Int J Parallel Prog (2011) 39:553–581
DOI 10.1007/s10766-010-0156-z

Adaptive Task Pools: Efficiently Balancing Large Number of Tasks on Shared-address Spaces

Ralf Hoffmann · Thomas Rauber

Received: 26 July 2010 / Accepted: 10 November 2010 / Published online: 26 November 2010

Abstract Task based approaches with dynamic load balancing are well suited to exploit parallelism in irregular applications. For such applications, the execution time of tasks can often not be predicted due to input dependencies. Therefore, a static task assignment to execution resources usually does not lead to the best performance. Moreover, a dynamic load balancing is also beneficial for heterogeneous execution environments. In this article a new adaptive data structure is proposed for storing and balancing a large number of tasks, allowing an efficient and flexible task management. Dynamically adjusted blocks of tasks can be moved between execution resources, enabling an efficient load balancing with low overhead, which is independent of the actual number of tasks stored. We have integrated the new approach into a runtime system for the execution of task-based applications for shared address spaces. Runtime experiments with several irregular applications with different execution schemes show that the new adaptive runtime system leads to good performance also in such situations where other approaches fail to achieve comparable results.

Keywords Task-based execution · Dynamic load balancing · Adaptive load distribution · Task pool

1 Introduction

The growing number of cores in current and future processors requires a large amount of parallelism to fully utilize the available performance. At the same time, high

R. Hoffmann (✉) · T. Rauber
Department of Computer Science, University of Bayreuth, Bayreuth, Germany
e-mail: ralf.hoffmann@uni-bayreuth.de

T. Rauber
e-mail: rauber@uni-bayreuth.de

bandwidth and low latency connections between the cores opens the opportunity to use a large number of small tasks. Especially for heterogeneous cores with different processing power, it is difficult to determine a good work partition statically so that all cores are equally loaded. A dynamic load balancing approach is well suited to handle this scenario for regular and irregular applications.

For regular, loop-based applications, dynamic loop scheduling methods like guided self scheduling [23], factoring or adaptive factoring [4,9], and fractiling [3] can be used and often lead to good performance, even if different loop iterations require a different amount of execution time.

For irregular applications, task-based load balancing methods can often be used successfully [27]. Using this approach, the application is decomposed into tasks, which are dynamically assigned to idle execution resources (cores) until the execution of all tasks is finished. Task creation can happen at program start, but tasks may also be created dynamically during the execution of other tasks. Usually there are many more tasks available than there are execution resources. Tasks that are ready for execution are stored in a specific task data structure from which they can be retrieved by idle cores for execution. Since multiple cores may try to retrieve a task at the same time, synchronization has to be used to avoid race conditions. Many different approaches have been proposed for the organization of the data structure to store the tasks. Similarly, many different strategies for balancing task execution have been considered, including sender-based techniques like task pushing [31] and receiver-based techniques like task stealing [8,14].

Using a central data structure to store executable tasks may lead to waiting times because of synchronization, especially if the number of execution resources is large or if the granularity of the tasks is small, thus requiring a frequent access. To reduce such waiting time, distributed data structures are often used to store the tasks, where each execution resource has its own part of the data structure, which it accesses as long as there are tasks available in this data structure. To ensure load balancing, task stealing is often used [7,22], i.e., a core whose local task data structure is empty steals tasks from the data structure of neighboring cores. This approach still requires the use of synchronization, but the retrieve accesses are usually distributed quite evenly among the different parts of the data structure, thus avoiding large waiting times [27].

Efficiency problems may arise, if the tasks stored in the data structure are too fine-grained. In this case, frequent retrieve accesses are necessary and may lead to a significant overhead, especially if many steal operations have to be performed. On the other hand, using fine-grained tasks may have the advantage that a good load balancing is possible even if not much computational work remains. In this context, we propose a new distributed data structure for storing tasks along with an efficient organization of task migration between different parts of the data structure, based on our previous work [19]. In particular, the new organization enables an efficient task migration by grouping the tasks into blocks for the stealing operation and by dynamically adjusting the size of the blocks that are moved. In particular, the block size is dynamically adapted to the number of tasks available in the data structure from which the tasks are stolen. Using a fixed block size could lead to situations where only a few tasks are remaining and are stored in a single block, which could be the victim of a steal

operation. We present an implementation of the new approach and an integration into a runtime system for shared address space architectures.

Experiments with several large irregular applications on different platforms show that the overhead for task stealing and other related operations can be significantly reduced by using the adaptive task stealing approach proposed in this paper.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the adaptive data structure and the corresponding task stealing technique and Sect. 4 describes the implementation within the KOALA runtime environment. Section 5 presents an experimental evaluation based on synthetic benchmarks as well as on realistic applications. Section 6 concludes the paper.

2 Related Work

Dynamic load balancing is provided for parallel programming in the form of libraries or as part of programming languages. Charm++ [20] is based on C++ and offers load balancing by using object migration. Task parallelism is also used in Cilk [6]. The Intel Threading Building Blocks (TBB) [25] is a C++ library, which allows the use of different programming paradigms, including task parallelism and data parallelism (parallel loops). TBB uses hardware operations to implement synchronization with low overhead, but it is currently available only for a limited number of architectures. Similar approaches are used in languages like Fortress [2] or X10 [10], which support both task parallelism and data parallelism. The language Chapel [11] also offers data parallelism but no explicit task parallelism. Efficient dynamic load balancing for distributed memory systems can be obtained based on a partitioned global address space (PGAS) [12].

Since it is expected that the number of cores on a single processor chip increases further in future architectures, task based execution is a good method to utilize the available processing power efficiently. For example, the Cell architecture is well suited for task based execution [24] and CellSs [5] proposes a programming environment using annotations similar to OpenMP to schedule tasks to the available processing cores (SPEs). In Hardware accelerated support for dynamic task scheduling can be used to achieve better performance for fine-grained tasks [21].

However, the computational size of tasks in irregular applications is often hard to predict, so effective load balancing is important. Methods which adapt to the actual amount of computations are considered to be favorable [32]. Dynamic loop scheduling with adaptive techniques has been shown to be effective for applications from scientific computing [4]. An adaptive scheduler which uses runtime information to dynamically adapt the number of processors used to execute a job has also been investigated [1].

Data structures which adapt the amount of work stolen for load balancing have also been proposed [16]. This method enables the stealing of about half of the tasks from a given queue, but the approach needs dynamically executed balancing steps. The use of adaptive grids for irregular computations has also been investigated [26], but this approach requires periodic repartition steps to balance the available computations. The task data structure proposed in this paper does not need rebalancing steps and has a lower limit for the number of task stolen, but it also takes more time to insert

and remove tasks. Adaptive methods can be used to limit the number of tasks created at runtime, which is well suited for some irregular applications but requires tests for every new task created [13].

Work stealing can also be applied for a distributed address space [12,17]. Important techniques to obtain good scalability are the use of split task queues and in-place task creation. For a distributed address space, task granularity cannot be too fine-grained because of the use of communication for task exchange between executing cores.

3 Adaptive Task Pool Organization

In this section, we discuss the dynamic load balancing and task execution model considered in this article. In particular, we describe the adaptive data structure (called *task pool* in the following) that is used to stored executable tasks and the actual runtime system that performs a task-stealing based load balancing using this data structure.

3.1 General Approach

Load balancing strategies for tasks and the corresponding data structures used to manage the tasks may differ in multiple aspects. Central data structures like a single linked list may be used, so there is no explicit load balancing required. Task are available for every thread accessing this data structure. However, using fine-grained tasks or a large number of threads leads to a bottleneck due to a high synchronization overhead. Distributed data structures can be used to solve this problem by using multiple data structures. For such distributed task pools one task data structures per thread is often used. Therefore, many threads can access the task pool without hindering other threads. A good load balancing strategy is required so that all threads can be kept busy. That usually means accessing task data structures from different threads or migrating tasks between empty and non-empty data structures. The former may introduce contention while the latter introduces an additional overhead.

Without task migration there might be situations where many threads access a single task data structure so it may basically become a central access point. Therefore, it is important to include an efficient task migration method like task stealing. This is especially important for applications which create a large number of tasks during runtime.

Moving blocks of tasks instead of single tasks between task data structures may reduce the overhead for task stealing, since fetching an entire block enables a processor to execute multiple tasks before searching for tasks again. However, this reduces the number of stealing operations only by a constant factor, and using blocks that are too large may limit the available parallelism. A suitable value for the block size depends on the application and its runtime behavior but also on the actual architecture the application is executed on.

In this paper, we propose an alternative approach to reduce the overhead for task stealing. Instead of moving a fixed-size block of tasks between the task data structures, the new approach is able to adapt the size of the task block to be moved to the specific execution situation of an application program. This adaptation is supported

by a specific organization of the distributed task data structures. In each structure, the tasks are stored in a list of balanced trees of increasing size, where each list entry i can store a maximum number of balanced trees of depth i. After their creation, tasks are inserted by the creating thread into the first non-full list entry of its local task data structure. This may lead to the combination of several smaller trees to a larger tree. Each thread retrieves tasks for execution from the smallest tree of its local task data structure, i.e., from the first non-empty entry. This may lead to the splitting of a larger tree to several trees of smaller size. Thus, retrieving a task does not necessarily return the task that has last been inserted, so no LIFO order is implemented. The details will be explained in the following subsections.

