Performance evaluation of task pools based on hardware synchronization

Ralf Hoffmann
ralf.hoffmann@uni-bayreuth.de

Matthias Korch
matthias.korch@uni-bayreuth.de

Thomas Rauber
rauber@uni-bayreuth.de

University of Bayreuth, Department of Computer Science
95440 Bayreuth, Germany

ABSTRACT

A task-based execution provides a universal approach to dynamic load balancing for irregular applications. Tasks are arbitrary units of work that are created dynamically at runtime and that are stored in a parallel data structure, the *task pool*, until they are scheduled onto a processor for execution. In this paper, we evaluate the performance of different task pool implementations for shared-memory computer systems using several realistic applications. We consider task pools with different data structures, different load balancing strategies and a specialized memory management. In particular, we use synchronization operations based on hardware support that is available on many modern microprocessors. We show that the resulting task pool implementations lead to a much better performance than implementations using Pthreads library calls for synchronization. The applications considered are parallel quicksort, volume rendering, ray tracing, and hierarchical radiosity. The target machines are an IBM p690 server and a SunFire 6800.

1. INTRODUCTION

Irregular applications are characterized by an unpredictable computational structure induced by the input data. This does not only concern the actual computations but also the access pattern to different data structures and their distribution. Thus, when solving such problems with multiprocessor machines, a static work distribution does not lead to an optimal load balance and, therefore, dynamic load balancing is required to execute such applications efficiently.

A universal approach to balance the load dynamically is the use of *task pools*. For that purpose, the application is split into several *types of tasks* which can be used as the minimum unit of parallelism. Every task type specifies a sequence of operations to be performed. At runtime, *tasks*, i.e., instances of these task types with corresponding arguments, are created dynamically. After their creation, the tasks are stored in the task pool until they are scheduled onto a processor for execution.

Typically, tasks can create new tasks during their execution. Thus, the visualization of the parent-child dependences between tasks during a typical run of a task-based application leads to an irregularly structured directed acyclic graph (DAG), the specific structure of which depends on the input data. The typical framework of a task-based application is illustrated in Figure 1. An introduction to the conception of task pools and different implementation strategies are presented in [20].

The contribution of this paper is to present task pool implementations that use hardware support to reduce the synchronization overhead. The impact on the performance is evaluated in extensive runtime experiments. The new task pools perform synchronization using either the hardware specific operations *Load & Reserve* and *Store Conditional* or *Compare & Swap*. We show that a suitable combination of synchronization method, memory management and load balancing leads to implementations that show a much better performance than task pools that use a pure software synchronization based on Pthreads [6]. This is shown by considering several realistic applications: parallel quicksort, volume rendering, ray tracing, and the hierarchical radiosity method. The applications use one thread per processor and each thread requests tasks from the task pool and executes them sequentially. In many cases, our task pool implementations obtain a higher performance than the application-specific load balancing strategies employed in the original source code. This is particularly interesting, since the task pools provide a general approach that cannot react on specific characteristics of the application whereas specialized load balancing strategies are tailored for one dedicated application.

The following applications are considered in the comparison: *Parallel quicksort* is a fast general sorting algorithm based on the divide and conquer programming paradigm. The investigated parallel implementation exploits the parallelism contained in the divide step. *Volume rendering* is used to visualize volume data, e.g., computer tomography data, by casting rays into the three-dimensional volume and sampling color and opacity along the rays. The parallel implementation is based on an image-order algorithm where separate image tiles are processed in parallel. *Ray tracing* is used to render three-dimensional geometric scenes to a two-dimensional image, particularly taking specular reflection into account. Similar to the volume rendering application,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC2004 Pittsburgh, Pennsylvania USA
0-7695-2153-3/04 $20.00 (c) 2004 IEEE.

```
struct Task { Type, Arguments };

// 1. Initialization Phase
for (each work unit U of the input data)
    TaskPool.create_initial_task(U.Type,
        U.Arguments);

// 2. Working Phase
each processor:
    loop
    {
        Task T ← TaskPool.get();
        if (T = ∅) exit;
        T.execute(); // may create new tasks
        T.free();
    }
```

Figure 1: Framework of a task-based application.

```
register R ← ∅; // reservation register

LR(address A, register B)
{
    R ← A; // set reservation
    B ← load(A);
}

SC(address A, register B)
{
    // check reservation
    if (R ≠ A) return false;

    store(A, B);
    R ← ∅ // clear reservation;
    return true;
}
```

Figure 2: The operations *Load & Reserve* and *Store Conditional*.

rays are cast through each image pixel, and parallelization is performed based on image tiles. The *hierarchical radiosity method* is a global illumination algorithm that renders geometric scenes computing the equilibrium distribution of light. For that purpose, a hierarchical subdivision of the object surfaces is performed at runtime, and interactions between the surface elements representing the transport of light are evaluated. Parallelism is exploited across interactions and subdivision elements.

The comparison of the task pools is performed using an IBM p690 server and a SunFire 6800 server. The IBM p690 is a symmetric multiprocessor (SMP) with a deep memory hierarchy consisting of several cache levels and hardware-supported distributed shared memory. The 32 Power4 processors of this machine are running at 1.7 GHz. The Power4 processor [38] provides hardware support for the emulation of atomic read-modify-write (RMW) operations by two special read and write instructions, *Load & Reserve* (LR) and *Store Conditional* (SC). The LR and SC operations can be combined with different modify instructions or even instruction sequences. If a memory location is read to a general purpose register using LR, the executing Power4 processor sets a reservation on this memory location in a special register. This reservation is released as soon as any processor issues a write operation targeting this location or another memory location that shares the same cache line. After reading the value to a register, the register can be modified using arbitrary machine instructions. Writing the value back to the memory location using SC only succeeds if there exists a valid reservation on the respective address in the executing processor (Figure 2). Thus, the complete RMW cycle needs to be executed repeatedly until the SC operation succeeds if a behavior similar to the atomic RMW operations of other architectures like *Compare & Swap* (CAS) and *Fetch & Op* (FOP) shall be emulated. The SunFire 6800 server is an SMP with 24 UltraSPARC3 processors running at 1.0 GHz. The processors provide atomic operations for hardware synchronization. This includes the atomic SWAP operation for unconditional data swaps and the atomic *Compare & Swap* for conditional data swaps. In contrast to LR/SC, these operations cannot be interrupted by other processors.

The rest of the paper is organized as follows: Section 2 describes the different task pool implementations. Sections 3–6 present a description of the irregular applications considered and show runtime experiments performed with these applications. Section 7 presents related work, and Section 8 concludes the paper.

2. TASK POOL IMPLEMENTATIONS

In this section, we describe the different central and distributed task pools used in our comparison and their realization. The task pools based on Pthreads are implemented in C to retain maximum portability. All task pools using hardware synchronization operations are implemented in assembler since this enables easier and more efficient usage of the hardware operations. Significant performance improvements resulting from the use of assembler language are not expected because the code structure of the task pools is suitable to be translated into very fast machine code by modern optimizing compilers. There are no extensive calculations, and most time is spent on synchronization operations and waiting for other processors.

2.1 Central task pools

Central task pools store the tasks in a single queue, which is accessed by all processors concurrently. Thus, an optimal load balance is possible, but there may be a bottleneck caused by concurrent accesses to the queue, particularly if the tasks are fine-grained and the queue is accessed frequently. All central task pools we investigate have in common that the tasks are processed in LIFO (last in first out) order, thus the queue is working as a stack.

