieve range from 3.15 to 3.27.

Since the three randomized task pools differ only in the way of random number computation, no advantage of one version over the other can be measured. Only the standard library function `rand_r()` used for `rq1` seems to do slightly better than the Mersenne Twister [22] used for `rq2` and `rq3`.

The results of the memory managers (Table 10) confirm the significance of their use. When run with four threads, on this machine some of our memory mangers are even 90 % faster than the memory manager that capsulates `malloc()` and `free()`, `bs`.

Comparing the memory managers on the Sun E420R is difficult. Investigating a selected task pool, the execution times for the different memory managers can often be classified into several groups of similar execution times. The difference of the execution times of two different groups is then quite large. For different task pools the number of groups and the affiliation of the memory managers to the groups varies.

Figure 8 shows the runtimes of the memory managers for `dq8` as an example. It can be seen that there are two groups of memory managers in the case of one thread. The memory managers in the first group show execution times of about 440 s, but the memory managers in the second group need only about 330 s. These large differences in the execution times of these groups cannot be explained by the different implementations of the memory managers since for other task pools different groups can be found. Additionally, considering only the `d*` and `c*` memory managers, the results for the Linux PC show that the impact of the different memory manager implementations is only very small compared to the ratio of $\frac{3}{4}$ that could be measured in the example of `dq8` on the Sun E420R.

Since the effects also occur in the single-threaded runs, the concurrent behavior of the threads cannot be the reason. Moreover, in contrast to the Linux PC, no clear correlation between locality and the execution time of a memory manager can be observed. This implies that the discovered grouping effect is a result of the complex relationship between the locality of memory references and implementation details of both the memory managers and the specific task pool variants as well as the characteristics of the hardware architecture of the machine.

Sun Fire 3800 and 6800

Due to the larger numbers of processors, smaller differences in the scalability of the task pools become visible on these machines. Figure 9 shows the results for the C versions of the synthetic algorithm measured with up to 22 threads. It can be seen that the scalability of the central task

	Number of Threads							
	1		2		4		8	
1	dgbkof	375.8 s	dbk	207.8 s	dbk	120.9 s	dbk	119.7 s
2	cl	+2.9 %	dgbkof	+0.8 %	dgbkof	+0.7 %	dgbkof	+1.3 %
3	dbk	+3.4 %	dbko	+2.7 %	dbko	+3.1 %	dbko	+5.1 %
4	clb	+3.8 %	dgbk	+4.1 %	dgbk	+3.8 %	dgbk	+5.7 %
5	dbko	+4.0 %	cl	+4.8 %	cgbk	+4.1 %	cgbk1b	+5.5 %
6	dgbk	+4.1 %	clf	+4.8 %	cgbkb	+6.2 %	cgbkb	+5.7 %
7	clf	+4.2 %	cgbk1b	+5.5 %	cgbk1b	+6.4 %	clf	+5.8 %
8	dl	+4.3 %	cgbkb	+5.7 %	clf	+6.7 %	cgbk	+7.2 %
9	cgbk1b	+5.1 %	dl	+6.0 %	cl	+8.2 %	cl	+9.4 %
10	cgbkb	+5.2 %	cgbk	+6.1 %	cgbk1	+8.6 %	cgbko	+10.2 %
11	cgbko	+6.2 %	clb	+6.5 %	dl	+9.0 %	dl	+10.5 %
12	cgbk	+6.2 %	cgbk1	+6.6 %	cgbko	+9.4 %	cgbk1	+10.9 %
13	cgbk1	+6.2 %	cgbko	+7.2 %	dgbko	+9.8 %	clb	+10.9 %
14	dlf	+7.2 %	dlf	+8.4 %	clb	+10.2 %	dlf	+12.0 %
15	bk	+12.3 %	dgbko	+12.1 %	dlf	+10.9 %	cgbkob	+12.0 %
16	dgbko	+13.0 %	cgbkob	+14.6 %	cgbkob	+13.2 %	dgbko	+12.6 %
17	bs	+13.0 %	bk	+27.1 %	bk	+91.4 %	bk	+151.8 %
18	l	+13.4 %	bs	+28.8 %	bs	+92.9 %	l	+181.4 %
19	cgbkob	+14.0 %	l	+27.9 %	l	+94.0 %	bs	+210.4 %

Table 10. Comparison of the averaged execution times of the memory managers for the synthetic algorithm on the Sun E420R.

pools is very poor. The maximum speed-up reached with a central task pool is 4.14. It was measured with `sq4` running with six threads.

The distributed task pools also restrict the achievable speed-up. They both reach speed-ups of 2.56 when 12 threads are used. Speed-ups for larger numbers of threads have not been measured, because they were expected to improve only insignificantly. Another task pool that limits the scalability of the synthetic algorithm is `dq6`. It reaches a speed-up of 5.34 with 8 threads.

All other task pools provide nearly linearly increasing speed-ups. The best speed-up of 21.29 has been measured with `dq8`. `sdq1` reaches a speed-up of 20.87. The maximum speed-up achieved with the reference implementation of dynamic task stealing, `dq2`, is 19.46. `dq7`, which implements its queues by arrays, can outperform this result. Due to the heuristics used, in most cases `dq5` is slightly faster than `dq2`. Only in the run with 22 threads its results are worse than those of `dq2`. The randomized task pools and `dq4` reach speed-ups of about 16.8 to 17.6. The best task pool of this group is `rq1`.

8.2. Java versions of the synthetic algorithm

Since no memory managers have been implemented in Java, only one version of each task pool exists, and the task pools can be compared directly by their execution times.

Linux PC

Table 11 compares the results of the Java versions of the synthetic algorithm on the Linux PC. The speedups measured are shown in Table 7.

In the run with a single thread, `sdq1` is the fastest task pool. The other task pools are between 2.2 % (`dq2`) and 18.1 % (`rq3`) slower. If two threads are used, `sdq1` obtains the best speed-up of 1.90. Then `dq10`, `dq8` and `dq9` follow. All of these four task pools execute fewer synchronization operations than the other task pools, except for `dq1`, the slow execution time of which results from the static data distribution it uses.