For load balancing, a task stealing approach is used. Using the tree-based task data structures, tasks are stolen as complete trees. In particular, stealing the largest tree typically moves a large number of tasks to the task data structure of the stealing thread. Thus, this thread does not need to steal tasks in the near future. It can even be the victim of steal operations by other threads. Therefore, in case of a load imbalance, only a few steal operations are usually required to distribute the tasks quite evenly among the different task data structures. Since only one tree at a time is stolen, the task data structure of the victim thread still contains enough tasks for local execution. In the following, we describe the proposed data structure and the operations to insert, retrieve, and steal tasks in more detail.

3.2 Task Trees and Forest Vectors

Since the use of fine-grained tasks implies a frequent access to the task pool, insertion and removal operations need to be fast. Task stealing may be expensive due to synchronization and remote access overhead and, therefore, should support the removal of a large number of tasks by a single operation. Distributed queues implemented as linear lists are often used to store the tasks, and their performance can be improved by additionally using blocks of tasks for task stealing. Linked lists take constant time for insertion or removal of tasks, but each task has to be removed in a separate step. Using blocks of tasks allows the stealing of several task by one operation. But this approach may limit the exploitable degree of parallelism if there are only a few executable tasks available, which might be stored in a single or a small number of blocks on a single queue. In this case, although there are several tasks available, only one processor can execute them.

The data structure described in the following solves this problem by using adaptive blocks of tasks which grow or shrink with the number of tasks available. The tasks are stored in a set of fully balanced trees so that for a tree T_i the length of every path from the root to a leaf is i. Each node stores a single task as well as a pointer to the first child and a pointer to the next sibling in the same tree level. Figure 1 shows an example of trees for different depths.

The trees are stored in a forest vector $F[0 \ldots w]$ forming a forest of fully balanced trees, where w is the depth of the largest tree to be stored. Each entry $F[i]$ is a pointer to a list of trees T_i of depth i, i.e., $F[0]$ only stores fully balanced trees of tasks with one level, $F[1]$ stores trees with two levels, and so on (see Fig. 2). Without breaking

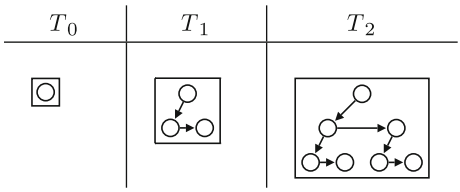

Fig. 1 Example tree structure of depth 0, 1, and 2

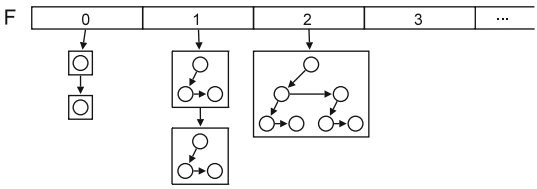

Fig. 2 Example of a partially filled forest vector

the balance requirement, new tasks may only be inserted in the first level $F[0]$, since a single task is already a tree of depth 0, or by combining existing trees of the same depth into a new tree with a larger depth.

The number of entries in each forest level is limited by a predefined value, so the trees can grow as new tasks are inserted. If the limit is reached no new tree can be inserted; instead a new and larger tree needs to be built. This length criterion can be used to control the growing of the individual trees. The limit can be any fixed number of children, and even dynamic limits can be used, e.g., based on the actual work described by each task. Using $l = 1$ leads to a sequence of linked lists, so in each level there is only one more task stored in addition to the next smaller level. This is not useful in practice, since the number of tasks that can be stored would be quite limited. Using values $l > 2$ is possible, but then the single groups in each entry become larger while the number of different levels used in the forest vector shrinks. So there might be less groups available for task stealing. We assume a fixed limit $l = 2$ for the following description of the insertion and removal operations, since this leads to good results in practice. This corresponds to the use of balanced binary trees.

3.3 Inserting New Tasks

New tasks are inserted into the forest vector based on the algorithm shown in Fig. 3. Regardless of the current state of the forest vector, a new tree of depth 0 is created storing the new task information. If the number of tasks in forest entry $F[0]$ is smaller than the length limit l, the new tree can be inserted by storing the current first entry in the sibling pointer and making the new tree the head element. Since this is basically a single linked list, the operation takes constant time.

If the first forest entry $F[0]$ is full (i.e., the length limit is reached), the new tree cannot be inserted directly but needs to be used to build a larger tree. Starting from 0, the first forest entry $F[i]$ is searched which can store another tree, i.e., the length of $F[i]$ is smaller than l. To create a valid fully balanced tree of depth i the new task becomes the root of all l entries of $F[i-1]$ by storing the head element of $F[i-1]$

Fig. 3 Algorithm to insert a new task into the forest vector

```
Create new tree t = new T_0 with:
      t → task = new task
      t → child = NULL
      t → sibling = NULL

if ( Number of trees in F_0 < l ) {
      t → sibling = F_0
      F_0 = t
} else {
      Search smallest i with Number of trees in F_i < l
      t → child = F_{i-1}
      F_{i-1} = NULL
      t → sibling = F_i
      F_i = t
}
```

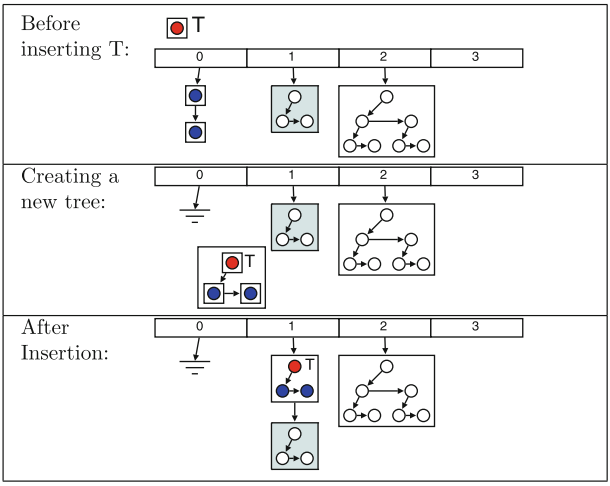

Fig. 4 Example for inserting a new task T

in the child pointer. The length of the path is increased by one and the tree is fully balanced. The new tree can be inserted into $F[i]$ by storing the current head element in the sibling pointer and using the tree as the new head element. After this insertion, $F[i-1]$ is empty.

Figure 4 gives an example for the insertion process. Since $F[0]$ is full (based on a limit $l = 2$), a new tree needs to be constructed to store the new task T. In this example, $F[1]$ only contains a single tree and can store another one. Therefore, all entries in $F[0]$ become children of the new task, forming a tree of depth 1. After this combining step, the new tree can be inserted into $F[1]$. The next two new tasks can be then inserted into $F[0]$ again, since $F[0]$ is empty now. After that, the next task would combine the entries in $F[1]$ into a tree of depth 2, and so on.

Fig. 5 Algorithm to remove a task from a forest vector F

```
Search smallest i with number of trees in F_i > 0
t = F_i
F_i = t → sibling
if ( i > 0 ) {
    F_{i-1} = t → child
}
return t → task
```

3.4 Removing a Task for Execution

The remove operation is almost (but not completely) the inverse operation of the insert operation. Tasks are always removed from the first non-empty forest entry. Since the insert operation may combine trees from larger entries, a removed task has not necessarily been inserted by the last insert operation. Figure 5 shows the corresponding algorithm for the remove operation.

The remove operation searches for the smallest entry i for which there is at least one tree in the corresponding list $F[i]$. The first tree is removed by a simple list remove operation. The root of the tree is the task to be executed next. If the forest entry i is larger than 0, the tree needs to be split. All subtrees of an entry from $F[i]$ are of depth $i-1$ and are therefore stored in $F[i-1]$, which is empty because $F[i]$ has been the first non-empty tree. Since the root node stores a pointer to the first child, the list of subtrees can be inserted by a single operation.

Figure 6 illustrates the removal of a task for a given forest vector. The smallest tree available is a tree of depth 2. The first tree in $F[2]$ will be removed and the root node contains the tasks executed next. The subtrees of depth 1 will be stored in $F[1]$.

Since searching for the first non-empty entry may take a significant portion of the execution time of the operation, the remove operation is designed to be fast instead of to be the exact inverse function of the insert operation. For example, if there are ten forest entries completely full, an insert operation will need eleven steps to find a place to store the new task. After the insertion and the corresponding inclusion of all trees from $F[10]$, $F[10]$ is empty. Implementing exactly the inverse operation would require to split the tree in $F[11]$ again. Instead, the presented algorithm always tries to access the first possible entry, so it is not a LIFO operation (Last In, First Out). The modified order may have an effect on the cache locality due to the fact that data for older tasks are less likely to be still in the cache. On the other hand, guaranteeing a LIFO order would have an runtime penalty for every operation. The data structure should be fast also for a large number of tasks stored, so the typical operation of task insertion and removal need to be fast. Since the number of tasks stored in the forest vector grows exponentially with each additional level, the time for the search operation still grows logarithmically.