- **SQ-PTH** (single queue, Pthreads lock): This task pool implementation uses a single-linked list as the central queue. Race conditions due to concurrent accesses to the queue are prevented by using a Pthreads lock.
- **SQ-SL** (single queue, simple lock): This implementation uses a simple spin-lock for mutual exclusion instead of the Pthreads lock. On the SPARC system we implemented this lock using *Compare & Swap* (CAS) on a shared memory location as a global lock variable, see Figure 4 [37]. On the Power4 processor we use the LR/SC operations to implement a behavior similar to CAS, see Figure 3.

```
lock:
      lwarx    <TReg>, 0, <AddrReg>
      cmpwi    <TReg>, 0
      bne-     lock
      li       <TReg>, 1
      stwcx.   <TReg>, 0, <AddrReg>
      bne-     lock
      isync
unlock:
      sync
      li       <TReg>, 0
      stw      <TReg>, 0(<AddrReg>)
```

Figure 3: The simple lock implementation on the Power4 CPU.

```
lock:
      mov      1, <TReg>
      cas      [<AddrReg>], %g0, <TReg>
      tst      <TReg>
      be       cont
      nop
wait:
      ld       [<AddrReg>], <TReg>
      tst      <TReg>
      bne      wait
      nop
      ba,a     lock
cont:
      membar   #LoadLoad | #LoadStore
unlock:
      membar   #StoreStore
      membar   #LoadStore
      st       %g0, [<AddrReg>]
```

Figure 4: The simple lock implementation on the SPARC CPU.

- **SQ-TL** (single queue, ticket lock): To improve the scalability, the simple lock is replaced by a scalable and fair ticket lock (cf. [27]). On the Power4 CPU we use the LR/SC operations to implement the atomic increment of the global ticket counter, see Figure 5. This implementation loads the current value of the counter with the LR operation and stores the new value with the SC operation immediately after the increment. On the SPARC CPU we use the *Compare & Swap* operation to implement the ticket lock, see Figure 6.

  The ticket lock generally offers a better performance at the cost of a possibly degraded performance on heavily loaded machines. The reason is that the ticket lock is a fair lock, i.e., the processors gain access to the protected area in the order in which they arrive. An interrupted processor in the wait queue blocks all subsequently arriving processor.

- **SQ-LF** (single queue, lock-free): This implementation tries to modify the task queue directly without the need of acquiring a lock using a lock-free approach similar to [28]. The idea is to modify the head of the queue atomically, such that the list is always consistent. On the SPARC system we use *Compare & Swap* to implement this lock free behavior, see Figure 8. We atomically modify the pointer to an element and a counter to overcome the ABA problem (cf. [28]). On the Power4 CPU we use LR/SC instead of CAS to perform the atomic modification, see Figure 7. This permits an easier implementation and simultaneously avoids the ABA problem.

```
lock:
      lwarx    <TReg1>, 0, <AddrRegTicket>
      addi     <TReg2>, <TReg1>, 1
      stwcx.   <TReg2>, 0, <TReg1>
      bne-     lock
wait
      lwz      <TReg2>, 0(<AddrRegServe>)
      cmpw     <TReg2>, <TReg1>
      bne      wait
      isync
unlock:
      sync
      lwz      <TReg>, 0(<AddrRegServe>)
      addi     <TReg>, <TReg>, 1
      stw      <TReg>, 0(<AddrRegServe>)
```

Figure 5: The ticket lock implementation on the Power4 CPU.

```
lock:
      ld       [<AddrRegTicket>], <TReg1>
      add      <TReg1>, 1, <TReg2>
      cas      [<AddrRegTicket>], <TReg1>, <TReg2>
      cmp      <TReg1>, <TReg2>
      bne      lock
      nop
wait:
      ld       [<AddrRegServe>], <TReg2>
      cmp      <TReg1>, <TReg2>
      bne      wait
      nop
unlock:
      ld       [<AddrRegServe>], <TReg1>
      inc      <TReg1>
      st       <TReg1>, [<AddrRegServe>]
```

Figure 6: The ticket lock implementation on the SPARC CPU.

```
insert:
      ldarx    <TReg>, 0, <AddrRegList>
      std      <TReg>, Offset(<AddrRegElem>,NEXT)
      sync
      stdcx.   <AddrRegElem>, 0, <AddrList>
      bne-     insert
remove:
      lwa      r30, ADR(tpool_n)
      la       r31, ADR(tpool)
loop1:
      ldarx    <TReg1>, 0, <AddrRegList>
      cmpdi    <TReg1>, 0
      beq      empty
      ld       <TReg2>, Offset(<TReg1>,NEXT)
      stdcx.   <TReg2>, 0, <AddrRegList>
      bne-     loop1
```

Figure 7: The lock free implementation on the Power4 CPU.

```
insert:
      sethi   %hi(0xffffffff), <TReg2>
      or      <TReg2>, %lo(0xffffffff), <TReg2>
loop1:
      ldx     [<AddrRegList>], <TReg1>
      and     <TReg1>, <TReg2>, <TRegCount>
      srlx    <TReg1>, 32, <TRegElem>
      st      <TRegElem>, Offset(<AddrRegElem>,NEXT)
      inc     <TRegCount>
      and     <TRegCount>, <TReg2>, <TRegCount>
      sllx    <AddrRegElem>, 32, <TReg3>
      or      <TRegCount>, <TReg3>, <TReg3>
      casx    [<AddrRegList>], <TReg1>, <TReg3>
      cmp     <TReg1>, <TReg3>
      bne     loop1
      nop
remove:
      ldx     [<AddrRegList>], <TReg1>
      and     <TReg1>, <TReg2>, <TRegCount>
      srlx    <TReg1>, 32, <TRegElem>
      tst     <TRegElem>
      beq     empty
      nop
      ld      Offset(<TRegElem>,NEXT), <TReg4>
      inc     <TRegCount>
      and     <TRegCount>, <TReg2>, <TRegCount>
      sllx    <TReg4>, 32, <TReg3>
      or      <TRegCount>, <TReg3>, <TReg3>
      casx    [<AddrRegList>], <TReg1>, <TReg3>
      cmp     <TReg1>, <TReg3>
      bne     remove
      nop
```

Figure 8: The lock free implementation on the SPARC CPU.

2.2 Distributed task pools

Using several queues distributed among the processors can improve the performance of a task pool by reducing the overhead and the number of access conflicts. To achieve a good load balance, the processors may need to search several queues for new work if their local queues are empty. This is called *task stealing*. Because new tasks are stored locally and, as long as no task queue gets empty, all processors can execute locally created tasks, concurrent accesses due to stealing are less frequent than accesses to the single queue of a central task pool. This execution scheme may also improve the data locality. To reduce the number of conflicts in the stealing process, the search for new tasks is controlled by an array for each processor containing the permuted IDs of all processors. Using these arrays, every processor checks the queues in a different order.