The heuristics used in `dq5` attains better results than the reference implementation of `dq2`. Because of the expensive random number computations, the randomized task pool, `rq3`, is even slower than the distributed task pool, `dq1`. The central task pool, `sq1`, reaches the worst results. This is due to the contention for the central queue.

A difference between the mechanisms for conditional waiting in POSIX threads and Java is that in Java the operation to wake up sleeping threads must only be called when the corresponding lock has been acquired. The results for `dq9` and `dq10` show that the performance of the task pools improves if the number of such calls is reduced, because thus the sequencing of calls to `put()` can be avoided.

While in C synchronization behavior, the number of ex-

	Number of Threads					
	1		2		4	
1	sdq1	335.9 s	sdq1	176.7 s	sdq1	178.4 s
2	dq1	+2.2 %	dq10	+4.0 %	dq8	+3.4 %
3	dq2	+3.7 %	dq8	+4.4 %	dq10	+7.1 %
4	dq10	+4.9 %	dq9	+8.6 %	dq9	+9.1 %
5	dq8	+7.7 %	dq5	+10.9 %	dq1	+11.3 %
6	dq9	+8.8 %	dq2	+13.6 %	dq5	+34.9 %
7	sq1	+9.0 %	dq1	+23.3 %	dq2	+37.1 %
8	dq5	+12.2 %	rq3	+34.5 %	rq3	+44.8 %
9	rq3	+18.1 %	sq1	+34.7 %	sq1	+94.4 %

Table 11. Comparison of the execution times of the Java versions of the synthetic algorithm on the Linux PC.

ecuted operations and locality are the main factors that influence the execution time of an algorithm, the execution time of a Java program also depends on the abilities of the virtual machine to speed up the interpretation of the intermediate code. Certainly different programs are different to access for optimization techniques. And current virtual machines reduce the execution time of a program remarkably. For example, with Java 1.3, the best task pool executed on the Linux PC with one thread lags only about 7 % behind the best C implementation. When a version of Java 1.2 without a just-in-time compiler was used, the Java programs were about 40 times slower than the C programs.

Sun Enterprise 420R

The execution times of the Java versions of the synthetic algorithm on the Sun E420R are shown in Table 12. The speed-ups achieved are illustrated in Figure 10.

On this machine the results measured with Java are very similar to the C versions. The only distributed Java task pool, `dq1`, gives the best result for one thread, but suffers from load imbalance when run with multiple threads and only attains a speed-up of 2.15.

As in C, `sdq1` and `dq8` have the best parallel performance. With four threads they reach speed-ups of 3.64 and 3.54 respectively.

`dq10` and `dq9` only reach speed-ups of 3.03 and 2.96, respectively. Compared to `dq8`, they have to execute more synchronization operations, because they have to acquire a lock every time they access their local queue.

`dq5` and `dq2` achieve speed-ups of 2.57 and 2.54, respectively. They acquire an additional lock to execute a wake-up call every time a new task is inserted into the pool.

Because the computation of the random numbers is expensive, the randomized task pool, `rq3`, and the distributed task pool, `dq1`, reach about the same speed-ups of 2.11 and 2.15, respectively.

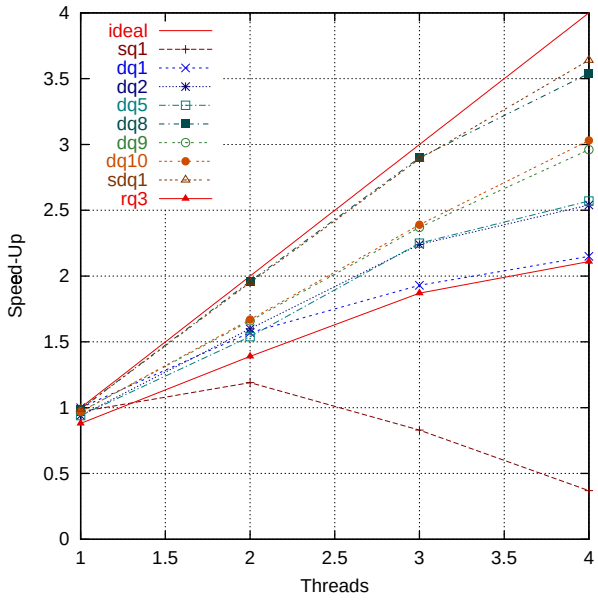

Figure 10. Speed-ups of the Java versions of the synthetic algorithm on the Sun E420R.

	Number of Threads							
	1		2		4		8	
1	dq1	711.5 s	dq8	358.4 s	sdq1	189.6 s	dq8	182.7 s
2	dq8	+0.1 %	sdq1	+0.3 %	dq8	+0.3 %	sdq1	+2.0 %
3	sdq1	+0.4 %	dq9	+6.2 %	dq9	+8.1 %	dq10	+12.3 %
4	sq1	+1.1 %	dq10	+8.6 %	dq10	+9.0 %	dq9	+12.5 %
5	dq10	+1.4 %	dq2	+8.7 %	dq2	+11.9 %	dq1	+68.0 %
6	dq9	+1.9 %	sq1	+9.0 %	dq5	+14.5 %	dq2	+100.2 %
7	dq5	+2.8 %	dq5	+11.7 %	rq3	+25.2 %	dq5	+111.1 %
8	dq2	+4.5 %	rq3	+18.2 %	dq1	+71.5 %	rq3	+121.6 %
9	rq3	+6.4 %	dq1	+24.2 %	sq1	+303.8 %	sq1	+405.2 %

Table 12. Comparison of the execution times of the Java versions of the synthetic algorithm on the Sun E420R.