3.5 Adaptive Task Stealing

The task data structure proposed in the previous subsections is only effective if more than one of these structures is used in the parallel task pool. The current runtime system

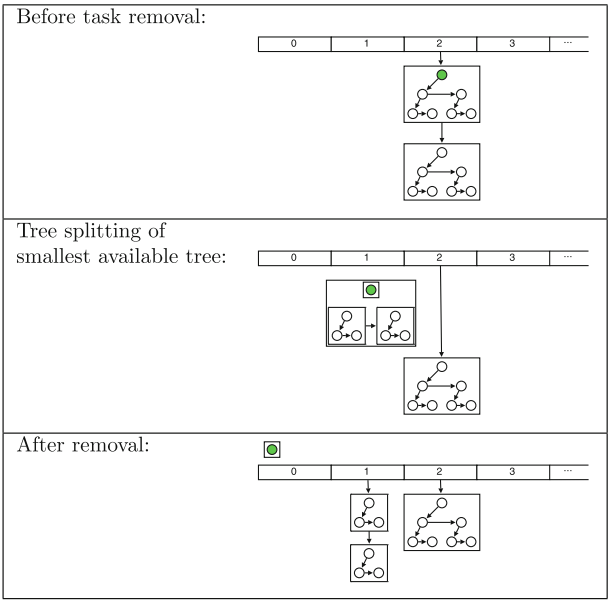

Fig. 6 Example for the removal of a task

Fig. 7 Algorithm for task stealing by thread p_1 of a tree owned by thread p_2

Search largest i with number of trees in $F_i(p_2) > 0$
Remove tree from p_2 :
$t = F_i(p_2)$
$F_i(p_2) = t \rightarrow sibling$

Insert tree in $F(p_1)$ and split tree:
if ($i > 0$) {
$F_{i-1}(p_1) = t \rightarrow child$
}

return $t \rightarrow task$

uses one data structure for each thread, so the threads can access their corresponding forest vector without conflicts if there are enough task available in each of the forest vectors.

Using distributed data structures requires to balance work between the threads whenever a forest vector becomes completely empty. Therefore, tasks or trees of tasks need to be moved between the data structures. The task stealing operation as described in Fig. 7 can take advantage of the growing trees to fill an empty data structure with a larger number of tasks in only few operations.

To reduce the number of stealing operations, and therefore the task stealing overhead, it is beneficial to steal the largest available task tree after having found a non-empty forest vector. A thread p_1 with an empty forest vector searches from the opposite end of the forest vector of another thread p_2 for available trees. A tree will be removed from the non-empty forest entry $F[i]$ with the largest task trees. This tree will be

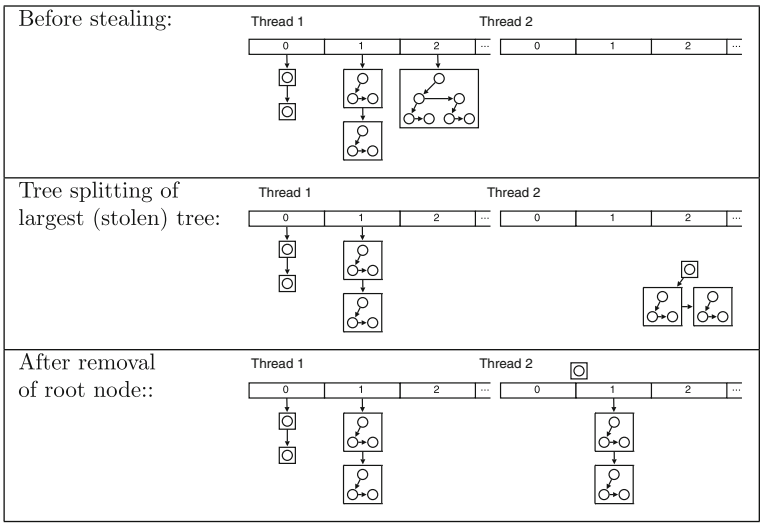

Fig. 8 Example for a stealing operation

removed from the data structure of p_2 and will be inserted into $F[i]$ of p_1. Since the task stealing operation needs to return a task to be executed next, this tree will be split into subtrees and the task in the root node will be returned for execution to p_1.

Figure 8 gives an example for a stealing operation for a typical situation. Thread 1 stores several trees of different depths in its forest vector while the forest vector of thread 2 is empty. For stealing, the largest tree in $F[2]$ will be removed from the forest vector of thread 1 and moved to the forest vector of thread 2. Then, the tree is split into two trees of depth 1, which are stored in the forest vector of thread 2 for later execution, and the root task is executed next. In this example, at the beginning there were 15 tasks stored in the forest vector of thread 1 and after the task stealing operation there are 8 left for thread 1 and 7 are available for execution by thread 2. Thus, in just a single step the load is almost balanced.

Due to the recombination of subtrees into larger trees, tasks of a given tree are usually related in the sense that they are created temporally close, so they might share some data that they access. Hence, stealing trees may give a locality advantage in contrast to stealing tasks one by one on demand.

3.6 Complexity Considerations

It is clear that the number of tasks that are stolen in a single operation depends on the size of the corresponding tree. In the following, we give an approximation for the actual number of tasks stored in a tree for a given tree degree limit l.

The proposed data structure uses completely balanced and filled trees with a fixed degree l. The number of tasks $t(i)$ stored in a tree of depth i can therefore calculated by

$$t(i) = \frac{l^{i+1} - 1}{l - 1}. \tag{1}$$

The number of tasks stored in a completely filled forest vector $F[0 \ldots w]$ up to the level w is

$$t_{sum} = \sum_{i=0}^{w} l \cdot t(i), \tag{2}$$

since in every entry i there are l trees of depth i. So the total number of tasks stored is

$$\begin{aligned}
t_{sum} &= \sum_{i=0}^{w} l \cdot \left(\frac{l^{i+1} - 1}{l - 1}\right) = \frac{l}{l-1} \sum_{i=0}^{w} \left(l^{i+1} - 1\right) \\
&= \frac{l}{l-1}\left(\sum_{i=0}^{w} l^{i+1} - \sum_{i=0}^{w} 1\right) = \frac{l}{l-1}\left(\sum_{i=0}^{w} l^{i+1} - (w+1)\right) \\
&= \frac{l}{l-1}\left(\sum_{i=0}^{w+1} l^{i} - 1 - (w+1)\right) = \frac{l}{l-1}\left(\frac{l^{w+2} - 1}{l-1} - 1 - (w+1)\right) \\
&= \frac{l}{l-1}\left(\frac{l^{w+2} - 1}{l-1} - \frac{l-1}{l-1} - \frac{(w+1)(l-1)}{l-1}\right) \\
&= \frac{l}{l-1}\left(\frac{l^{w+2} - (w+2)l + (w+1)}{l-1}\right) \\
&= \frac{l^{w+3} - (w+2)l^2 + (w+1)l}{(l-1)^2}.
\end{aligned} \tag{3}$$

The fraction f of tasks stolen in a single operation can be determined by

$$f = \frac{t(w)}{t_{sum}}, \tag{4}$$

since always the largest tree is removed (i.e., tree in $F[w]$). Using Eq. 3 gives

$$\begin{aligned}
f &= t(w) \cdot \frac{1}{t_{sum}} \\
f &= \frac{l^{w+1} - 1}{l - 1} \cdot \frac{(l-1)^2}{l^{w+3} - (w+2)l^2 + (w+1)l} \\
&= (l-1)\,\frac{l^{w+1} - 1}{l^{w+1}} \frac{1}{l^2 - \frac{(w+2)l^2 - (w+1)l}{l^{w+1}}}.
\end{aligned} \tag{5}$$

The largest value is $f = 1/l$ for $w = 0$ and the smallest value is $f = \frac{l-1}{l^2}$ for $w \to \infty$. When using binary trees ($l = 2$ as in our tests), a single operation moves between $1/4$ and $1/2$ of all tasks (of a given data structure) to another thread. If the

forest vector is not completely filled, the ratio might be even higher up to $f = 1$ but the lower boundary still applies.

The equations above consider only a single forest vector. When using distributed forest vectors, each one may have a different size, and so different values for w need to be considered. But even if all tasks are stored in a single forest vector, only a few operations are required to distribute the tasks among all threads, no matter how many tasks are actually stored.

Inserting or removing a task takes $O(\log t_{sum})$ steps, as the number of tasks stored in the forest vector grows exponentially with its length. This is, however, the worst case due to the operation to find the first empty or non-empty entry, which does not need to walk through the whole vector all the time. Task stealing is also $O(\log t_{sum})$, but can be made constant when storing the largest $i \leq w$ with $size(F_i) > 0$.

4 Implementation for the KOALA Framework

The KOALA framework is a generic task pool runtime environment. KOALA is component based and allows an easy exchange of several task pool components. Figure 9 gives an overview of the architecture. The first level contains basic components for efficient memory management and synchronization. Since tasks need to be described by their argument, such memory locations need to be allocated and freed often during runtime. The memory blocks are often of the same size and can therefore be handled more efficiently than with basic OS function calls.

