Fine-Grained Task Scheduling Using Adaptive Data Structures

Ralf Hoffmann and Thomas Rauber

Department for Mathematics, Physics and Computer Science
University of Bayreuth, Germany
{ralf.hoffmann,rauber}@uni-bayreuth.de

Abstract. Task pools have been shown to provide efficient load balancing for irregular applications on heterogeneous platforms. Often, distributed data structures are used to store the tasks and the actual load balancing is achieved by task stealing where an idle processor accesses tasks from another processor. In this paper we extent the concept of task pools to adaptive task pools which are able to adapt the number of tasks moved between the processor to the specific execution scenario, thus reducing the overhead for task stealing significantly. We present runtime experiments for different applications on two execution platforms.

1 Introduction

The efficient parallel execution of applications with an unpredictable computational behavior requires the use of sophisticated load balancing methods. In this paper, we consider task-based load balancing methods for platforms with a shared address space. The computations performed by an application are arranged as fine-grained single-processor tasks which can be executed by an arbitrary processor. Tasks can be created dynamically during the execution of the application, and usually there are much more tasks available than processors for execution. Tasks that are ready for execution are stored in specific data structures, so-called task pools, from which they can be accessed by idle processors for execution. The task pool runtime environment schedules these tasks to the available processors on demand using varying strategies. The processors independently fetch new tasks from the task pool as soon as their previous task has been completed. As long as there are enough tasks available for execution the processors remain busy up to the end of the application. Smaller tasks enable a better load balancing but lead to a larger overhead.

A simple implementation of a task pool for storing and scheduling tasks is a central data structure. Since all processors need to access this shared data structure, scalability may be limited due to the synchronization required, especially if the data structure is accessed often. Distributed data structures are often better suited as the synchronization overhead can be reduced, e.g., by using a separate task list for each processor. For load balancing the distributed data structure still needs to be shared in some way so that idle processors can obtain tasks from other processors for execution. Moving tasks from one processor to another is referred to as task stealing.

In previous work we have shown that the task pool overhead can be reduced by improving the synchronization operations [1]. This enables an efficient realization of

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 253–262, 2008.
© Springer-Verlag Berlin Heidelberg 2008

task-based applications also if fine-grained tasks are used. A small task granularity and therefore many fine-grained tasks enables a good load balancing especially for large parallel systems. But using many fine-grained tasks may increase the number of stealing operations performed, thus increasing the overhead for the load balancing. In this paper we present an adaptive task pool implementation which is especially suited for storing a large number of tasks. The new implementation adapts the number of tasks accessed by a single stealing operation to the total number of tasks stored in the task pool. Experiments with several large applications show that the overhead for task stealing and other operations can be significantly improved by using the adaptive implementation.

The rest of the paper is organized as follows: Section 2 presents the adaptive data structure and Section 3 describes the adaptive task pool. Section 4 presents experimental results. Section 5 discusses related work and Section 6 concludes the paper.

2 Adaptive Data Structure

For fine-grained tasks, an important requirement for the data structure to store the executable tasks is a fast insertion and removal of tasks. To improve the task stealing operation the data structure needs to support the removal of large chunks of tasks in a single operation. Distributed queues implemented as linear lists are often used to store the task, and their performance can be improved by additionally using blocks of tasks for task stealing. A linear list takes constant time for insertion or removal of tasks but each task has to be removed in a separate step. Using blocks of tasks allows to steal several task with one operation but it may limit the parallelism if there are only a few executable tasks available which might be stored in a single block on a single processor.

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 of a tree stores a pointer to the first child and a pointer to the next sibling in the same level.

The trees are stored in a forest vector $F[0..w]$ 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. Since all trees are fully balanced, new tasks can only be inserted in the first level $F[0]$ or by combining existing trees with the same depth into a new tree with a larger depth.

Although the use of a list of children does not limit the width of a tree level we still limit the length by a specific criterion. This criterion can be used to control the growing of the individual trees. The limit can be any fixed number of children or even dynamic limits based on the actual work described by each tasks. For simplicity we assume a fixed limit l for the following description of the insertion and removal process.