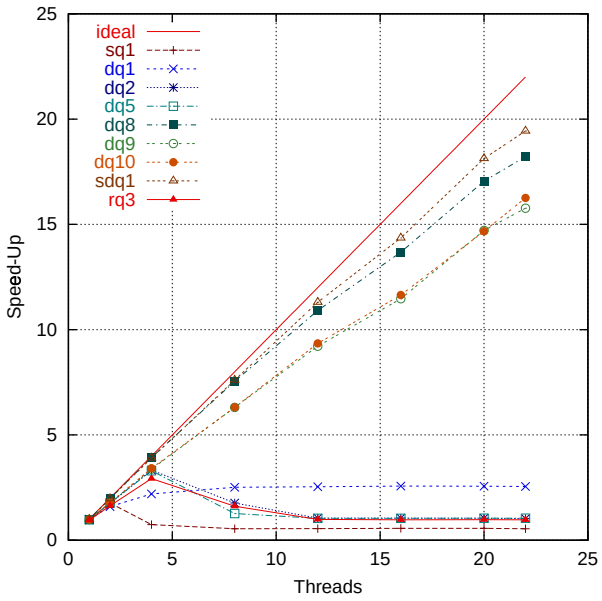

Figure 11. Speed-ups of the Java versions of the synthetic algorithm on the Sun Fire.

Due to the contention for the central queue the central task pool, `sq1`, reaches its maximum speed-up of 1.19 with two threads. When executed with more threads, the performance of `sq1` even decreases.

Sun Fire 3800 and 6800

Figure 11 shows the speed-ups for the Java versions of the synthetic algorithm for up to 22 threads. Here the best task pool is `sdq1`. Its speed-up increases up to 19.41 measured with 22 threads. `dq8` obtains the second best speed-up of 18.87.

`dq9` and `dq10` have about equal speed-ups. Because they both do not execute wake-up calls every time a new task is inserted, their speed-up curves are nearly linear and reach a maximum speed-up of 16.07 and 16.59 with 22 threads, respectively.

`dq2`, `dq5` and `rq3` all suffer from the bottleneck created by the wake-up call. So they reach their maximum speed-up of 2.93 to 3.33 with only four threads. The central task pool, `sq1`, performs even worse. Its maximum speed-up of 1.77 was measured with only two threads. The distributed task pool, `dq1`, reaches its maximum speed-up of 2.57 with about 16 threads.

8.3. C versions of the *radiosity* application

The **C versions of the *radiosity* application** are more difficult to evaluate than those of the synthetic algorithm because the execution times of different program runs may vary due to nondeterministic behavior. We therefore decided to take statistical information about the computations into account. More precisely, we used the sum of completely and partially visible interactions divided by the execution time as the assessment basis. In particular, we use the average over the interaction rates of all memory managers to compare the task pools.

Linux PC

The results for the *radiosity* application show a similar order of the execution times of the different types of task pools as the results for the synthetic algorithm (Table 7). But it is hardly possible to compare different task pool versions of the same type, because the results are very close to each other. The only remarkable exception is `dq8`, the results of which amount to those of the two distributed task pools.

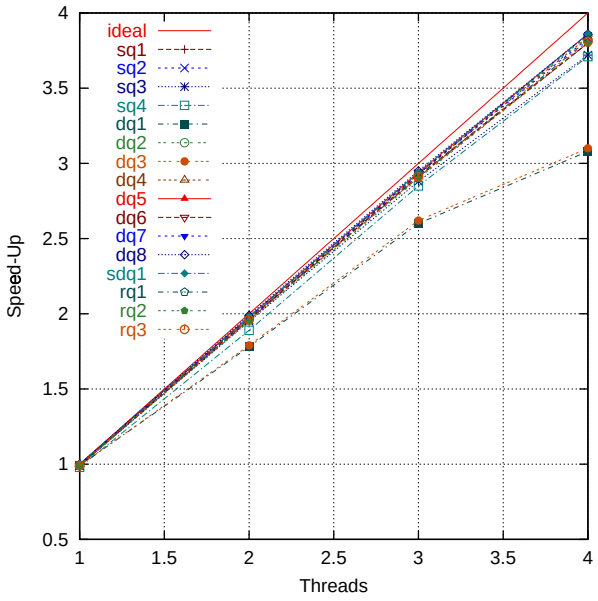

Figure 12. Speed-ups of the C versions of the radiosity application (scene “largeroom”) on the Sun E420R.

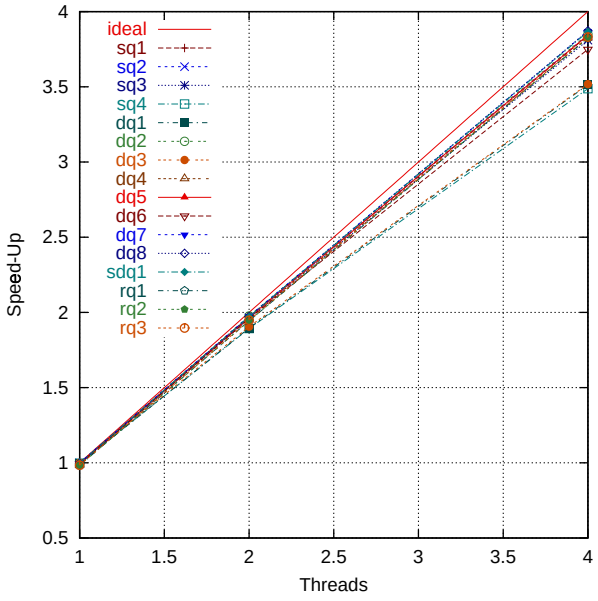

Figure 13. Speed-ups of the C versions of the radiosity application (scene “Halle”) on the Sun E420R.

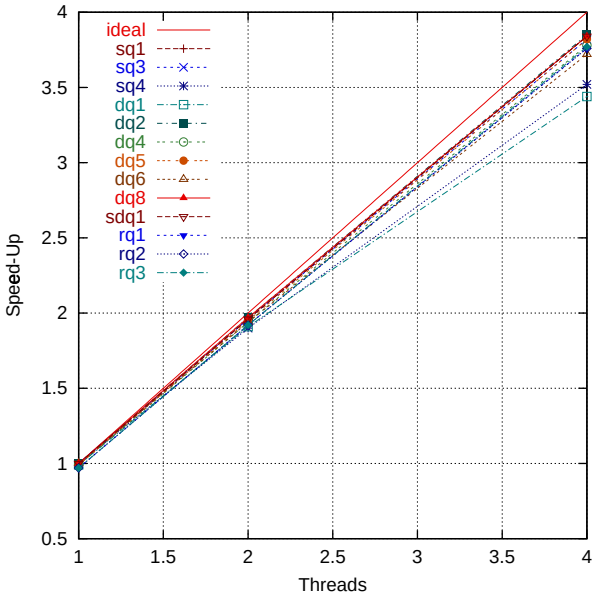

Figure 14. Speed-ups of the C versions of the radiosity application (scene “Raum11w”) on the Sun E420R.