2.2.1 Distributed task pools without grouping

We consider several implementations of distributed task pools without grouping of tasks using different synchronization operations:

- **DQ-PTH** (distributed queue, Pthreads lock): Each processor is assigned a single-linked list as a local queue. One Pthreads lock is used per queue to avoid race conditions.
- **DQ-SL** (distributed queue, simple lock): In contrast to DQ-PTH, this task pool uses simple locks based on LR/SC or *Compare & Swap* to protect concurrent accesses to the local queues.
- **DQ-SL-PRIV** (distributed queue, simple lock, private buffer): The use of an additional private task queue per processor can improve the performance of a task pool since access conflicts due to task stealing become less frequent. However, this may lead to load imbalance, because tasks stored in a private queue are not available to other processors. In this implementation, we use a private buffer that can store a single task to reduce the number of access conflicts and keep the load imbalance at a minimum.
- **DQ-SL-PRIV-C** (distributed queue, simple lock, private buffer, cache optimized): In addition to the use of a private buffer, this implementation aligns the data structures of the task pool to the cache line size of the processor. So the private buffer, the public queue, the lock variable and additional data for each processor are stored in different cache lines. This improves the performance by reducing unnecessary cache invalidation. For example, the reservation set by the LR operation on the Power4 system covers not only the specified memory address. Instead, any write operation to the memory area corresponding to the cache line in which the reserved address is located invalidates the reservation even if this write operation is not related to the synchronization operation.
- **DQ-SL-TRY-PRIV-C** (distributed queue, simple lock, single try, private buffer, cache optimized): This implementation improves the stealing process by using a seminonblocking approach. If a lock on a task queue cannot be acquired, this implementation does not block but immediately continues with the queue of the next processor in its search order.
- **DQ-TL-PRIV-C** (distributed queue, ticket lock, private buffer, cache optimized): Since the ticket lock used in SQ-TL increased the performance compared to the simple lock in SQ-SL, this implementation uses a ticket lock realized by LR/SC (Power4 CPU) or *Compare & Swap* (SPARC CPU) to protect the distributed queues. All other implementation details correspond to DQ-SL-PRIV-C.
- **DQ-CLH-PRIV-C** (distributed queue, CLH lock, private buffer, cache optimized): This implementation is based on the CLH lock [26], another fair lock that reduces the memory bandwidth required even further. The implementation of this lock only requires an atomic *Fetch & Store* operation that can be emulated easily using LR/SC. Since the lock uses one memory location per processor to store a pointer to a memory location of another processor (thus realizing an implicit list of waiting processors), slightly more memory space is required than for the ticket lock.
- **DQ-SL-SP-PRIV-C** (distributed queue, simple lock with SINGLEPUT extension, private buffer, cache optimized): This implementation exploits the fact that enqueuing of new tasks into a queue can only be performed by the processor assigned to the queue. Changing the storage order of the task queue from LIFO (stack) to FIFO (first in first out), task removal is performed at the head of the queue and insertion is done at the tail. Thus, only dequeuing of tasks, i.e., task stealing or fetching of tasks by the local processor, requires synchronization. Therefore, the head of the queue is protected by a simple lock. By inserting a dummy element into the list, the queue owner can insert new tasks without synchronization. The FIFO order of the queue may lead to a decrease in data locality because newly created tasks are executed only after all

previously created tasks have been executed. Thus, the output data of the parent task that might be required as the input data of the new task may no longer be in the cache.

- **DQ-LF-PRIV-C** (distributed queue, lock-free, private buffer, cache optimized): This task pool has been implemented on the Power4 and the SPARC processor similarly to DQ-SL-PRIV-C, but here we use the same lock-free method as for SQ-LF described in Section 2.1 to avoid contention adherent to locks.

2.2.2 Block-distributed task pools

The number of task stealing operations and thus the number of access conflicts involved in this operation can be reduced by applying a block-oriented grouping of the tasks such that a task stealing operation always retrieves one block containing several tasks. This block-oriented approach can also improve the data locality because the tasks combined in one block often access closely related data. A further reduction of the number of access conflicts is possible by using private queues that are accessed by the local processor exclusively in addition to the conventional local queues, which are publicly available to all other participating processors.

The drawback of both approaches is that they may have a negative effect on the load balance since the grouping of tasks increases the granularity of the work that can be moved to other processors, and tasks stored in a private queue can only be processed by the owner of that queue even if other processors are idle. Therefore, the blocksize of the task blocks and the capacity of the private queues should be chosen carefully in order to find the best compromise between load balance and contention due to task stealing. The right choice depends on the granularity of the application considered, but also on the speed and the architecture of the parallel computer system used. In all the experiments we present, a blocksize of four tasks and public queues that can store two task blocks have been used since these parameters present a good compromise in many experimental scenarios.

We include the following block-distributed task pools in our comparison:

- **DQ-BL-PTH** (block distributed queue, Pthreads lock) is a block-distributed task pool implemented as described above that uses Pthreads locks to protect the public queues.
- **DQ-BL-SL-C** (block-distributed queue, simple lock, cache optimized): This implementation uses cache-aligned data structures and protects concurrent accesses to the public queues by a simple lock.
- **DQ-BL-SL-TRY-C** (block-distributed queue, simple lock, single try, cache optimized) uses the semi-nonblocking approach similar to DQ-SL-TRY-PRIV-C.
- **DQ-BL-TL-C** (block-distributed queue, ticket lock, cache optimized) uses ticket locks to realize mutual exclusion on the public queues.
- **DQ-BL-SP-C** (block-distributed queue, simple lock with SINGLEPUT, cache optimized) utilizes the SINGLEPUT approach that exploits the exclusive insertion strategy described for DQ-SL-SP-PRIV-C. Additionally we use the semi-nonblocking method to protect the removal of tasks from public queues.
- **DQ-BL-TL-TRY-C** (block-distributed queue, ticket lock, single try, cache optimized): To combine the semi-nonblocking method with a ticket lock, we implement a special ticket lock that only fetches a ticket (i.e., increases the counter) when the lock is free (i.e., the second counter is equal to the first counter). The resulting implementation uses this method to continue the search procedure on another processor when the tested lock is not free.

2.3 Memory management

The memory management is an important part of a task pool implementation because memory must be allocated or freed every time a new tasks is created and every time a task finishes its execution. Typically, the total number of tasks created by an irregular application is very large in order to achieve a good load balance. The memory space required to store the information associated with a single task typically is relatively small, i.e., only a few bytes. Using operating system calls to allocate and free a large number of such small memory blocks often leads to a large overhead and may even create a severe bottleneck if, for example, the memory manager of the operating system relies on centralized data structures.

Therefore, several scalable and efficient memory managers supporting general memory blocks have been proposed, e.g., [2, 41, 42]. However, a specialized memory manager that exploits the special properties of task pools can obtain a higher performance than those general methods. For example, we can take advantage of the fact that in our task pool implementations the size of the data structure that stores the task type and the arguments of a task has a fixed size.

The memory manager of a task pool can use various approaches to reduce the number of system calls. The most important 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, freed blocks are collected in free-lists. If a processor later requests a block of the same type, it can be taken from one of the free-lists.

In the evaluation of the task pools we use a memory management strategy based on private distributed free-lists. Thus, no synchronization operations are issued by the memory manager and we can observe the scalability of the task pools without interference provoked by the memory manager. We use different implementations of this memory management strategy in our evaluation: all task pools based on Pthreads synchronization, which are implemented in C, are executed with an implementation of the memory manager written in C, whereas task pools utilizing hardware operations for synchronization, which are implemented in the assembler language of the respective machines, are executed with an assembler version of the memory manager.

3. PARALLEL QUICKSORT

Quicksort [17] is a fast and efficient general sorting algorithm. Typically, sequential quicksort is implemented using recursive procedure calls following the divide-and-conquer paradigm. Our implementation sorts integer arrays of variable size which are initialized with random numbers. In order to enable the use of the same input data in different program runs, the random seed can be specified as an argument to the program. For parallelization, we exploit the parallelism present in the divide step by creating two new tasks where a sequential implementation would perform re-

cursive procedure calls. This leads to an irregular computational structure since the divide step may split the array into two uneven parts and a different number of swap operations may be necessary to partition the sub-arrays.