The synchronization component provides an interface for thread synchronization and may use architecture-dependent implementations for efficient synchronization.

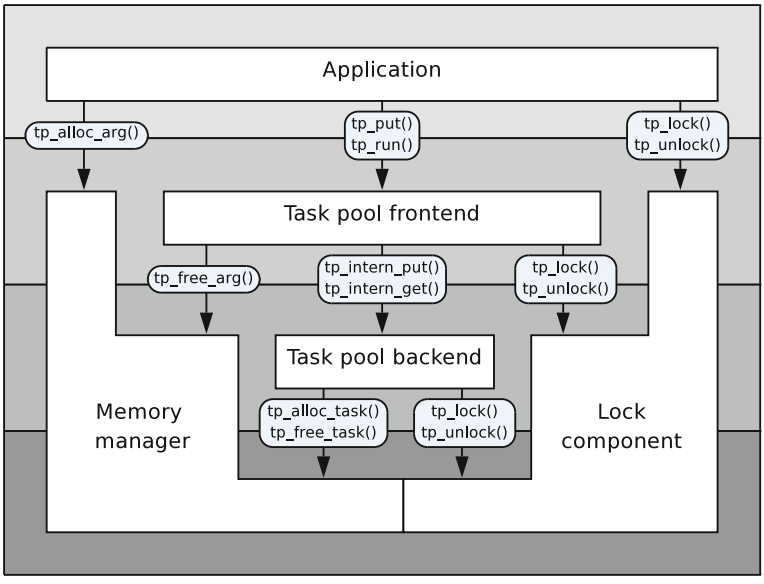

Fig. 9 Overview of the KOALA framework

Several hardware based synchronization implementations are available for different processor architecture. For example, for the IBM Power processors, two special read and write instructions, *Load & Reserve* (LR) and *Store Conditional* (SC) are provided to support the emulation of atomic read-modify-write (RMW) operations. The Itanium processors provide a *Compare & Swap* (CAS) operation. These operations can be used to implement specialized lock operations for task pool data structures.

Both components can be used by the higher levels of the framework to offer efficient methods for different task pool implementations and the actual applications.

The second KOALA level contains the task pool backend. This component manages the actual mechanism for storing and retrieving tasks. The adaptive task pool implementation described in the following section is done in this level. The interface offers methods to put and get tasks for a given thread and uses the underlying memory management to acquire storage and the lock component for synchronization (for example, for task stealing reasons).

The third KOALA level manages the actual task execution. The component retrieves new tasks whenever a thread finishes executing its current task and stores new tasks received from the application. This component can also be used to implement logging or profiling mechanism.

The top KOALA level is the actual task based application. An application inserts initial tasks into the frontend and may insert new tasks during runtime. The application may also use the memory manager component to acquire memory for storing task arguments and the lock component to use efficient synchronization at application level.

The actual implementation of each component including the application can be easily exchanged to use the best implementation for a specific target architecture.

4.1 Adaptive Task Pool Implementations

The adaptive task pool implementation (abbreviated as "ATP" in the following) uses the previously described data structure with a fixed limit $l = 2$ and a forest vector with 32 entries ($w = 31$), which enables to store over 17 billion tasks. A separate instance of the forest vector is assigned to every thread, implementing a distributed task pool.

Insertion and removal is always performed on the corresponding forest vector of the executing thread. The forest vectors of other threads are only accessed by the task stealing algorithm. Executing tasks that have been stolen from other threads usually implies a higher number of cache misses. Stealing large trees may reduce the number of cache misses as tasks in a tree are more likely created together, so they might share some data. For example, in a ray tracing application successively created tasks usually describe neighboring pixels in the final image. Rays traced for two neighboring pixels often hit the same object, which improves cache locality.

For further improvement in cache locality, a thread first searches for new tasks in its neighborhood when a stealing operation has to be executed; the neighborhood is identified by the thread ID, i.e., a thread i first checks the forest vectors of its direct neighbors $i - 1$ and $i + 1$, then $i - 2$ and $i + 2$, and so on. This takes possibly shared caches into account.

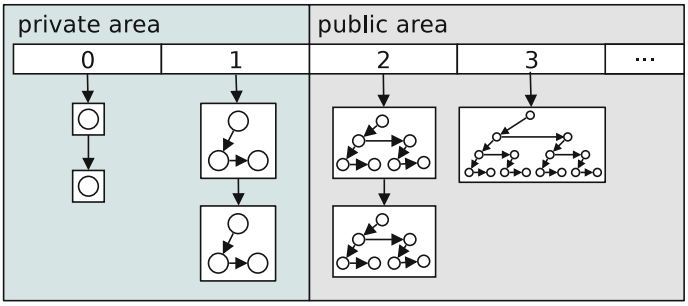

Fig. 10 Example of a partitioned forest vector into a private area and a public area

An alternative adaptive task pool implementation (abbreviated as "ATP-PRIV" in the following) additionally splits the forest vector into a private and a public area. This has the advantage that the corresponding thread can access the private area of its forest vector without any synchronization operations. The public area must still be accessed with synchronization, because other threads may try to perform stealing operations concurrently. The size of the private and public areas must be chosen carefully. If the private area is too large compared to the public area, other threads may be prevented from stealing tasks if they run out of tasks. On the other hand, using a private area that is too small may reduce the effect of lock-less access.

The forest vector can be used to dynamically adjust the private area, so that there are always enough trees available for stealing, while keeping the private area large enough for reduced access overhead. To do so, the forest vector is partitioned into a private area $F[0 \ldots priv\_length - 1]$ and a public area $F[priv\_length \ldots w]$ (see Fig. 10 for an illustration). This means that the private area covers the smaller trees where local insertion and removal operations are performed while the public area covers the larger trees for stealing. Two parameters are used to dynamically adjust the variable $priv\_length$. The first parameter *MAX_PRIVATE_LENGTH* assures that the private area does not grow to the whole vector:

$$0 \leq priv\_length \leq MAX\_PRIVATE\_LENGTH. \tag{6}$$

Since it is important that there are some trees available for stealing in any situation, a second parameter guarantees that at least

$$pub\_length \geq MIN\_PUB\_LENGTH \tag{7}$$

entries are available in the forest vector for stealing. This means that there should be at least *MIN_PUBLIC_LENGTH* entries available for stealing before the private area may grow up to *MAX_PRIVATE_LENGTH* entries. A relative boundary could also be used, but using a fixed upper limit for the private area has the advantage that the variable does not need to be updated too often if enough task are stored in the forest vector. Once the upper limit is reached, no further updates are required unless the size

of the public area shrinks under the lower boundary. The current implementation uses the parameters *MIN_PUBLIC_LENGTH* = 2 and *MAX_PRIVATE_LENGTH* = 3.

Adjustments to the $priv\_length$ variable are only done by the owner of the task data structure, since changes might make private entries public, which requires synchronization for access. Between task execution, the thread checks whether the variables still fulfill the limits mentioned above and adjusts the value $priv\_length$ accordingly, if necessary. As a side effect, new tasks may become available for stealing even if a thread removes a task from its data structure for execution.

In contrast to plain linked lists, the adaptive data structure can profit from the existence of a large number of tasks, giving it the option to create private areas for lock-free access. Therefore, the access overhead decreases with an increased number of tasks stored.

After starting a task-based program, the main thread creates some initial tasks and starts the worker threads, which then start to execute the initial tasks. More tasks can be created by the main thread as well as during the execution of tasks. Task-based application should be structured such that a large number of tasks is created as quickly as possible after program start to avoid idle times of the worker threads.

For the "ATP-PRIV" version, the size of the private and public areas is adapted to the number of tasks stored in the forest vector as described above. If less than pub_length entries of the forest vector are filled, the size of the private area is reduced to zero and all entries are public. This is important for load balancing if only a few tasks are left for execution. When the number of tasks stored in the forest vector increases again because of task creation or steal operations, the private area can be expanded again.

5 Experimental Evaluation

In this section, we present an experimental evaluation with different task pool implementations for different task-based applications. As applications, synthetic benchmarks as well as real-world applications have been investigated.

5.1 Experimental Setup

For the experimental evaluation, both adaptive task pool implementations "ATP" and "ATP-PRIV" are compared with two non-adaptive task pool implementations based on plain linked lists. The non-adaptive task pool *DQ* (distributed queues) uses a plain linear list for each thread to store the tasks. The block-oriented implementation *DQ-BL* (block-distributed queues) stores a fixed number of tasks in a single block and stores the resulting blocks in distributed linear lists, one for each thread. The block size is set to four for our experiments. This relatively small value for the block size is chosen to limit the load imbalance in case that there are only few tasks available. The implementation guarantees that at least one block is always private, so it can be accessed without locking. The adaptive task pool implementations *ATP* and *ATP-PRIV* use the data structure as described in the previous sections. The forest vector is completely public in the *ATP* implementation while *ATP-PRIV* uses private and public areas using the parameters $MIN\_PUBLIC\_LENGTH = 2$ and $MAX\_PRIVATE\_LENGTH = 3$.