Sun Enterprise 420R

The results for the C versions of the *radiosity* application on the Sun E420R are shown in Figure 12. The best speed-up measured here (3.86) is about equal to the best speed-up of the synthetic algorithm. But for the *radiosity* application all task pools show good scalability on this machine. The reason for this behavior is that the *radiosity* application executes a smaller number of tasks which are computationally more intensive.

The worst speed-ups, which have been measured for the two distributed task pools, are 3.10 and 3.08, respectively. `sq3` and `sq4` respectively reach speed-ups of 3.72 and 3.71. These two task pools execute the tasks in FIFO order. All other task pools reach a speed-up of at least 3.80. Because of the close results, it is hardly possible to compare the task pools within this group.

The speed-up curves we have measured for the scenes “Halle” and “Raum11w” are very similar to those of “largeroom”, so they also do not allow a more detailed comparison of the task pools (Figures 13 and 14).

Sun Fire 3800 and 6800

Because of the higher number of processors, on these machines those memory managers that were known to strongly limit the performance of the task pools were not included in the average interaction rate.

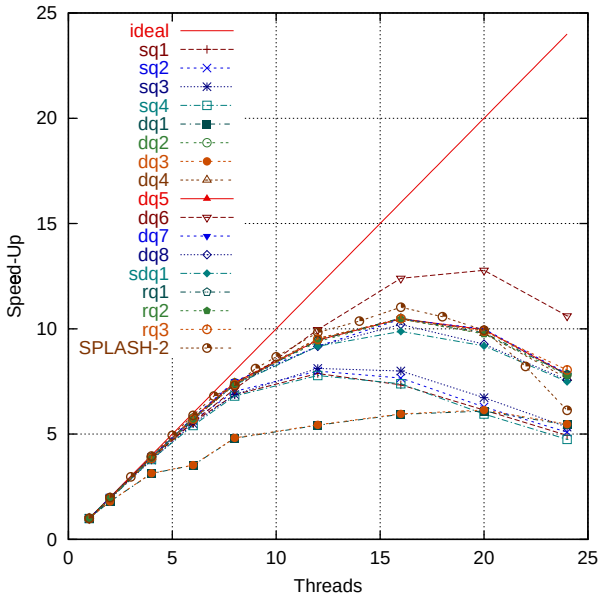

Figure 15. Speed-ups of the C versions of the radiosity application (scene “largeroom”) on the Sun Fire.

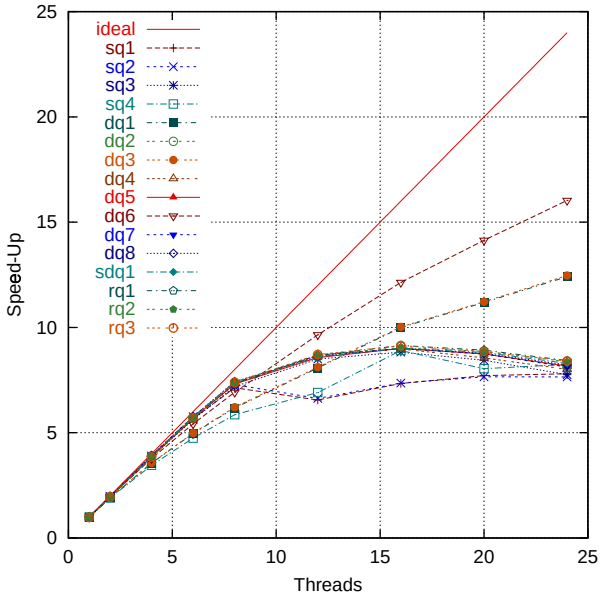

Figure 16. Speed-ups of the C versions of the radiosity application (scene “Halle”) on the Sun Fire.

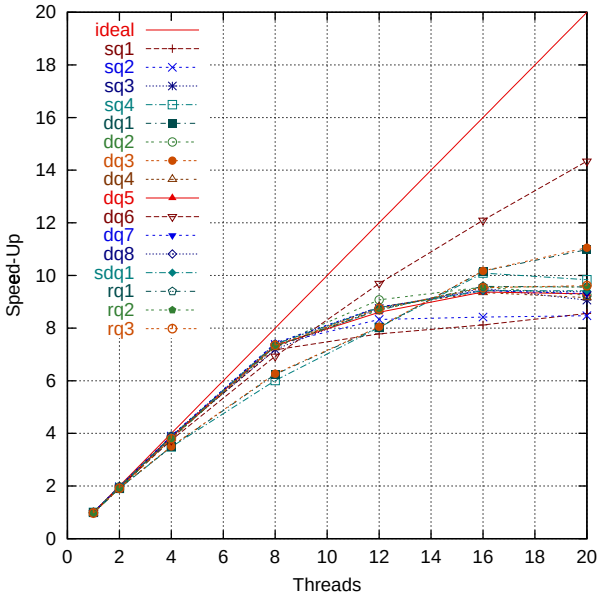

Figure 17. Speed-ups of the C versions of the radiosity application (scene “Raum11w”) on the Sun Fire.

All speed-up curves for the scene “largeroom” reach their maximum with less than 24 processors (Figure 15). The speed-ups of the distributed task pools slowly increase to 6.1 measured with 20 threads. The central task pools reach their maximum speed-ups of 7.8 to 8.1 with 12 threads. The best speed-up of 12.40 has been measured with `dq6` and 20 threads.

The other task pools reach their maximum speed-ups of 9.9 to 10.5 with 16 threads. Up to 20 threads the speed-ups of the SPLASH-2 implementation are slightly better than the speed-ups of the task pools in this group. The reason for this is that our task pools have been assessed by the average interaction rate of several memory managers. If only the best memory managers had been considered, some task pools of this group would have shown better results than the SPLASH-2 implementation. This is particularly surprising since our implementations are not optimized to exploit the special properties of the *radiosity* application.