Because at the beginning of the algorithm only one task exists, and every task creates two new tasks if the size of its input is large enough to be split, the theoretical parallel execution time of this parallel quicksort implementation in the average case is given by

$$\begin{aligned} T_{\text{par}}(N,p) &= \sum_{i=1}^{\lceil \log p \rceil} \frac{N}{2^{i-1}} + \sum_{i=\lceil \log p \rceil+1}^{\log N} \frac{N}{p} \\ &= 2N\left(1 - \frac{1}{2^{\lceil \log p \rceil}}\right) + \frac{N}{p}(\log N - \lceil \log p \rceil), \end{aligned} \quad (1)$$

where N is the size of the array to be sorted and p is the number of processors.

We have performed experiments using this quicksort implementation with all task pools described using different array sizes and random seeds. Figure 9 presents the speedups measured on the IBM p690 system for selected task pools using an array size of $100,000,000$. The task pools shown present the best task pools of each category of central, distributed or block-distributed task pools using Pthreads or hardware synchronization. Using Equation (1) we calculate the ideal speedup $S_{\text{ideal}}(N)$ of the parallel quicksort by

$$S_{\text{ideal}}(N,p) = \frac{T_{\text{seq}}(N)}{T_{\text{par}}(N,p)} = \frac{N \cdot \log N}{T_{\text{par}}(N,p)}. \quad (2)$$

To reach this ideal speedup the quicksort implementation would have to split the array into two equally sized parts to provide a sufficient degree of parallelism but the array partitions usually are of slightly different sizes. Equation (2) also does not take the additional overhead due to the task based execution into account. Ideally we can stop the creation of new tasks at a level of $\log p$ where p is the number of threads. But because the quicksort algorithm usually does not split the array equally, it is reasonable to provide more tasks to keep all threads busy. To limit the overhead introduced by the task handling we stop the task creation at an array size of $1,000$. As shown in Figure 9, the parallel implementation cannot achieve the ideal speedup for a larger number of processors.

The block-distributed task pools perform worst. The task grouping usually reduces the overhead of task stealing by transferring several tasks at once to another thread. But for the parallel quicksort application, the task grouping hinders other processors to execute tasks that were available if the task grouping had not been used. Each task creates two tasks for sufficiently large array sizes where both new tasks should be executed by different threads. By storing tasks in groups of tasks, these new tasks are not available to waiting threads and therefore the maximum parallelism available cannot be exploited. The best speedup achieved is 2.86 for 8 threads, so block-distributed task pools only offer an efficiency of 0.36.

The central and distributed task pools can utilize the parallelism provided by the quicksort algorithm and perform significantly better. For 8 threads they can even achieve a better speedup than the calculated ideal speedup of 5.66. The distributed task pool using Pthreads synchronization operations even reaches a speedup of 5.86 for 8 threads. This is due to an increase in locality, which is not covered by Equation (1).

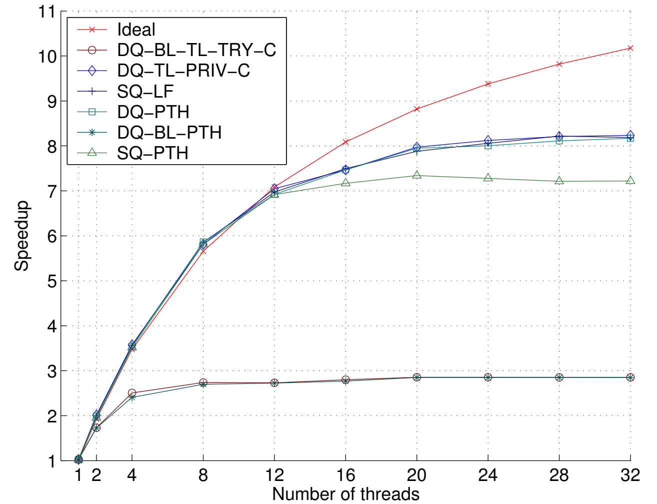

Figure 9: Speedup of parallel quicksort on the IBM p690 for selected task pools using an array size of 100,000,000.

The central task pool using Pthreads synchronization operations achieves a maximum speedup of 7.34 with 20 threads. For the distributed task pools, the Pthreads implementation achieves a maximum speedup of 8.17 with 32 threads and the task pool using hardware operations for the ticket lock is 0.7 % faster (speedup 8.23). The lock-free central task pool implemented with hardware synchronization operations (SQ-LF) has a much lower overhead and can outperform the central task pool that uses Pthreads locks. With a speedup of 8.22 for 28 threads this task pool implementation is even slightly better than the distributed task pool with Pthreads synchronization operations.

To point out that the runtime differences are not due to the usage of assembler language, we consider the sequential runtime of the different implementations. The difference between the average execution times of the best Pthreads task pool and the best assembler task pool is less then 0.1 % while the variation of the execution time for multiple runs is almost 2 %. The benefit of using hardware synchronization operations is only marginal with 0.7 %. The reason is that the major part of the runtime is spent with larger tasks from the first levels of quicksort. Thus, the influence of the task pool is small and the new task pools cannot improve the performance significantly.

In summary, because of its task creation scheme, our parallel implementation of quicksort, which exploits the parallelism contained in the divide step, cannot obtain a linear scalability. In fact, the maximum speedup attainable is determined by the size of the input and also the input data itself since they influence the number of tasks created and the balance of the resulting task graph. But compared with the theoretical maximum speedup our task pool implementations obtain a good performance. Using a random array containing $100,000,000$ elements where the theoretical maximum speedup is 10.2, a speedup of 8.2 could be measured for the best of our task pool implementations. Distributed Task pools based on hardware synchronization are slightly faster than the corresponding Pthreads implementations but due to the larger task size, the influence

of different synchronization methods is only marginal. The central task pool utilizing a lock-free strategy could even outperform a distributed task pool based on Pthreads synchronization. In contrast to the central Pthreads task pool, the use of hardware synchronization operations in the central assembler task pool improves the performance so there are no significant performance differences between the three different task pool types.

4. VOLUME RENDERING

The second benchmark considered is a volume rendering algorithm that is part of the SPLASH-2 application suite [43] (*volrend* application [29]). Volume rendering is a technique used to visualize sampled functions of three spatial dimensions, e.g., electron density maps for molecular dynamics, magnetic resonance (MR) and computed tomography (CT) data. To compute a 2D image, rays are cast into the volume data for each pixel. The final color of the pixels is computed by compositing color and opacity at evenly spaced locations along the ray. To speed up the algorithm, adaptive methods, such as hierarchical spatial enumeration [23], adaptive termination of ray tracing [23], and adaptive refinement [24] can be applied.

The parallel algorithm divides the image into equally sized contiguous blocks that can be used as the initial work distribution. Each block consists of a number of tiles used as the minimum work unit. To balance the work between the processors, a task based execution scheme is used where only one task type exists that processes one of the image tiles. No additional tasks are created during the working phase. If adaptive refinement is not applied, no synchronization on shared variables is necessary since the image tiles are non-overlapping and the volume data are read-only. However, if adaptive refinement is used, pixel sharing arises because adjacent image sample regions share the corner pixel values required for measuring their image complexity. The communication of these pixel values avoids the cost of replicated ray tracing but demands for synchronization.

Figure 10 shows runtime experiments performed on the Power4 system for a scene called *head* where we use all adaptive methods provided by the volume rendering algorithm because this increases the degree of irregularity. For up to 20 threads only a few of our new task pool implementations can achieve a slightly better speedup than the original SPLASH-2 implementation. The best task pool using block-distributed task queues with the ticket lock for mutual exclusion can achieve a speedup of 12.5 for 20 threads. The SPLASH-2 implementation reaches a speedup of 18.75 for 24 (and 32) threads. Although with more than 20 threads all of our task pool implementations are slower than the specialized load balancing method of the original SPLASH-2 implementation, the task pools using hardware operations for synchronization are 66 % faster than the best Pthreads implementation. Even the central task pool using hardware operations for the ticket lock is faster than the central and distributed Pthreads task pool due to the reduced overhead.