Some of these applications have also been implemented in Intel's Threading Building Blocks [25]. It needs to be noted that TBB is a C++ library while our task pool library is written in C. Not all applications have been ported to TBB for our experiments. The hierarchical radiosity application, for example, uses dynamic task creation during the execution of tasks, and this cannot be directly transferred to TBB. The application could be re-written accordingly, but this would lead to a completely different implementation that is difficult to compare with our original version of the application. Moreover, TBB could only be used for the Itanium2 system but not for the Power architecture.

The following section gives a brief overview of the applications used to evaluate the overhead of the task pool implementation. Two synthetic benchmarks have been used to model different task execution schemes. Additionally, three more complex, real-world applications have been used to measure the performance.

Runtime tests have been performed for the applications as described in the following subsections on an SGI Altix 4700 machine with 256 dual-core Itanium2 processors per partition, running at 1.6 GHz, and on an IBM p690 machine with 32 Power4+ processors running at 1.7 GHz. For the SGI Altix, experiments with up to 64 cores have been performed. Each experiment has been repeated 10 times to calculate the resulting speedups. The calculated speedups are based on the best sequential runtime of the best implementation considered.

5.1.1 Benchmark to Determine Task Management Overhead

A synthetic application "sim1" is used to determine the overhead of the task management. A fixed number of tasks is created beforehand without further task creation during runtime. Every task has a predefined payload to simulate a certain task size. The payload may be empty to simulate empty tasks and therefore measuring the plain task management overhead. The tasks are created in a single location to stress the load balancing mechanism. The time measured excludes the time for creating the tasks, so it will directly show the actual load balancing overhead of the different task pool implementations.

5.1.2 Benchmark for Highly Dynamic Task Creation

A second synthetic application "sim2" is used to model a highly dynamic application with task creation during runtime. Starting with a small number of initial tasks, more tasks are created in a recursive manner. The task creation scheme is unbalanced.

Two parameters are used for controlling the task size f and for controlling the number of tasks t created initially. The work S of a task with an argument i can be described by the following recursive definition:

$$S(i) = \begin{cases} 100f & \text{for } i \leq 0 \\ 10f + S(i-2) + 50f + S(i-1) + 100f & \text{otherwise} \end{cases}$$

For $i \geq 1$, task $S(i)$ creates two tasks $S(i-1)$ and $S(i-2)$ and simulates some work depending on the parameter f before, between, and after the task creation.

For $f = 0$, the tasks do not perform any work but just create new tasks, so the tasks can be considered as "empty tasks". At the beginning, the synthetic application creates one task $S(i)$ for each $i, 0 \leq i < t$, so there are t tasks available for execution after initialization. For our experiments, we use a constant parameter $t = 35$, which creates 78, 176, 299 tasks during the execution.

5.1.3 Quicksort

The sorting application quicksort is well suited for task parallel execution, but shows an irregular behavior. The sequential quicksort program [18] creates two sub-arrays based on a pivot element (e.g., left/middle/right or random position) and recursively sorts both arrays. The parallel implementation creates new tasks for sorting the sub-arrays (if the array is large enough based on a fixed limit). Since one sub-array contains all elements less then or equal to the pivot element and the other sub-array contains all other elements, both arrays usually are not of equal size. The splitting depends on the input and cannot be predicted. Therefore the tasks created take different time to complete. A task-based execution with dynamic load balancing can handle this irregularity, but it is also important to make the existing parallelism available. At the beginning, there is only a single task partitioning the whole array. In the next step, there exist two tasks, then four, and so on. Because of this limitation, the resulting speedup cannot be linear. In the best case the array can be split into two arrays of the same size, yielding two tasks with the same execution time. In the first $\lceil \log p \rceil$ steps there are not enough tasks available for p threads, thus limiting the speedups. The parallel execution time can be calculated by

$$\begin{aligned} T_{\text{par}}(N, p) &= \sum_{i=1}^{\lceil \log p \rceil} \frac{N}{2^{i-1}} + \sum_{i=\lceil \log p \rceil + 1}^{\lceil \log N \rceil} \frac{N}{p} \\ &= 2N \left(1 - \frac{1}{2^{\lceil \log p \rceil}}\right) + \frac{N}{p}(\lceil \log N \rceil - \lceil \log p \rceil), \end{aligned} \tag{8}$$

where N is the size of the array to be sorted and p is the number of threads used for the sorting. The first term of Eq. 8 describes the part of the execution with limited parallelism (the first step considers the whole array, then the execution time is divided by 2, then divided by 4, and so on). The second term takes the rest of the computation into account where enough tasks are available for each of the threads.

Using Eq. 8 the ideal speedup can be calculated by

$$S_{\text{ideal}}(N, p) = \frac{T_{\text{seq}}(N)}{T_{\text{par}}(N, p)} = \frac{N \cdot \log N}{T_{\text{par}}(N, p)}. \tag{9}$$

Figure 11 shows some examples for different array sizes. Even for sorting a billion integers the speedup is still limited to 11. But the computation of S_{ideal} assumes the best case where every array is split into half. Moreover, several other points that may have an effect on the execution time are also not taken into account in Eq. 8. The execution time of the partitioning step depends on the number of swaps actually performed.

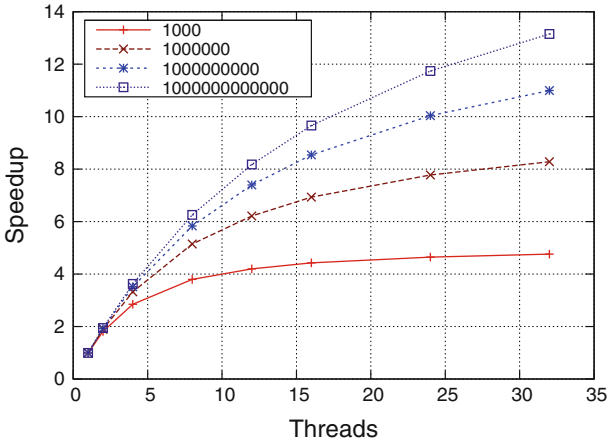

Fig. 11 Ideal speedups for the parallel quicksort application with several array sizes

The overhead for the task management is also not considered. So the actual speedup may be slightly smaller. But the resulting speedup can also be a little bit larger than S_{ideal} , since cache effects due to execution on multiple processors with local caches are not considered either.

To achieve a good speedup for this type of application it is important to make all parallelism available from the beginning of the execution, so task stealing even for single tasks is mandatory for a good speedup.

5.1.4 Ray Tracing

The parallel ray tracing application [28] for creating images from three-dimensional scenes is a more complex irregular application with a static task structure. For each pixel in the image, a ray is traced through the geometrical scene until the ray hits an object. Depending on the characteristics of the object surface, reflection and transmission rays are computed and traced through the scene. Therefore, the computations associated with a ray cannot be predicted and depends on the location and complexity of the objects in the scene. Load balancing is required to achieve good speedups. Task may be single lines of pixels or blocks of pixels. There are no data dependencies between the tasks, as the tracing algorithm only requires read-only access to the scene, which makes the parallel implementation easier. We use a modified implementation from the SPLASH-2 suite [30].

5.1.5 Hierarchical Radiosity

As second large irregular application, we consider the hierarchical radiosity method [15] which computes the light distribution between the objects of a three-dimensional scene. The application uses several computation phases and an iterative scheme to determine a stable solution for the light distribution. The application contains four different task types. One task type is used to compute the visibility factor for each pair

of patches of the scene. A second task type computes the light distribution between patches based on the incoming and outgoing light. A third task type refines the patches of the objects of the scenes into smaller patches as required to achieve a specific error bound. A fourth task type performs some post-processing actions. Especially the refinement tasks modify global data structures and create new tasks for all created sub-patches, so there are read-write accesses from different tasks creating additional dependencies. Due to the larger number of interactions between patches there is a large number of tasks which may create tasks dynamically. The actual amount of computations depends on the input, since the actual light distribution depends on which patches are (partly) visible. It is therefore not predictable which patches need to be refined due to incoming light and which lie in the shadow. Efficient load balancing for the large number of tasks is important for this kind of application. We use a modified implementation from the SPLASH-2 suite [28,29] for evaluation.