For scene “Halle” (Figure 16) again `dq6` obtains the best results. Its speed-up increases up to 16.03 when 24 threads are used. In contrast to the results mentioned above, the speed-ups of the distributed task pools, `dq1` and `dq3`, can also profit from all 24 processors. They reach speed-ups of 12.41 and 12.47, respectively.

All other task pools reach their maximum speed-ups of about 9.0 with 16 threads. The only exceptions are `sq1`, `sq2` and `sq4`. This is caused by the non-linear speed-ups of some combinations of task pools and memory managers

which influence the average interaction rates displayed. If only the minimum execution times of all memory managers were displayed, the results of `sq1` and `sq2` would be about equal to the other task pools of this group. `sq4` would even reach better speed-ups than the distributed task pools.

For our measurements with the scene "Raum11w" only 20 processors could be used (Figure 17). The results for this scene are similar to those for scene "Halle". `dq6` achieves the best speed-up of 14.34 with 20 threads. The distributed task pools, `dq1` and `dq3`, achieve speed-ups of 10.98 and 11.06, respectively.

The other task pools also reach their maximum speed-ups with 16 threads, but a decrease of the ascents of their speed-up curves can be observed that implies that they would hardly profit from additional processors.

8.4. Java versions of the *radiosity* application

Linux PC

The results for the Java versions of the *radiosity* application on this machine (Table 7) divide the task pools into two groups. The group with the better interaction rates consists of `dq5`, `dq9`, `sdq1`, `dq8` and `dq10`. Their results only vary by 1 %. The results of the other group, which consists of `dq1`, `dq2`, `sq1` and `rq3`, are about 4 % worse than those of the best task pool, `dq5`. The differences of the interaction rates in this group are even smaller than in the first group.

Except for `dq5` and `dq1`, the task pools in the first group execute fewer synchronization operations than the members of the second group. `dq1` is slower because of the static data distribution it uses. The reason why `dq5` does best probably lies in the stealing heuristics.

Sun Enterprise 420R

To compare the Java versions of the *radiosity* application is even more difficult than to compare the C versions, because the speed-ups reached only range from 2.39 to 2.57 in Java. This is especially the case since the order of the task pools varies with the number of threads and does not correspond with the results of the C versions.

The speed-ups for the Java versions of the *radiosity* application are generally lower than those of the C versions. Furthermore, the ascent of the speed-up curves decreases with the number of processors (see Figures 18, 19 and 20).

Sun Fire 3800 and 6800

The Java versions of the *radiosity* application could only be measured for up to 20 processors on the larger of the two machines. The results are shown in Figures 21, 22 and 23.

Though the speed-up curves differ for different scenes, the individual task pools bring about very similar results.

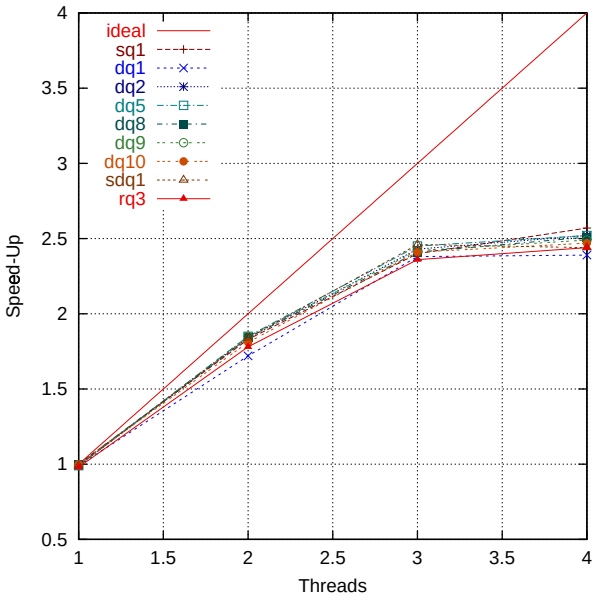

Figure 18. Speed-ups of the Java versions of the radiosity application (scene "largeroom") on the Sun E420R.

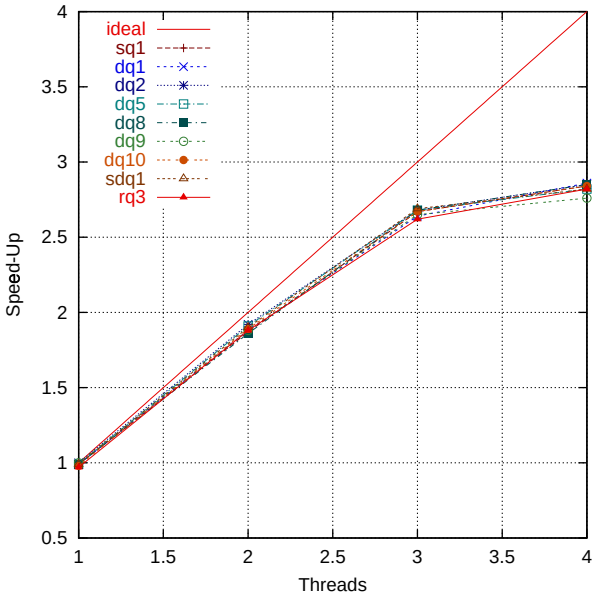

Figure 19. Speed-ups of the Java versions of the radiosity application (scene "Halle") on the Sun E420R.