The bad performance measured for higher thread numbers can be explained by the highly irregular computational structure of the adaptive volume rendering algorithm and the small task granularity. When a thread finishes the work initially assigned to it, it searches cyclically for a thread with available tasks and continues the computation with the task from the selected thread until all tasks are finished. The

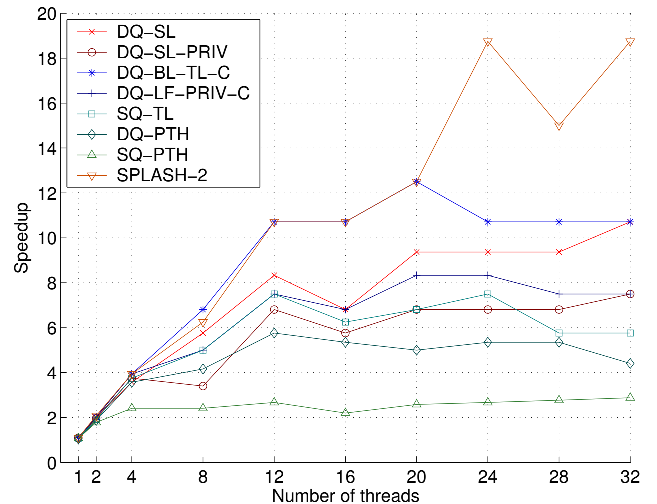

Figure 10: Speedup of the volume rendering application on the IBM p690 for selected task pools for the scene *head*.

SPLASH-2 implementation uses a simple counter for each thread which is protected by a Pthreads lock. The number directly addresses the next image tile to be computed. The task pools have a much larger overhead because in addition to the synchronization operations required the task pool has to search for available work which requires task stealing. For each image tile the SPLASH-2 implementation only has to increase a counter protected by a lock while in the task pool implementations each image tile is represented by a task which needs to be searched for and possibly stolen. The results show that the block-oriented task pools perform better because of the reduced overhead for task stealing. The task pools steal a group of tasks at once and therefore need to search for tasks less often. For the same reason the block-distributed task pools are the only implementations which can achieve a better speedup than the SPLASH-2 implementation for up to 20 threads. So the overhead introduced by the task pools is the major reason for the bad performance for higher thread numbers.

This is confirmed by runtime experiments without adaptive refinement which show that our task pool implementations can outperform the original SPLASH-2 implementation if the task granularity is sufficiently high (see Table 2). For example, the block-distributed task pool using hardware operations to implement the simple lock achieves a speedup of 29.11 for the scene *head* whereas the original SPLASH-2 program only reaches a speedup of 26.12. The best Pthreads task pool can only achieve a speedup of 18.19 even for this larger task size.

The reason for the poor performance when adaptive refinement is used is the very small task size. The number of tasks created is 35344 for the scene *head*. Using the sequential execution time of the reference task pool used for the speedup calculation, the average computation time of a task is $\approx 2\,\mu s$ on the IBM system. For such small tasks the overhead introduced by the general-purpose task pools is too large to improve the performance of the application. The special-purpose load balancing strategy implemented in the original SPLASH-2 code can handle such small tasks better.

We also examine the differences in the runtime of the sequential execution to observe the impact of the use of assembler language. Considering the average execution time of several program runs, the best assembler task pool implementation is only 0.6 % faster than the best Pthreads task pool. However, the variation of the execution time of the different program runs is 0.9 %. With respect to the better performance of the original SPLASH-2 implementation with more than 20 threads, the major performance improvement of our new implementations compared with the Pthreads implementation results from the use of the hardware synchronization operations. The best new task pool implementation (block-distributed task pool using the ticket lock) is $\approx 66\,\%$ faster than the best implementation using Pthreads synchronization operations (block-distributed task pool using Pthreads locks).

Even though the input data of the volume renderer are read-only and only a small synchronization overhead is required to perform the computations, in general the speedups measured for this application are relatively low because the granularity of the tasks is very small. Therefore, the specialized load balancing method provided by the SPLASH-2 implementation of the volume renderer that is based on a simple counter can provide a better performance than our general-purpose task pool implementations for high numbers of processors. However, the experiments with this application, which pose a particular challenge for a general-purpose task pool, show a significantly better performance of the task pools using hardware operations compared with all Pthreads-based implementations. In contrast to parallel quicksort, block-oriented task pools perform relatively well since they increase the granularity of the work units exchanged between the processors and thus reduce the synchronization overhead.

5. RAY TRACING

Ray tracing is a global illumination method that renders three-dimensional geometric scenes to a two-dimensional image. To compute the color of the image pixels, primary rays are fired from a viewpoint, through the pixels in the image plane, and into the object space. For each ray, the intersection with an object surface closest to the image plane is computed. At the intersection points, the rays are reflected toward every light source to compute the contribution of the light sources to the color of the rays. The same procedure is applied recursively to the reflected rays. The rays are terminated when they leave the enclosing volume or when a user-defined criterion is satisfied (e.g., the maximum number of reflection levels).

In our evaluation, we use a parallel implementation that results from a modification of the implementation provided by the SPLASH-2 application suite [43] (*raytrace* application [35]). Here, an efficient traversal of the scene data is realized using a hierarchical uniform grid, and early ray tracing and adaptive sampling are implemented. The parallelization exploits the parallelism across rays. Similar to the *volrend* application, the image is divided into rectangular blocks which are processed by the participating processors using a task-based execution scheme for load balancing. Load balancing is required since the work connected to each primary ray is unpredictable as it depends on the overall number of reflected rays that must be followed to compute the color of the corresponding image pixel. Since the geometric data is read-only and the processors work on disjoint image tiles, no synchronization is required on the application data. Thus, an efficient parallel execution of the ray tracing application is possible.

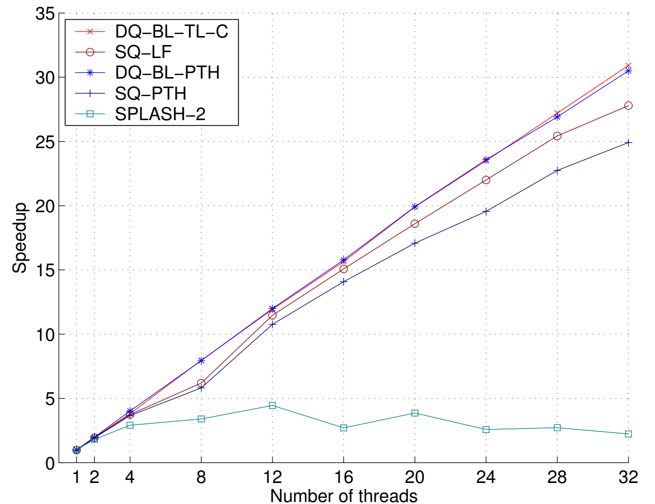

Figure 11: Speedup of the modified ray tracing application for selected task pools on the IBM p690 for the scene *car*.