5.2 Experiments with Synthetic Applications

5.2.1 Task Management Overhead

The benchmark "sim1" from Sect. 5.1.1 is used to determine the time that is necessary to fetch and execute a certain number of tasks. Since each task created is uniform, the total time divided by the number of tasks executed gives a good estimation of the task management overhead per task (for empty payloads). Additional to the runtime of each task, the number of total stealing operations is also collected. To measure the overhead of the adaptive task pool implementations, runtime tests with 20 million empty task have been executed. Figure 12 shows the results on the SGI Altix 4700 machine. The time required to execute a single task is shown in Fig. 12a. When executed sequentially there is no contention for the synchronization operations and no stealing operations, so it shows the time needed to modify the data structure. The fastest sequential implementation is the block-distributed task pool "DQ-BL" with ≈100 ns management time per task. Using the adaptive data structure requires three times longer (≈300 ns per task). This is also due to the fact that the block-distributed task pool only needs to access the list of blocks every fourth task while the adaptive data structure removes tasks one by one. The distributed task pool "DQ" takes slightly longer than the adaptive task pool because the adaptive task pool uses a slightly more efficient two-level-lock. By using this lock type all accessing threads except the queue owner need to acquire an outer lock. The queue owner and the thread holding the outer lock try to acquire the inner lock before continuing. Thus, the inner lock must only synchronize two thread and can be therefore faster. With this implementation the queue owner can access its queue faster while the access of all other threads is slightly slower since they need to acquire both locks. Since the adaptive task pool is optimized to reduce the number of stealing operation, the advantage for the queue owner is usually larger than the disadvantage for all other threads.

When using more threads, the load balancing strategy is important for an efficient task management. In Fig. 12a for four threads, the adaptive task pool only needs ≈80 ns (averaged over four threads so it translates to ≈320 ns per thread) so the overhead for

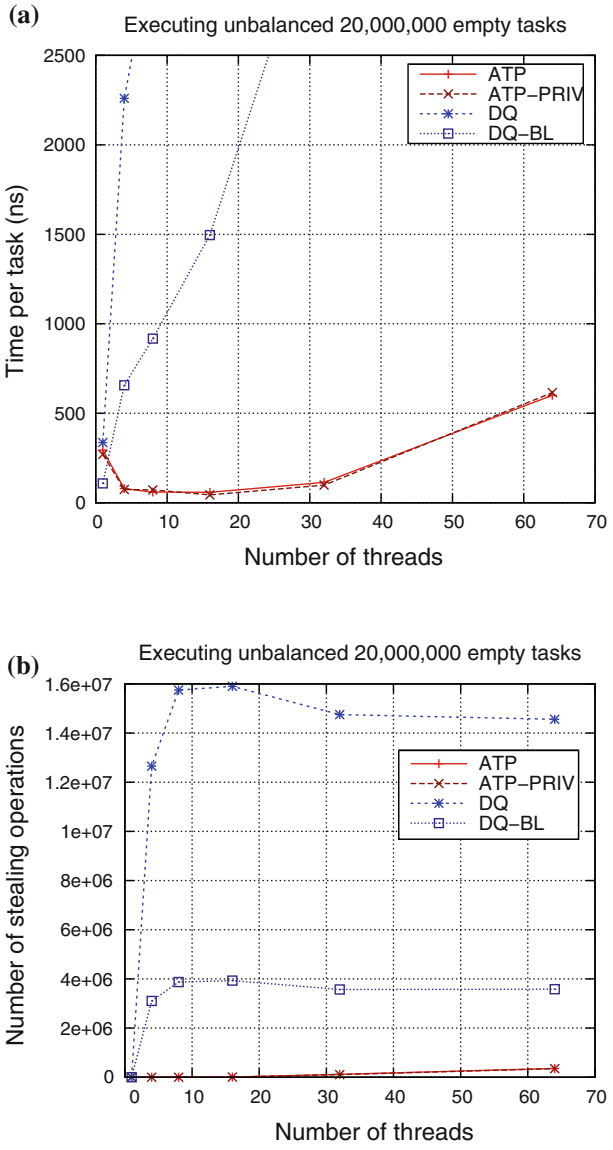

Fig. 12 Overhead statistics for scheduling 20,000,000 empty tasks (application sim1, SGI Altix 4700). **a** Time per task, **b** number of stealing operations

each thread is not significantly increased. In contrast, the block-distributed task pool "DQ-BL" takes ≈650 ns, since every thread except the master thread needs to steal a block of tasks for every four tasks executed.

Figure 12b shows the number of stealing operations successfully executed for each task pool implementation. Since all tasks are stored at the master thread, the distributed and block-distributed task pool execute a large number of stealing operations (for four

thread, these are more than three million for "DQ-BL", consequently four times more for "DQ"). The adaptive task pool on the other hand uses only ≈50 stealing operations during the execution of all 20 million tasks. With 32 and more threads, the number of stealing operations slightly decreases for the non-adaptive task pools, while the number of stealing operations slightly increases for the adaptive task pool. This is due to the increased contention: More threads try to steal adaptive blocks, so the size of the blocks become smaller. As a result, a larger number of smaller blocks is stolen in contrast to a smaller number of larger blocks with less contention. But still the number of stealing operations is significantly smaller than for any non-adaptive implementation, and the increase is actually not a real problem, since the synchronization overhead for so many threads trying to execute empty tasks dominates the runtime in this case anyway.

The results from this benchmark show that the adaptive task pool is able to efficiently manage a large number of tasks even in the case where the load is not equally distributed. Using blocks of fixed size decreases the overhead significantly, but may limit the available parallelism. However, such fixed blocks may also be applied to the adaptive data structure to further improve performance. The main advantage of the adaptive task pool is to be able to migrate single tasks between threads to achieve best possible parallel performance if only limited parallelism is available. On the other hand, when many tasks are available, the number of stealing operations is drastically reduced due to the stealing of complete task trees.

5.2.2 Highly Dynamic Task Creation

The synthetic application "sim2" from Sect. 5.1.2 is used to explore the behavior of applications for which a significant portion of the tasks is created dynamically during the execution of other tasks. For the tasks, different granularities, expressed by the parameter f are investigated. Figure 13 shows the speedups achieved when using empty tasks (parameter $f = 0$). Without any computations only the task management overhead is measured. The speedups on the SGI machine (Fig. 13a) are relatively good for such fine-grained tasks. The best speedup of 22 is achieved by the block-distributed task pool "DQ-BL", since it is guaranteed that only every four tasks a synchronization operation is necessary. The adaptive task pool (without private areas) requires synchronization operation for every insertion or removal, but the achieved speedup of ≈20 is only slightly smaller. In contrast, the distributed task pool with linked lists ("DQ") only reaches a speedup of ≈5. The TBB implementation is slower due to the C++ library overhead, but scales well and achieves the same speedup for 64 threads as the adaptive task pools. The block-distributed task pool can achieve even better speedups, but the available parallelism is too small for 64 threads, since there are only 35 tasks available at the beginning of the execution which are stored in nine blocks. The adaptive task pool will build larger trees when more tasks become available during execution, but due to the tree splitting there are enough tasks available for all threads even at the beginning. The use of private areas in the "ATP-PRIV" implementation does not improve the performance due to the additional overhead for managing the boundary updates.

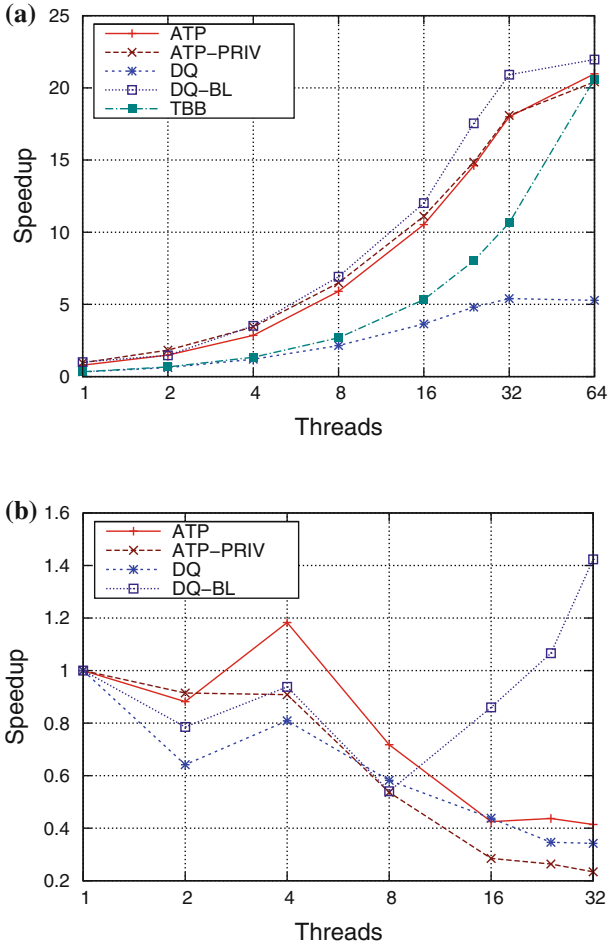

Fig. 13 Speedups of the synthetic application sim2 for empty tasks ($f = 0$) with adaptive task pools. **a** SGI Altix4700, **b** IBM p690

The speedups on the IBM machine (Fig. 13b) are always below 2, so the fine-grained task cannot be executed equally well. The adaptive implementation are even slightly faster than the block-distributed task pool when using a small number of threads.

For slightly larger tasks (Fig. 14, $f = 10$), the scalability is much better on the SGI machine (Fig. 14a). The speedups are above 50 for all but the distributed task pool with linked lists ("DQ"). The overhead of the TBB implementation is much smaller and the speedups are similar to the task pool implementation. In absolute runtimes, the adaptive task pool "ATP" is the fastest.