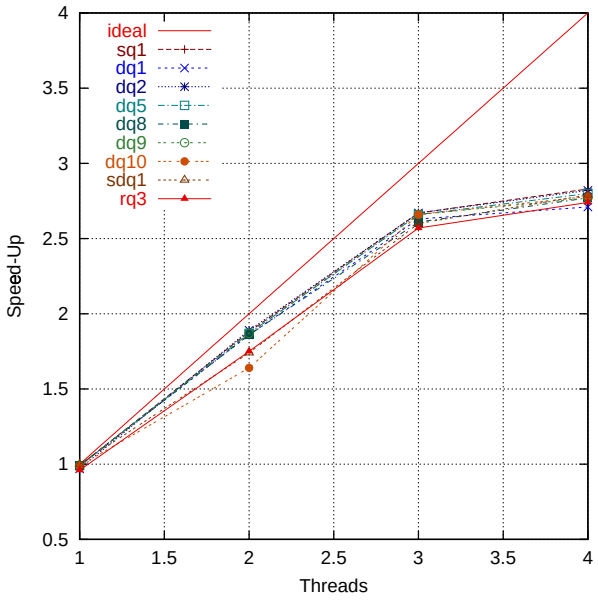

Figure 20. Speed-ups of the Java versions of the radiosity application (scene “Raum11w”) on the Sun E420R.

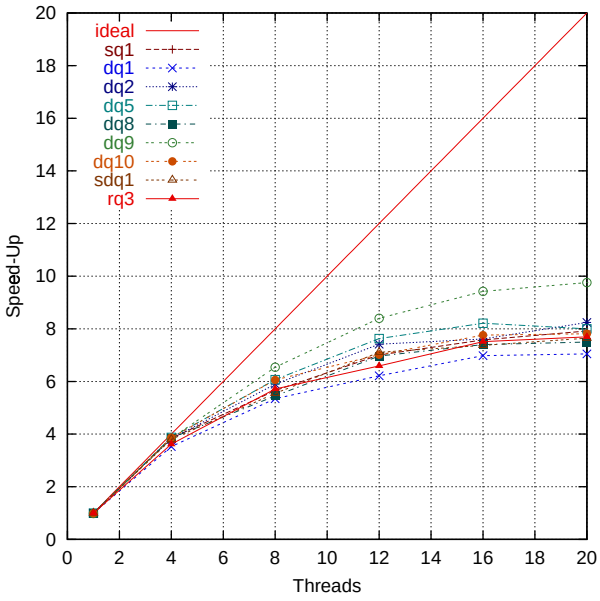

Figure 22. Speed-ups of the Java versions of the radiosity application (scene “Halle”) on the Sun Fire.

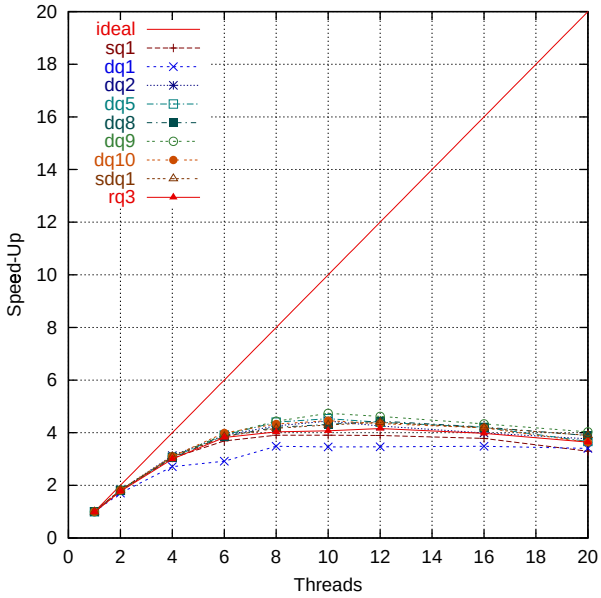

Figure 21. Speed-ups of the Java versions of the radiosity application (scene “largeroom”) on the Sun Fire.

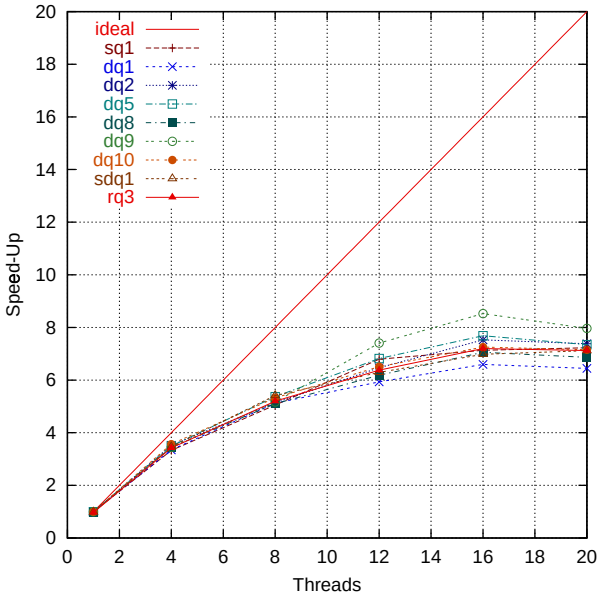

Figure 23. Speed-ups of the Java versions of the radiosity application (scene “Raum11w”) on the Sun Fire.

For scene “largeroom” the maximum speed-up measured was 4.74. It was measured with `dq9` running with 10 threads. `dq5`, `dq10` and `dq2` achieve the next best results. Their speed-ups are 4.54, 4.46 and 4.44, respectively. The speed-ups of `dq8`, `sdq1`, `rq3` and `sq1` range from 4.31 to 3.91. `dq1` can only obtain the worst speed-up of 3.46.

For scene “Halle” the task pools reach their maximum speed-ups with 16 or 20 threads. The best task pool is `dq9`, which achieves a speed-up of 9.76. `dq2`, `dq5`, `sq1` and `dq10` are next with speed-ups ranging from 8.24 to 7.81. The speed-ups of `rq3`, `sdq1` and `dq8` are lying between 7.70 and 7.50. `dq1` is the slowest with a speed-up of 7.05.

The maximum speed-ups for scene “Raum11w” were also reached with 16 or 20 threads. Again `dq9` obtains the best speed-up of 8.53. The speed-ups of `dq5` and `dq2` are 7.68 and 7.53, respectively. As for the other scenes, the slowest task pool is `dq1` with a speed-up of 6.60 The other task pools achieve speed-ups between 7.25 and 7.06

9. Related work

A lot of research has been done in developing *static* scheduling algorithms for DAGs. An overview can be found in the book of Brucker [4] or the article of Kwok and Ahmad [20]. Static scheduling provides that a complete task graph with task execution times and communication costs is given. This graph is used to compute a schedule that can later be used to execute the tasks of the DAG in an efficient order.