We modified the original SPLASH-2 code by removing an unused ray identification number protected by a lock, because this lock significantly reduced the performance in preliminary experiments. For our runtime experiments on the Power4 system with the modified version we render the scene *car* using a resolution of 512 by 512 pixels, see Figure 11. The figure shows that all of our task pool implementations perform very well. The slowest implementation which uses a central task pool and Pthreads synchronization operations already achieves a speedup of 24.92. The best task pool implementation with a block-oriented task distribution protected by a ticket lock (DQ-BL-TL-C) can even achieve an almost ideal speedup of 30.92. The corresponding block-distributed task pool using Pthreads operations is only slightly slower (1.3 %) reaching a speedup of 30.5. In contrast, the original SPLASH-2 implementation can hardly profit from the modifications we made. The maximum speedup improves from 2.37 for 4 threads in the original version to 4.45 for 12 threads in our modified version. The load balancing strategy used in the original SPLASH-2 implementation also realizes a task-based execution. But its scalability is limited mainly because the memory management used for task allocation is based on a central free-list protected by mutual exclusion.

Table 1 shows the speedups of all implementations included in the experiments with the ray tracing application. For this application it is easy to select the best task pool. Distributed task pools perform better than central task pools and block-distributed task pool implementation are even slightly faster. The best task pool is the block-distributed task pool which uses the ticket lock to protect the task queues. Although task pools using hardware operations to implement mutual exclusion provide a better performance than the corresponding Pthreads implementation, the actual performance gain is only marginal.

Implementation	Speedup
SPLASH-2	4.45
SQ-PTH	24.92
SQ-TL	27.82
SQ-LF	27.79
SQ-SL	26.08
DQ-PTH	28.66
DQ-SL	29.35
DQ-SL-PRIV	29.88
DQ-SL-PRIV-C	29.42
DQ-SL-TRY-PRIV-C	29.32
DQ-TL-PRIV-C	29.86
DQ-SL-SP-PRIV-C	29.57
DQ-CLH-PRIV-C	29.64
DQ-LF-PRIV-C	27.18
DQ-BL-PTH	30.50
DQ-BL-SL-C	30.52
DQ-BL-SL-TRY-C	30.14
DQ-BL-TL-C	**30.92**
DQ-BL-TL-TRY-C	30.77
DQ-BL-SP-C	30.82
DQ-BL-SP-TRY-C	30.48
DQ-BL-CLH-C	30.23

Table 1: Speedup of all task pool implementations for the ray tracing scene *car*.

Our task pool implementations perform much better for the ray tracing application than for the volume rendering application, because the computational work connected to a task in the ray tracing implementation is much higher. The number of tasks created for the scene *car* using a resolution of 512 by 512 pixels is 16384. Using the execution time of the reference task pool used for the speedup calculation we can determine the average computation time of a task to be $\approx 420\,\mu s$. So the task size is more than 200 times higher for the ray tracing application than for the volume renderer (Table 2). Therefore, the impact of the task pool overhead is much smaller and our implementations can exploit the parallelism provided by the ray tracing algorithm.

For the ray tracing application we observe that the use of hardware synchronization operations cannot significantly improve the performance. Because of the larger task size, the Pthreads task pool already achieves an almost ideal speedup and the improvement of $\approx 1.4\,\%$ of the best new task pool is only marginal. As for the other applications, the use of assembler language for the new task pools does not influence the runtime significantly. The sequential runtime difference between the best assembler task pool and the best Pthreads task pool is almost 0.5 % while the runtime variation for multiple runs is about 3 %.

6. HIERARCHICAL RADIOSITY

The hierarchical radiosity method [14] is a global illumination algorithm that computes radiosity values for a given geometric scene by computing the equilibrium distribution of light. The hierarchical method performs an adaptive hierarchical subdivision of the object surfaces induced by the error of the radiosity computation.

As for the volume renderer and the ray tracer, we use the parallel implementation provided by the SPLASH-2 suite [43] (*radiosity* application [35, 36]) in our comparison. The *radiosity* application can be considered as a non-deterministic application because different execution orders of the tasks can be observed to produce a slightly different number of surface elements, and, as a consequence, there is a minor variation in the number of interactions processed. To minimize these effects, we use statistical information combined with the execution time to assess a particular run of the application. For example, the total number of interactions processed per second or the number of visible interactions processed per second allow a good comparison. The advantage of this approach is 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. For all runtime experiments we use a builtin scene called *largeroom* to measure the speedups with respect to the sequential execution time of a reference task pool which does not use synchronization.

Our experiments on the Power4 system lead to the following observations. Considering the central task pools, the task pools which utilize LR/SC are significantly faster than task pools based on Pthreads synchronization. The SQ-SL implementation is slower than the Pthreads task pool (SQ-PTH) with more than 16 processors, but the SQ-SL task pool does not contain any of the further optimizations. The Pthreads task pool, SQ-PTH, reaches a speedup of 9.14 whereas the task pool using the ticket lock achieves a maximum speedup of 11.25. The lock-free task pool performs best achieving a speedup of 15.84. The scalability of the original SPLASH-2 implementation is limited. It achieves a speedup of 11.42 for 24 threads. The SPLASH-2 implementation uses a distributed work queue for load balancing, and therefore a better performance than for the central task pools can be expected. But for 32 threads (cf. Figure 12) the original implementation is even slightly slower than the central Pthreads task pool. Due to the low overhead of the lock-free task pool using hardware operations, this implementation can outperform the original SPLASH-2 version significantly.

All distributed task pools using hardware operations are significantly faster than the distributed Pthreads task pool, which achieves a maximum speedup of 12.08. While some implementations like the ticket lock task pool or the cache-optimized simple lock task pool are only slightly better, other task pools can achieve higher speedups. The distributed task pool with the SINGLEPUT extension and the task pool using the single-try lock even reach a speedup of 18.13 and 18.97, respectively. The best implementation is the non-cache-optimized task pool using distributed queues and the implementation of the simple lock with hardware operations (DQ-SL-PRIV). The speedup achieved is 19.98. Although the original SPLASH-2 implementation also uses distributed queues for load balancing, it is slower than all other distributed task pools investigated.

For the block-distributed task pools we observe a similar situation. The Pthreads task pool achieves a maximum speedup of 12.4. Thus, the best task pool that uses hardware operations is 1.6 times faster than the best Pthreads task pool. The task pool implementing the SINGLEPUT extension (DQ-BL-SP-C) and the task pool using the simple lock (DQ-BL-SL-C) are only slightly better. The seminonblocking task pool implementing the single-try lock (DQ-

	Volume Rendering (adaptive) scene *head*	Volume Rendering (non adaptive) scene *head*	Ray Tracing scene *car*	Radiosity scene *large-room*	Quicksort array size of 100,000,000
execution time	75 ms	204 ms	6876 ms	14931 ms	16502 ms
number of tasks	35344	35344	16384	177881	150894
average task size	$\approx 2\,\mu s$	$\approx 5.8\,\mu s$	$\approx 420\,\mu s$	$\approx 84\,\mu s$	$\approx 109\,\mu s$
maximum speedup SPLASH-2	18.75	26.12	4.45	11.42	N/A
maximum speedup of task pools using Pthreads	7.50	18.19	30.50	12.40	8.17
maximum speedup of task pools using HW-Ops	12.50	29.11	30.92	19.98	8.23

Table 2: Speedups and task sizes of the investigated applications.