Insertion of tasks: New tasks are inserted into the forest vector as outlined in Figure 1. Figure 2 illustrates the process of inserting a new task (T) for $l = 2$. Since the first level of F is fully occupied and the first non-full level is 1, the new task becomes the root of a new tree containing all trees of level 0 as children. The new tree now forms a fully balanced tree of level 1 and is therefore inserted into $F[1]$.

```
Create new tree t = new T_0 {
  t → task = new task
  t → child = NULL
  t → sibling = NULL
}
if ( size(F_0) < l ) {
  t → sibling = F_0
  F_0=t
} else {
  find first i with size(F_i) < l
  t → child = F_{i−1}
  F_{i−1} = NULL
  t → sibling = F_i
  F_i = t
}
```

Fig. 1. Task insertion

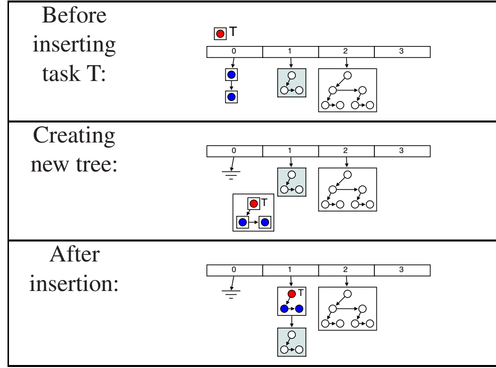


Fig. 2. Example insertion of a new task T

```
Find first i with size(F_i) > 0
t = F_i
F_i = t → sibling
if ( i > 0 ) {
  F_{i−1} = t → child
}
return t → task
```

Fig. 3. Task removal

```
Find largest i with size(F_i(p_2)) > 0
Remove tree from p_2 {
  t = F_i(p_2)
  F_i(p_2) = t → sibling
}
Insert into F(p_1) and split {
  if ( i > 0 ) {
    F_{i−1}(p_1) = t → child
  }
}
return t → task
```

Fig. 4. Task stealing from processor p_1 to p_2

Task removal: The removal is done similarly (in opposite direction) as described in Figure 3 but the ordering is not strictly FIFO (first in, first out). This may have a negative impact on the locality of the executed tasks, but implementing strict FIFO order would definitely have a performance penalty for both task insertion and removal, since the removal operation would need to search for an empty slot and split the following tree into sub-trees to undo the insertion. Especially for a large number of tasks the insertion and removal operations would always work with large trees. On the other hand, the proposed implementation tries to access tasks from the smaller side of the forest making the find operation faster.

Task stealing: The data structure facilitates migration of several task with one operation by moving a tree from $F[0..w]$ from one processor to another. To reduce the total number of stealing operations, we choose to remove the largest tree from another processor using the algorithm described in Figure 4.

Access times and number of tasks: To approximate the number of task stolen in a single operation we consider a completely filled data structure using a fixed limit l. Each tree in level i stores $\frac{l^{i+1}-1}{l-1}$ tasks and a full forest $F[0..w]$ contains

$$t_{sum} = \sum_{i=0}^{w} l * \left(\frac{l^{i+1}-1}{l-1} \right) = \frac{l^{w+3} - (w+2)l^2 + (w+1)l}{(l-1)^2}$$

tasks. The steal operation removes a tree from level i so the fraction of tasks stolen in one operation is

$$f = \frac{l^{i+1} - 1}{l - 1} * \frac{(l-1)^2}{l^{i+3} - (i+2)l^2 + (i+1)l}$$
$$= (l-1)\frac{l^{i+1} - 1}{l^{i+1}} \frac{1}{l^2 - \frac{(i+2)l^2 - (i+1)l}{l^{i+1}}}$$

The largest value for f is $f = 1/l$ for $i = 0$ and the smallest value for f is $f = \frac{l-1}{l^2}$ for $i \to \infty$. For a binary tree ($l = 2$) one steal operation removes between $1/4$ and $1/2$ of all available tasks. The fraction may be higher if the forest is not completely filled but the lower limit still applies. After removing a tree from a processor the task stealer puts the remaining sub-trees into its own data structure so $\approx 1/l$ of these tasks can be stolen by other processors again in a single operation. In a situation where only one processor stores most of the tasks the work can still be redistributed using only a few operations.

Inserting or removing a task takes $O(\log t_{sum})$ steps in the worst case as the number of tasks stored in the forest vector grows exponentially with its length. Typically the operation uses much less steps as the find operation does not need to walk through the whole vector all the time but can stop at the first non-empty entry. Task stealing is also $O(\log t_{sum})$ but is $O(1)$ when storing the largest $i \leq w$ with $size\,(F_i) > 0$.

3 Adaptive Task Pool

The current task pool implementation uses the data structures as described in the previous section with a limit $l = 2$, thus using fully balanced binary trees. Each processor stores its own instance of this data structure. To reduce the synchronization overhead the locks uses hardware operations for mutual exclusion.