On the IBM machine (Fig. 14b) the speedups are slightly improved for larger tasks but the tasks are still too small for efficient execution. The block-distributed task pool works relatively well, but the adaptive implementations are still faster than the distributed task pool with linked lists.

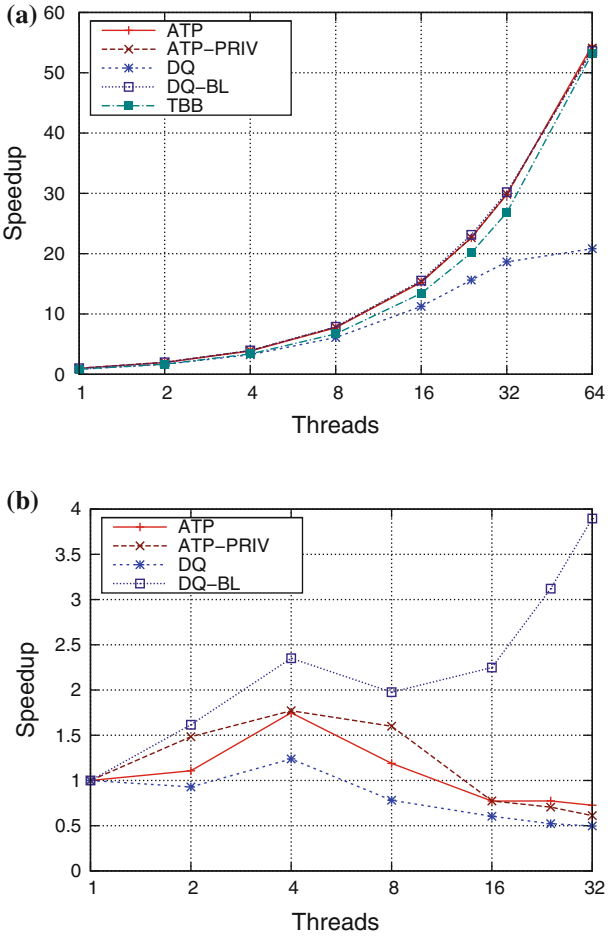

Fig. 14 Speedups of the synthetic applications for small tasks ($f = 10$) with adaptive task pools. **a** SGI Altix4700, **b** IBM p690

5.3 Experimental Results with Real-world Applications

5.3.1 Quicksort

Using the available parallelism is especially important for the parallel quicksort application, since there are only few tasks available at the beginning of the computation. Figure 15 shows the resulting speedups for sorting 100 million integers. The best speedup of 7.01 on the SGI machine is achieved by the adaptive task pool for 16 threads; on the IBM machine the best speedup is 6.97 for 32 threads.

The block-distributed task pool cannot keep up with the performances of the other implementations. On the SGI machine its best speedup is only 4.1 and on the IBM machine its best speedup is 2.74. The blocks of tasks hide some parallelism especially

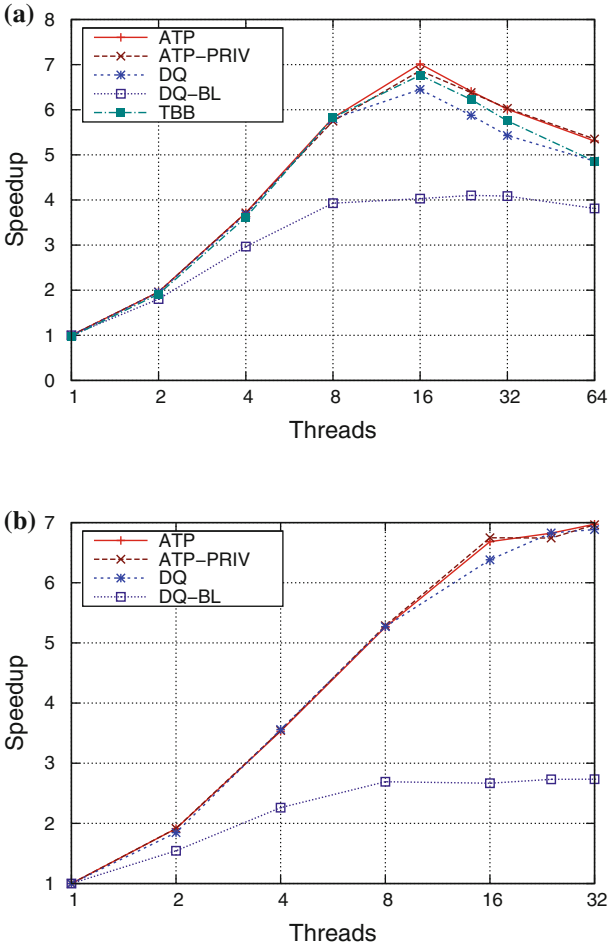

Fig. 15 Speedups of the quicksort application with adaptive task pools and an array size of 100,000,000. **a** SGI Altix4700, **b** IBM p690

at the beginning, since single tasks within a block cannot be stolen by other threads. Therefore, idle threads cannot take part in the computation. The adaptive task pools are better suited for this kind of task creation scheme, since there are no large trees at the beginning but growing trees during the recursive task creation.

5.3.2 Ray Tracing Method

The ray tracing application can be efficiently parallelized and since the scene data is read-only, good speedups can be achieved. The implementation considered does not create new tasks during runtime and since all tasks are distributed equally over all threads, load imbalance only occurs due to different computation times for different pixels.

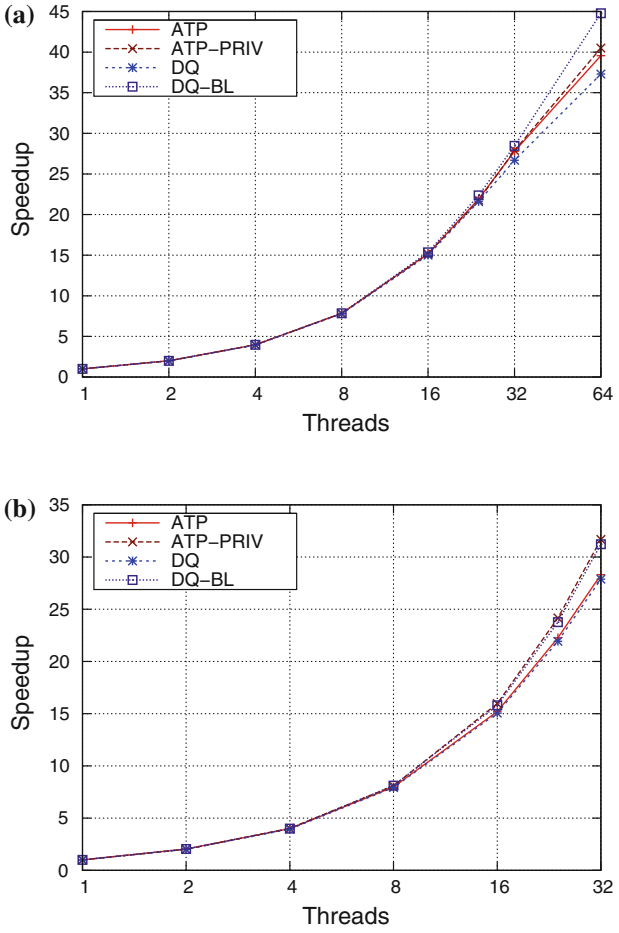

Fig. 16 Speedups of the ray tracing application with adaptive task pools (scene "car", 1024 × 1024). **a** SGI Altix4700, **b** IBM p690

Figure 16 shows the resulting speedups for a given scene on both machines. The speedups are quite similar for the different implementations except for the largest number of threads used. On the SGI machine, the best speedup of ≈45 for 64 threads is reached by the block-distributed task pool, followed by the adaptive task pool implementations with a speedup of ≈40. The distributed task pool with linked list is the slowest. On the IBM machine the block-distributed task pool achieves a speedup of 31.2, but the adaptive task pool with private areas is even slightly faster (speedup 31.7). The adaptive task pools achieve good performance also for this kind of application, although the task handling is not the bottleneck here.