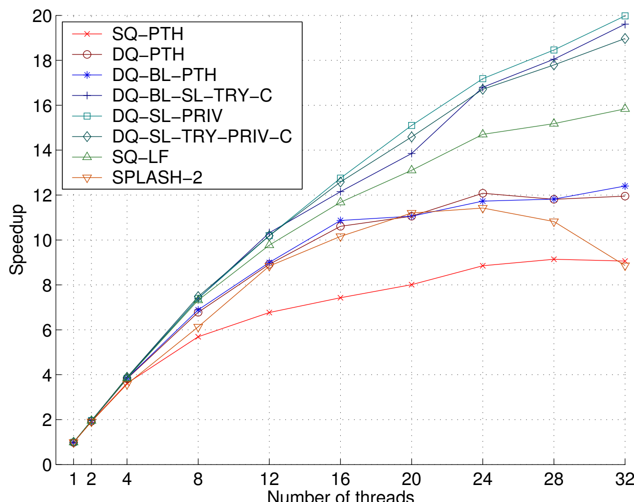

Figure 12: Speedup of the radiosity application for the scene *largeroom* on the IBM p690 system.

BL-SL-TRY-C) outperforms the other implementations and reaches a speedup of 19.61.

Figure 12 presents a summary of the best task pools from each category. The distributed Pthreads task pools are faster than the central task pools. The maximum speedup of the best Pthreads task pool is 12.4 whereas the best implementation using hardware operations achieves a speedup of 19.98. We also measured a good performance for the lock-free central task pool. The central task pool implementation outperforms even the distributed Pthreads task pools with a speedup of 15.84. The SPLASH-2 implementation achieves a speedup of 11.42. The best task pools from each category of task pools considered are implementations which use hardware operations. These task pools perform better than the original SPLASH-2 implementation. Considering the Pthreads task pool implementations, the central task pool (speedup 9.14) is the only task pool which is slower than the original SPLASH-2 implementation.

For the radiosity application using the best new task pool improves the performance by 61 % compared with the best Pthreads task pool. The reason for these large performance improvements observed from the experiments is the use of the hardware operations which reduces the synchronization overhead. This can again be shown by examining the sequential runtimes. The difference between the best Pthreads task pool and the best assembler task pool is only 0.3 % while the variation of the runtimes of multiple runs is again greater with almost 0.6 %.

Our runtime experiments with the radiosity application on the SPARC system show different results. Using central task pools, the implementations with hardware synchronization can outperform the Pthreads implementation. While the Pthreads task pool only achieves a speedup of 7.09, the best hardware-based implementation (using the ticket lock) can achieve a speedup of 9.26. The other lock implementations are slower but still clearly faster than the Pthreads implementation. For the distributed task pools and the block-distributed task pools the hardware-based implementations are not faster than the Pthreads implementation. The best Pthreads implementation is the distributed task pool achieving a speedup of 10.5 while the ticket lock implementation is 2.5 % slower (speedup of 10.24). Figure 13 shows the best task pool implementations on the SPARC system. The central Pthreads task pool is clearly the slowest. The central task pool using the ticket lock is much faster and can almost achieve the same speedup as the distributed task pools for up to 12 processors. This result is similar to the Power4 system where the lock free central task pool can nearly keep up with the distributed task pool implementations. The runtime differences between the distributed task pools are small, so the limiting factor is not the task pool for this application on the SPARC system.

7. 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 [22]. Static scheduling is based on the assumption 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. For the case that task execution times cannot be modeled exactly, Tongsima *et al.* have proposed modeling the execution times by probabilistic distributions [40] or fuzzy sets [8]. In special cases, static scheduling can be

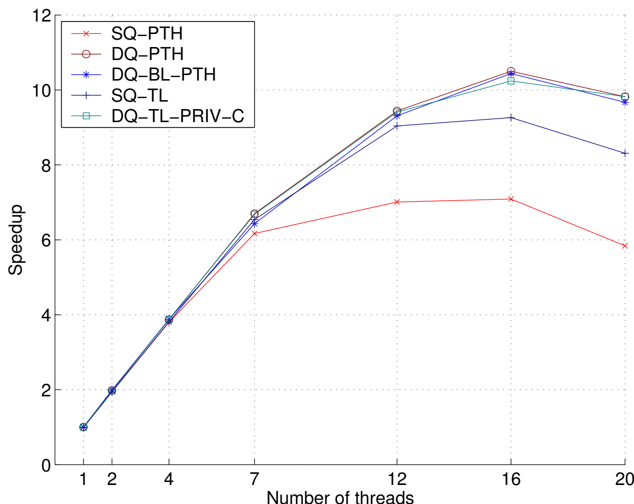

Figure 13: Speedup of the radiosity application for the scene *largeroom* on the SPARC system.

used to speed up irregular applications. Gerasoulis *et al.* [13] have applied the static scheduling system PYRROS [44] to the Fast Multipole Method (FMM).

In general, *dynamic* scheduling is necessary to exploit parallelism in irregular applications efficiently. Dynamic techniques aiming at distributing work equally among several processors are often also referred to as *load balancing* algorithms. Kumar *et al.* [21] compare several load balancing techniques. Osman and Ammar [30] propose a general taxonomy for describing and classifying different load balancing techniques.

Various approaches have been proposed towards the efficient parallel execution of irregular computations: Johnson [18] proposes a Dynamic Task Graph (DTG) used to store tasks created at runtime. Cosnard *et al.* [10] propose a Symbolic Linear Clustering (SLC) algorithm for Parameterized Task Graphs (PTGs). Further approaches concentrate on automatic loop scheduling, e.g., [7, 9, 33]. Schloegel *et al.* [34] consider the parallel execution of computations on irregular adaptive grids where a periodic re-partitioning of the grid is required to minimize inter-processor communication and to balance the load; this is, e.g., supported by the Unified Repartitioning Algorithm (URA). Dandamudi and Ayachi [11] present a processor scheduling scheme based on a hierarchical run queue organization. Podehl *et al.* [31] have used a parallel task queue implemented by multiprefix operations to improve the performance of the hierarchical radiosity method on the SB-PRAM. Thomé *et al.* [39] investigate different load balancing strategies for the computation of macroscopic thermal dispersion in porous media. Hippold and Rünger [16] investigate task pools that are distributed among the address spaces and load balancing is achieved by exchanging tasks using special communication threads.

There exist several programming environments dedicated to provide dynamic load balancing in irregular applications. Charm++ [19] extends the C++ language providing load balancing based on Adaptive Contracting within Neighborhood (ACWN). DOTS [3] and Cilk [32] provide a fork-join model for multithreaded computations, where load balancing is achieved by centralized methods or work stealing, respectively. VDS [12] combines several programming paradigms such as message passing, independent tasks, multithreaded computations, and DAG scheduling.

Mechanisms for the synchronization of threads on shared-memory systems have been studied extensively in the past. An overview of several lock-based algorithms for mutual exclusion can be found in [1, 27]. To avoid performance degradation due to inopportune preemption on multiprogrammed systems, non-blocking (also called lock-free) data structures have been proposed, see for example [15, 28]. Herlihy *et al.* [25] introduce obstruction-freedom as a property that is weaker than lock-freedom and wait-freedom but allows greater flexibility in the design of efficient implementations. Bush *et al.* [5] provide a theoretical analysis of the costs of atomic read-modify-write operations, which are frequently used in the implementation of synchronization mechanisms.

8. SUMMARY AND CONCLUSIONS

We have evaluated the performance of different task pool implementations for shared-memory computer systems based on hardware synchronization using several realistic applications. Our runtime experiments show that the task-based execution of irregular applications can often achieve a reasonable performance. In many of our experiments, the general approach of using task pools for load balancing leads to a higher performance than the original, application-specific load balancing strategies employed. Compared with implementations based on Pthreads synchronization, the new implementations which utilize special hardware operations often lead to a lower overhead and thus can reduce the execution time and increase the scalability significantly.

The results presented in this paper show that the selection of a suitable task pool is crucial for the performance of an irregular application. However, the investigation of the different applications presented also shows that the performance is influenced by the task pool implementation and the structure of the application as well as by the underlying hardware architecture. The task size as shown in Table 2 is a major criterion for the task pool selection.