If a processors runs out of work, i.e. its forest vector is empty, it will search for new work by task stealing. For this, the processor visits all other processors in a given order for an available tree in their forest vector. The tree is then split into two sub-trees which are stored in the forest vector of the stealing processor for later execution but they are also available for stealing. The root of the stolen tree is the task to be executed next.

Stealing work from other processors may have negative locality effects as the data for the work units is possibly not in the cache of the current processor, so it needs to be transferred from remote memory (especially on NUMA systems there is an additional access cost for remote memory from other processors). But grouped stealing as implemented by the adaptive task pool may reduce this effect. Task created by the same processor in direct succession often solves sub-problems of some part of the whole input problem so they may access the same portion of input data. Each tree in the forest vector can potentially contain such related tasks so stealing it may impose an access penalty for some but not all tasks in this group. Due to the tree combine and split steps in the algorithm some sub-trees might be created by a different processor than some other sub-tree. In rare scenarios a tree may contain tasks created by many different processors but these potentially related tasks will still be grouped in some sub-trees. To additionally reduce the locality problems when stealing tasks, the processors search for

available tasks in their neighborhood (based on their IDs). So a processor steals work from the nearest processor with available work.

Since lock contention can be a problem when using a large number of processors, private and public areas are used to reduce the overhead as the data structure owner can access the private area without any locking. The size of the private area is a tradeoff between synchronization overhead and available parallelism. If the private area stores a relatively large number of tasks the parallelism may be limited as these task cannot be stolen by other processors. Smaller private areas on the other hand limit the positive effect of lock-less access for the owner.

We use the forest vector to implement a dynamical resizing of private and public areas. Two parameters are used to control these areas. The forest vector F is divided into a private part $F[0..priv_length - 1]$ and a public part $F[priv_length..w]$. The private area is limited to $0 \leq priv_length \leq MAX_PRIV_LENGTH$. As an additional criterion for the size of the private area, we use another variable pub_length with a higher priority than $priv_length$. The private area is only non-empty if $pub_length \geq MIN_PUB_LENGTH$. This decision is made because exploiting parallelism has a high priority, so there will only be a private area if there are enough trees available in the public area. Limiting the private area to an upper limit also reduces the number of updates of these variables for frequent accesses.

The adaptive task pool implementation takes significant advantage of the existence of a large number of tasks. The trees grow with the number of tasks; therefore the number of tasks stolen also increases. The size of the private area also depends on the number of available tasks, i.e., so creating many tasks actually reduces the overhead for storing and scheduling them.

4 Experimental Results

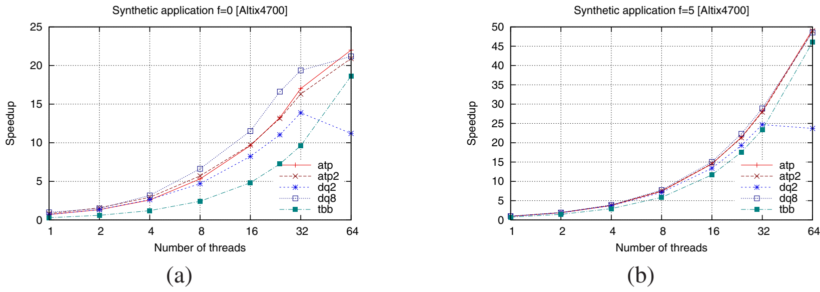
Runtime experiments are performed on a SGI Altix 4700 machine with 4864 dual-core Itanium2 processors running at 1.6 GHz and an IBM p690 machine with 32 Power4+ processors running at 1.7 GHz. To evaluate the implementation we use several irregular applications. Additionally, the use of a synthetic application makes it possible to directly measure the overhead of the task pool implementation.

4.1 Synthetic Task Application

The synthetic application uses two parameters f for controlling the task size and t for controlling the number of tasks created. The work S of a task with an argument i can be described by the following recursive definition:

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

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. Initially there is one task $S(i)$ created for each $0 \leq i < t$ so there are t tasks available for execution. For our experiments we use a constant parameter $t = 35$. For comparison



(a) (b)

Fig. 5. Speedups for the synthetic application on the Altix4700

we use two different implementations of the adaptive task pool and two non-adaptive task pool implementations. Additionally we also use the TBB [2] library to execute the tasks of the synthetic application. For the comparison with TBB it should be noted that TBB is based on C++ while the adaptive task pool framework is based on C. The non-adaptive task pool *dq2* uses a plain linear list for each processor to store each tasks. The block-oriented implementation *dq8* stores a fixed number (4) of tasks in a single block and stores these blocks in linear lists, one for each processors. Due to the implementation at least one block is always private so it can be accessed without locking. The adaptive task pool implementations *atp* and *atp2* use the data structure as described in the previous sections. The forest is completely public in the *atp* implementation while *atp2* uses private and public areas using the parameters $MIN_PUB_LENGTH = 2$ and $MAX_PRIV_LENGTH = 3$.