If task execution times cannot be modeled exactly, Tongsima, Chandrapornchai et al. have proposed modeling the execution times by probabilistic distributions [31] or fuzzy sets [7].

In special cases, static scheduling can be used to speed up irregular applications. Gerasoulis, Jiao and Yang [12] have applied the static scheduling system PYRROS [40] to the Fast Multipole Method (FMM).

In general, *dynamic* scheduling is necessary to exploit parallelism in irregular algorithms optimally. Various approaches have been proposed towards the efficient parallel execution of irregular computations: Johnson [17] proposes a Dynamic Task Graph (DTG) used to store tasks created at runtime. Cosnard, Jeannot and Yang [8] propose a Symbolic Linear Clustering (SLC) algorithm for Parameterized Task Graphs (PTGs). A parallel incremental scheduling algorithm is proposed by Wu [39].

Further approaches concentrate on automatic loop scheduling. Rauchwerger [28] proposes runtime parallelization techniques that use inspector/executor or speculative methods to schedule fully and partially parallel loops at runtime. Saltz et al. [16] have used the inspector/executor approach to implement the CHAOS runtime support system that provides a global address space for distributed arrays on message-passing machines.

A combined dynamic and static scheduling that simultaneously balances processor loads and maintains locality is the fractiling technique that combines factoring and tiling [15, 2]. This technique can be applied to parallel loops whose iterations have variable running times [2] and has been successfully used to the fast multipole method by Banicescu [1]. Different dynamic scheduling methods for parallel loops including factoring, weighted factoring and adaptive weighted factoring have been compared by Carino and Banicescu [6] for loops with different characteristics.

Another class of scheduling algorithms aims at distributing runnable tasks equally among the processors. This class of algorithms is usually referred to as *load balancing* algorithms. Kumar et al. [19] compare several load balancing techniques. Tucker [33] has done research in scheduling on multiprogrammed shared-memory multiprocessors. Durand et al. [10] study load balancing in self-scheduling schemes on Non-Uniform Memory Access (NUMA) machines.

Load balancing schemes using task queues are described in Dandamundi and Ayach [9], Rudolph et al. [29], Wen [36] and Podehl et al. [25]. Dandamundi and Ayach [9] present a processor scheduling scheme based on a hierarchical run queue organization. Rudolph et al. [29] propose a load balancing scheme using a collection of local workpiles. Wen [36] describes the implementation of a distributed task queue on the CM5. Podehl et al. [25] have used a parallel queue implemented by multiprefix operations to improve the performance of the *radiosity* application on the SB-PRAM.

10. Conclusions

We have presented results of several task pool implementations for shared-memory systems. These have been obtained using a synthetic algorithm and the *radiosity* application from the SPLASH-2 suite.

Task pools provide an easy way to implement irregular algorithms but have the disadvantage to compromise locality. The implementations presented have been designed to be usable with any task-based algorithm. In practice, task-based algorithms may use specialized task pools, that are optimized according to the needs of the algorithm.

The implementation of a task pool has great impact on its performance. In order to avoid bottlenecks, no central data structures should be used that are accessed concurrently. Therefore, from the task pools implemented in this paper, dynamic task stealing provides best scalability. The best of our implementations of dynamic task stealing uses a private and a public queue for each thread. Thus the number of synchronization operations is reduced as well as the number of

conflicts caused by task stealing, and speed-ups of 21.29 in C and 19.41 in Java could be measured using 22 processors.

The combination of central and distributed queues has shown good scalability as well. For the C versions of the synthetic algorithm its performance could nearly match dynamic task stealing with private and public queues. But for the Java versions its scalability was limited to 16 processors, probably due to the bottleneck created by the central queue.

As expected, central task pools show poor scalability because of the bottleneck that the central queue imposes. Distributed task pools which only provide a static data distribution are also not able to meet the performance of dynamic task stealing. Reducing lock granularity often decreases the performance due to higher overheads.

Most of our implementations execute wake-up calls every time a new task is inserted into the pool. Because in C these calls are executed very fast, this does not introduce much overhead but reduces the idle time of processors out of work. In Java the `notify()` calls provided for this purpose must only be called inside of a corresponding critical region. Thus a bottleneck is created that limits the scalability of most of our Java implementations and hinders a comparison of these implementations for larger numbers of threads.

But the C versions of the synthetic algorithm allow a good comparison of the task pools. It can be observed that queues implemented by arrays do better than the corresponding list implementations. Using heuristics to steal larger tasks can slightly improve the performance of dynamic task stealing. Even though the efficiency of randomized task pools is lower than that of dynamic task stealing, randomized task pools can profit from all 24 processor that have been available.

Our investigations of the cache behavior have shown that locality is exploited best if distributed queues are used which are processed in LIFO order. Especially central queues and randomized behavior increase the cache miss rates.

Due to the high number of memory operations typically executed in a task-based algorithm it is very important to use an appropriate memory manager. Recycling memory blocks in distributed data structures and anticipatory allocation strategies can remarkably improve the performance. This way, on the Sun E420R the execution time of the C versions of the synthetic algorithm could be reduced by more than 90 %. Like the task pools, memory managers should not use central data structures that are accessed concurrently.

The *radiosity* application is a typical irregular application that intensively uses common variables stored in the shared-memory. Because accesses to these variables must often be synchronized, additional limitations to scalability emerge, and often the maximum speed-up is already reached with less than 24 processors.

The speed-ups of the task pools measured for the *radiosity* application are often very close to each other. Because these small differences are influenced by errors of the same order in the measurement introduced by the operating system and other user processes, a definite comparison of the task pools is not possible for this application.

With the development of Java Virtual Machines, the execution times of Java programs get closer to C. But those compilers make it more difficult to compare Java programs, because their optimizations speed up different programs differently.

In general, the Java implementations reach smaller speed-ups than the C versions. One reason for this is that the synchronization mechanisms of Java are not as flexible as those of POSIX threads. There are no independent operations for locking and unlocking of objects, and the call to wake up conditionally waiting threads must be protected by the corresponding object lock.