For very small tasks as in the volume rendering application, the task pool overhead largely influences the execution time. The reduced overhead due to the use of hardware synchronization operations greatly improves the performance in contrast to the Pthreads implementation (factor 1.67). But for the volume rendering application, the general-purpose load balancing strategy realized by our task pools usually cannot outperform the specialized load balancing strategy used in the SPLASH-2 implementation. Task grouping as applied by the block-distributed task pools partly solves this problem so this type of task pool performs best. For up to 20 threads, the block-distributed task pool using the ticket lock can even slightly outperform the specialized load balancing strategy of the SPLASH-2 implementation.

On the other hand, for very large tasks as in the ray tracing application the influence of the task pool overhead is much smaller. However, the type of the task pool is still important. For this application the best task pool (block-distributed task pool using the ticket lock) is $\approx 24\,\%$ faster than the slowest task pool investigated (central task pool using Pthreads operations). Distributed task pools perform better than the central task pools although the synchronization overhead is small due to the large tasks. The best hardware operation task pool is $\approx 4.2\,\%$ faster than the

Pthreads distributed task pool. The block-distributed task pools using hardware synchronization operations perform best but are only 1.3 % faster than the corresponding implementations using Pthreads operations. Though the block-distributed task pools reduce the overhead of the synchronization operations, the actual implementation of the synchronization is not as important as for smaller tasks. The difference between the best block-distributed task pool (using the ticket lock) and the worst block-distributed task pool (using the single try simple lock) is only 2.5 %.

For medium task sizes the selection of the task pool is also very important to achieve a good speedup. Using the same synchronization operations, distributed task pools are faster than central task pools. But for the radiosity application, one central task pool using hardware operations can even outperform the distributed task pools using Pthreads operations. This shows that distributed task pools are not necessarily faster than central task pools but in contrast to the situation for larger tasks, the synchronization operations used are very important. For the radiosity application, the simple lock works best. The block-distributed task pools are not the fastest, but they are only slightly slower (about 2 %) than the distributed task pools. Although the task pools using Pthreads operations are already 8.6 % faster than the SPLASH-2 implementation, the use of hardware synchronization operations improves the performance by 61 % compared with the Pthreads task pools.

A different type of irregular application, the quicksort application, shows that, in contrast to the previous results, the block-distributed task pools can be much slower than distributed or even central task pools. Due to the task grouping applied, block-distributed task pools cannot utilize the degree of parallelism offered by the quicksort application. Because the task size is large and so the overhead of the task pool is small, the differences between different synchronization methods are only marginal. The ticket lock again performs best but is only $\approx 1\,\%$ faster than the best Pthreads task pool.

Without knowledge about the application the best task pool implementation cannot be determined. However, distributed task pools show a good performance for all applications we have investigated. For applications where the initial work distribution already provides a high degree of parallelism, the block-distributed task pools perform very well especially for fine-grained tasks.

We have used several different synchronization methods to realize mutual exclusion. The smaller the task size, the more important is the method used for synchronization. To achieve the highest speedup possible, the use of hardware synchronization operations is very important. The lock free synchronization method performs well but either the ticket lock or the simple lock achieve the best performance for the applications investigated.

For each of the different applications considered, we have shown that switching to assembler language for the task pool implementations does not significantly influence the runtime of the applications utilizing our task pools. The performance gain of the parallel execution originates from the highly reduced synchronization overhead due to the use of hardware synchronization operations offered by the investigated systems.

The new task pool implementations presented in this paper utilize the *Load & Reserve/Store Conditional* operations for synchronization, which are provided by, for example, the Power, the Alpha, and the MIPS microprocessors. We also presented implementations using the *Compare & Swap* operations provided by, for example, the UltraSPARC3 processor. But the load balancing strategies and synchronization methods presented can be adapted to many other systems since similar operations are provided by most modern microprocessors.

9. REFERENCES

[1] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–100, 2003. Special issue celebrating the 20th anniversary of PODC.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, Nov. 2000.

[3] W. Blochinger, W. Küchlin, and A. Weber. The distributed object-oriented threads system DOTS. In A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, editors, *Fifth Intl. Symp. on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*, number 1457 in LNCS, pages 206–217, Berkeley, CA, U.S.A., Aug. 1998. Springer.

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

[5] C. Busch, M. Mavronicolas, and P. Spirakis. The cost of concurrent, low-contention read-modify-write. In *Proceedings of the 10th Colloquium on Structural Information and Communication Complexity (SIROCCO 2003)*, Umeå, Sweden, 2003.

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

[7] R. L. Cariño and I. Banicescu. A load balancing tool for distributed parallel loops. In *Proceedings of the International Workshop on Challenges of Large Application in Distributed Environments (CLADE) 2003*, pages 39–46. IEEE Computer Society Press, June 2003.

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

[9] A. T. Chronopoulos, S. Penmatsa, and N. Yu. Scalable loop self-scheduling schemes for heterogeneous clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02)*, page 353, Nov. 2002.

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

[11] 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.

[12] T. Decker. Virtual data space – load balancing for irregular applications. *Parallel Computing*, 26(13–14):1825–1860, Dec. 2000.

[13] 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*, Apr. 1995.

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

[15] M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking data structures. In *CATS*, 2003.

[16] J. Hippold and G. Rünger. Task pool teams for implementing irregular algorithms on clusters of SMPs. In *Proc. of IPDPS*, Nice, France, 2003. CD-ROM.

[17] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(4):10–15, 1962.

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

[19] L. V. Kale and S. Krishnan. CHARM++. In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, chapter 5, pages 175–214. MIT Press, Cambridge, MA, 1996.

[20] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16:1–47, Jan. 2004.

[21] 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.

[22] 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.

[23] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–251, July 1990.

[24] M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, Feb. 1990.

[25] V. Luchangco, M. Moir, and N. Shavit. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[26] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. `http://www.cs.rochester.edu/u/scott/synchronization/pseudocode/ss.html`.

[27] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[28] 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.

[29] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory MIMD architectures. In *Proceedings of the Boston Workshop on Volume Visualization*, pages 17–24. ACM Press, Oct. 1992.

[30] A. Osman and H. Ammar. Dynamic load balancing strategies for parallel computers. In *International Symposium on Parallel and Distributed Computing (ISPDC)*, July 2002.

[31] 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.

[32] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[33] L. Rauchwerger. Run-time parallelization: It's time has come. *Journal of on Parallel Computing, Special Issue on Languages & Compilers for Parallel Computers*, 24(3–4):527–556, 1998.

[34] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing 2000*, 2000.

[35] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, July 1994.

[36] 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, June 1995.

[37] SPARC International, Inc. *The SPARC Architecture Manual Version 9*, 2000.

[38] J. M. Tendler, S. Dodson, S. Fields, H. Lee, and B. Sinharoy. *POWER4 System Microarchitecture*. IBM Server Group, Oct. 2001.

[39] V. Thomé, D. Vianna, R. Costa, A. Plastino, and O. da Silveira Filho. Exploring load balancing in a scientific SPMD application. In *Proceedings of the 2002 ICPP Workshops*, pages 419–426, Aug. 2002.

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

[41] V.-Y. Vee and W.-J. Hsu. A Scalable and Efficient Storage Allocator on Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 230–235, Fremantle, Australia, 1999.

[42] K.-P. Vo. Vmalloc: A general and efficient memory allocator. *Software Practice & Experience*, 26:1–18, 1996.

[43] 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.

[44] 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., July 1992.