5.3.3 Hierarchical Radiosity Method

The hierarchical radiosity is a complex irregular application with four different task types and a large number of dynamically created tasks, which requires efficient task

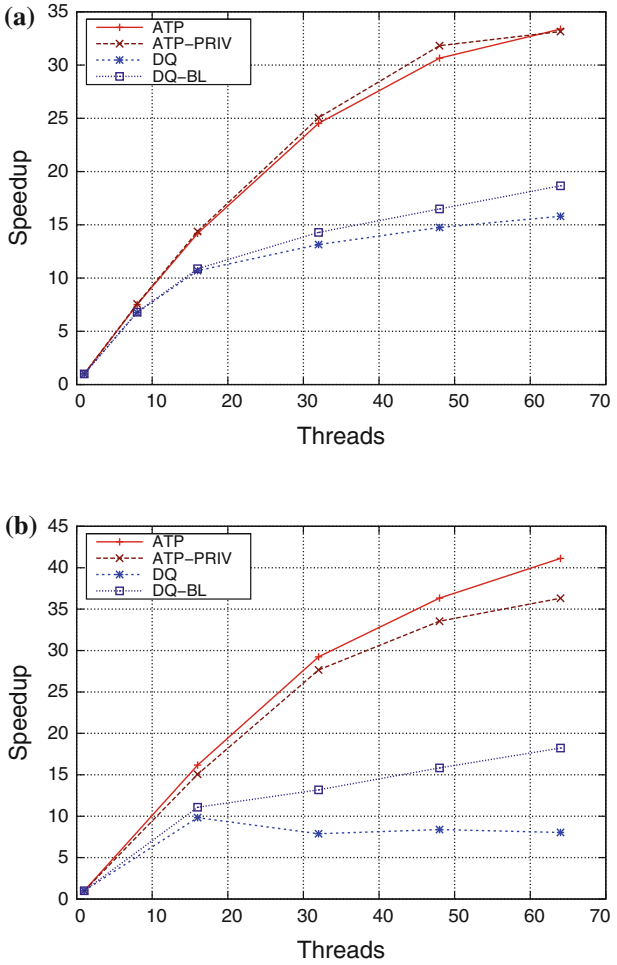

Fig. 17 Speedups of the hierarchical radiosity application with adaptive task pools (scene "room11"). **a** SGI Altix4700, **b** IBM p690

management and load balancing. Figure 17 shows the resulting speedups for a given scene on both machines. The adaptive task pool implementations are able to achieve good speedups on both machines. On the SGI machine, "ATP-PRIV" is slightly better than "ATP", whereas on the IBM machine, "ATP" achieves better speedups. On the SGI machine, the best speedups is ≈33 for 64 threads, and on the IBM machine the best speedup is ≈41 for 64 threads. The block-distributed task pool is significantly slower, reaching a speedup of ≈18 on both machines. The distributed task pool with linked list is even slower, especially on the IBM machine.

The non-satisfactory results for the non-adaptive task pools come from the specific task creation method in the application. Initial tasks cannot be easily distinguished from task created during runtime. Therefore, many initial tasks are stored in a single data structure instead of distributing them among all data structures. This requires a lot

of task stealing operations, which can be handled significantly better by the adaptive task pools. Even if all tasks are stored in a single forest vector, the load balancing can be done in a few steps, in contrast to the non-adaptive variants, which require task stealing for every task executed ("DQ") or every fourth task executed ("DQ-BL").

6 Summary and Conclusions

The adaptive data structure presented in this paper can be used to efficiently handle a large number of executable tasks in a distributed way. Executable tasks are stored in larger groups with a growing number of tasks, while the group sizes shrink as the number of tasks shrinks. Runtime tests have shown that the resulting overhead is small and that the performance is usually at least as good as the performance of the distributed task pools with plain linked lists. The data structure is able to balance a large number of tasks in only a few operations, so there is no need for a good initial task distribution. The runtime tests have shown that the adaptive task pool achieves good results for all applications considered where non-adaptive task pools cannot always work well. This includes scenarios with many fine-grained tasks and situations with limited parallelism.

Using the adaptive data structure, the cost for task stealing is reduced, since a large number of tasks can be moved in a single operation with only one write access to a remote forest vector. The stolen tree may contain related tasks, which can reduce the locality penalty for stealing tasks. The adaptive task pool implementations have been integrated into the KOALA framework, which provides a library-based interface to the implementations. Thus, task-based applications can use the KOALA interface to benefit from the adaptive task stealing mechanism provided. The KOALA framework can be customized for a specific hardware architecture by replacing the memory manager and synchronization component by specialized versions which exploit the specific architecture of the given machine.

Using a library-based approach like in the KOALA framework has the advantage that it can easily been integrated into and combined with other frameworks and languages, thus leading to a great flexibility. Nevertheless, it would also be useful to use the adaptive approach for an efficient support of task-based executions in languages like Fortress or X10 or environments implementing the OpenMP 3.0 API. To do so, the task pool backend of the KOALA framework could be integrated into the corresponding runtime system. This would enable the use of the adaptive mechanisms without a change in the language features.

Acknowledgments We thank LRZ Munich and NIC Jülich for providing access to their computing systems.

References

1. Agrawal, K., He, Y., Leiserson, C.E.: Adaptive work stealing with parallelism feedback. In: Yelick, K.A., Mellor-Crummey, J.M. (eds.) Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (22th PPOPP'2007), pp. 112–120. ACM, New york (2007)

2. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, G.L. Jr., Tobin-Hochstadt, S.: The Fortress Language Specification, version 1.0beta. Technical report, SUN, Mar (2007)
3. Banicescu, I., Hummel, S.F.: Balancing processor loads and exploiting data locality in n-body simulations. In: Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM), p. 43. ACM, New York, NY, USA (1995)
4. Banicescu, I., Velusamy, V., Devaprasad, J.: On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. Clust. Comput. J. Netw. Softw. Tools Appl. **6**, 215–226 (2003)
5. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: A programming model for the cell BE architecture. In: Proceedings of the 2006 ACM/IEEE SC'06 Conference. IEEE (2006)
6. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming (PPOPP'1995), pp. 55–69. ACM (1995)
7. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science, pp. 356–368. IEEE Computer Society (1994)
8. Burton, F.W., Sleep, M.R.: Executing functional programs on a virtual tree of processors. In: FPCA '81: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, pp. 187–194. ACM, New York, NY, USA. (1981)
9. Cariño, R., Banicescu, I.: Dynamic load balancing with adaptive factoring methods in scientific applications. J. Supercomput. **44**(1), 41–63 (2008)
10. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: Johnson, R., Gabriel, R.P. (eds.) Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 519–538. ACM, New york (2005)
11. Callahan, D., Chamberlain, B.L., Zima, H.P.: The cascade high productivity language. In: 9th international workshop on high-level parallel programming models and supportive environments (HIPS'04), pp. 52–60. IEEE (2004)
12. Dinan, J., Larkins, D., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11. ACM (2009)
13. Duran, A., Corbalan, J., Ayguade, E.: An adaptive cut-off for task parallelism. In: SC'08 USB Key. ACM/IEEE, Austin, TX, Nov. 2008. Universitat Politecnica de Catalunya (2008)
14. Halstead, R.H. Jr.: Implementation of multilisp: Lisp on a multiprocessor. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, pp. 9–17. ACM, New York, NY, USA. (1984)
15. Hanrahan, P., Salzman, D., Aupperle, L.: A rapid hierarchical radiosity algorithm. ACM SIGGRAPH Comput. Graph. **25**(4), 197–206 (1991)
16. Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing (PODC'02), pp. 280–289. ACM (2002)
17. Hippold, J., Rünger, G.: Task pool teams for implementing irregular algorithms on clusters of SMPs. In: Proceedings of IPDPS. Nice, France, CD-ROM (2003)
18. Hoare, C.A.R.: Quicksort. Comput. J. **5**(4), 10–15 (1962)
19. Hoffmann, R., Rauber, T.: Fine-grained task scheduling using adaptive data structures. In: Proceedings of Euro-Par 2008, vol. 5168 of *LNCS*, pp. 253–262. Springer (2008)
20. Kalé, L.V., Krishnan, S.: CHARM++. In: Wilson, G.V., Lu, P. (eds.) Parallel Programming in C++, chap. 5, pp. 175–214. MIT Press, Cambridge, MA (1996)
21. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. ACM SIGARCH Comput. Arch. News **35**(2), 162–173 (2007)
22. Kumar, V., Grama, A., Vempaty, N.: Scalable load balancing techniques for parallel computers. J. Parallel Distrib. Comput. **22**(1), 60–79 (1994)
23. Polychronopoulos, C., Kuck, D.: Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Trans. Comput. **C-36**(12), 1425–1439 (1987)
24. Power Architecture editors, developerWorks, IBM: Just Like Being There: Papers from the Fall Processor Forum 2005: Unleashing the Power of the Cell Broadband Engine—A Programming Model Approach. IBM developerWorks (2005)

25. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly (2007)
26. Schloegel, K., Karypis, G., Kumar, V.: A Unified algorithm for load-balancing adaptive scientific simulations. In: Proceedings of Supercomputing'2000, pp. 75–75. IEEE (2000)
27. Singh, J.: Parallel Hierarchical N-Body Methods and their Implication for Multiprocessors. PhD thesis, Stanford University (1993)
28. Singh, J.P., Gupta, A., Levoy, M.: Parallel visualization algorithms: Performance and architectural implications. IEEE Comput. **27**(7), 45–55 (1994)
29. Singh, J.P., Holt, C., Tosuka, T., Gupta, A., Hennessy, J.L.: Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. J. Parallel Distrib. Comput. **27**(2), 118–141 (1995)
30. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd International Symposium on Computer Architecture, pp. 24–36. ACM, Santa Margherita Ligure, Italy (1995)
31. Wu, M., Li, X.-F.: Task-pushing: A scalable parallel GC marking algorithm without synchronization operations. In: Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007). IEEE (2007)
32. Xu, C., Lau, F.C.: Load Balancing in Parallel Computers: Theory and Practice. Kluwer Academic Publishers, Dordrecht (1997)