Another reason is that the Java programming language does not allow to use memory blocks as it can be done in C, and therefore many optimizations used in C programs can not be applied in Java. Particularly, a memory manager in Java would probably be more expensive than using the built-in functions of the language.

Furthermore, the scalability of any Java application is limited by the ability of the virtual machine used to execute it. Particularly, in the virtual machine we have used only a sequential garbage collector was implemented. But there are other sources of performance loss in the structure of the Java programming language. For example, efficient memory management is far more complicated, and the synchronization mechanisms of Java are not as flexible as those of POSIX threads.

Of course there are many other task pool implementations that may be thought of. For example, the balancing of tasks could be done in a separate phase or by a separate thread. Other implementations can even model the whole task pool as a graph that is searched by all processors in parallel [18]. Also many variations and improvements of our implementations are possible.

The implementations we have presented only show the general capability of task pools to be used for parallel irregular algorithms. Even though load balancing strategies that are adapted to the specific application often lead to better performance, in our example, the *radiosity* application, scalability was limited by the application itself. The results for the synthetic algorithm proved that good task pool implementations are well able to efficiently use all processors of our machines.

Future investigations may concern alternative algorithms as well as new types of task pools. Most interesting al-

gorithms are real-world applications which are deterministic and provide extensive user control. New types of task pools should be aiming at a further reduction of the overhead caused by the thread library and at avoiding the bottlenecks present in most of the task pools presented in this paper. Further investigations may also consider heuristic approaches to improve the schedule.

References

[1] I. Banicescu. *Load Balancing and Data Locality in the Parallelization of the Fast Multipole Algorithm Parallelism*. PhD thesis, Polytechnic University, 1996.

[2] I. Banicescu and S. F. Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *Proc. Supercomputing'95 Conference*, 1995.

[3] R. Berrendorf and H. Ziegler. PCL – The performance counter library: A common interface to access hardware performance counters on microprocessors. Internal report FZJ-ZAM-IB-9816, Forschungszentrum Jülich, 10 1998.

[4] P. Brucker. *Scheduling Algorithms*. Springer, Berlin, 3rd edition, 2001.

[5] D. R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[6] R. Carino and I. Banicescu. Dynamic scheduling of parallel loops with variable iterate execution times. In *Proc. IPDPS-Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*, 2002.

[7] C. Chantrapornchai, S. Tongsima, and E. H. Sha. Imprecise task schedule. In *Proceedings of the International Conference on Fuzzy Systems*, 1997.

[8] M. Cosnard, E. Jeannot, and T. Yang. SLC: Symbolic scheduling for executing parameterized task graphs on multiprocesors. In *International Conference on Parallel Processing*, 1999.

[9] S. P. Dandamudi and S. Ayachi. Performance of hierarchical processor scheduling in shared-memory multiprocessor systems. *IEEE Transactions on Computers*, 48(11):1202–1213, 1999.

[10] D. Durand, T. Montaut, L. Kervella, and W. Jalby. Load balancing performance of dynamic scheduling on NUMA multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 1201–1214, 11 1996.

[11] S. J. Eggers and R. H. Katz. The effects of sharing on the cache and bus performance of parallel programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 257–270, 4 1989.

[12] A. Gerasoulis, J. Jiao, and T. Yang. Experience with graph scheduling for mapping irregular scientific computation. In *Proceedings of the First IPPS Workshop on Solving Irregular Problems on Distributed Memory Machines*, 4 1995.

[13] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. In *Proceedings of SIGGRAPH*, 1991.

[14] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[15] S. Hummel, E. Schonberg, and L. Flynn. Factoring: A practical and robust method for scheduling parallel loops. *Comm. of the ACM*, 35(8):90–101, 1992.

[16] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software – Practice and Experience*, 1995.

[17] T. Johnson, T. A. Davis, and S. M. Hadfield. A concurrent dynamic task graph. *Parallel Computing*, 22(2):327–333, 1996.

[18] M. Korch. Einsatz von Taskpools in Pthreads und Java zur parallelen Implementierung irregulärer Algorithmen. Diplomarbeit, Martin-Luther-Universität Halle-Wittenberg, 2001.

[19] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[20] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[21] T. G. Lewis. *Foundations of Parallel Programming: A Machine-Independent Approach*. IEEE Computer Society Press, Washington, 1993.

[22] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensional equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*, 1998.

[23] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[24] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 2nd edition, Jan. 1999.

[25] A. Podehl, T. Rauber, and G. Rünger. Scalability and granularity issues of the hierarchical radiosity method. In *Proceedings of the Euro-Par '96*, volume 1, pages 789–798, Berlin, Germany, 1996. Springer-Verlag.

[26] A. Podehl, T. Rauber, and G. Rünger. A shared-memory implementation of the hierarchical radiosity method. *Theoretical Computer Science*, 196(1–2):215–240, 1998.

[27] S. Prakash, Y.-H. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical Report 91–002, Department of Computer and Information Sciences, University of Florida, 7 1991.

[28] L. Rauchwerger. Run-time parallelization: It's time has come. In *International Conference on Parallel Computing*, 1998.

[29] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[30] J. P. Singh, C. Holt, T. Tosuka, A. Gupta, and J. L. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, 6 1995.

[31] S. Tongsima, C. Chantrapornchai, E. H.-M. Sha, and N. Passos. Probabilistic rotation: Scheduling graphs with uncertain execution time. In *Proceedings of the Internationall Conference on Parallel Processing*, pages 292–297, 1997.

[32] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[33] A. Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 12 1993.

[34] A. M. Turing, P. N. Furbank, D. Ince, P. T. Saunders, J. L. Britton, R. Gandy, and C. E. M. Yates. *Collected Works of A. M. Turing*. North-Holland, Amsterdam, London, 1992, 2001.

[35] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[36] C. Wen. A distributed task queue for load balancing on the CM5.

[37] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 199–206, Santa Barbara, California, 1995.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[39] M.-Y. Wu. Parallel incremental scheduling. *Parallel Processing Letters*, 8 1995.

[40] T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437, Washington D.C., 7 1992.