Figure 5 shows the speedups for the implementations based on the best sequential execution for $f = 0$ (Figure 5a) and $f = 5$ (Figure 5b). For empty tasks the speedups do not exceed 22 for a maximum of 64 threads. For up to 32 threads the *dq8* task pool performs best. This is due to the lower overhead in handling the data structure and the availability of enough tasks for execution. In our test with $t = 35$ there are 35 tasks at the beginning distributed round robin over all threads. With more threads even new tasks will still be stored in the private block so some threads have to wait. The adaptive task pools (*atp* and *atp2*) can handle this situation better and are at least as fast as the block-distributed task pool *dq8*. The use of private and public areas does not give an advantage in this case, since the overhead of dynamically adjusting the areas is higher than the benefit from the reduced number of locks required in the private area. The *dq2* implementation with public lists cannot compete and is significantly slower with 64 threads. The alternative implementation of the synthetic application using TBB scales well (almost linearly in respect to sequential execution of the TBB implementation) but is still slower due to the additional overhead in the C++ implementation.

For slightly larger tasks with a factor $f = 5$ (Figure 5b) all implementations can achieve a significantly better scalability reaching a speedup of almost 50. The adaptive task pools and the block-distributed task pool *dq8* achieve similar speedups. Only the *dq2* implementation cannot keep up with the other implementations. This is remarkable as *dq2* as well as *atp* uses a lock to protect access to the data structure and no private

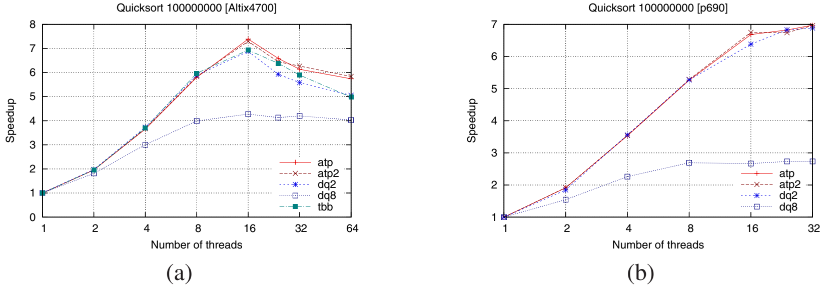


Fig. 6. Speedup for the quicksort application on the Altix4700 (a) and on the p690 (b)

area. The additional benefits for the adaptive task pool, especially task grouping and grouped stealing, actually improve the performance in this case.

4.2 Quicksort

The sequential quicksort creates two sub-arrays based on a pivot element and recursively sorts both arrays. The parallel implementation creates new tasks for sorting the sub-arrays (if the array is large enough based on the fixed limit). As both arrays typically do not have the same size, depending on the pivot element, both tasks created also take different time to complete. Task based execution can cope with this irregularity but the main key is making 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. In this implementation the speedup cannot be linear (as described in [1]) and it is important to allow task stealing even for a single task.

Figure 6 shows the speedups for sorting 100,000,000 integers. The best speedup for the Altix4700 machine is 7.39 for 16 threads reached by the adaptive task pool implementation *atp* without private areas and 6.97 for 32 thread on the p690 machine also for the implementation *atp*. The block-distributed task pool dq8 shows bad performance never reaching a speedup above 4.3 and 2.74 respectively. This is however expected as the tasks stored in the private block cannot be stolen by other idle threads. As the parallelism is already limited at the beginning, this significantly reduces the overall speedup. The adaptive task pool implementations can also handle this special situation making all task available if necessary and building large blocks for stealing when possible.

4.3 Ray Tracing and Hierarchical Radiosity

The parallel ray tracing application [3] is a more complex irregular application which creates images from three-dimensional scenes. For each pixel in the image a ray is traced through the geometrical scene. The work associated with a ray depends on the location and complexity of the objects in the scene. We use a modified implementation from the SPLASH-2 suite [4]. As second large application we consider the hierarchical radiosity [5] which computes the light distribution between the objects of a three-dimensional scene. The application uses four different task types to compute the

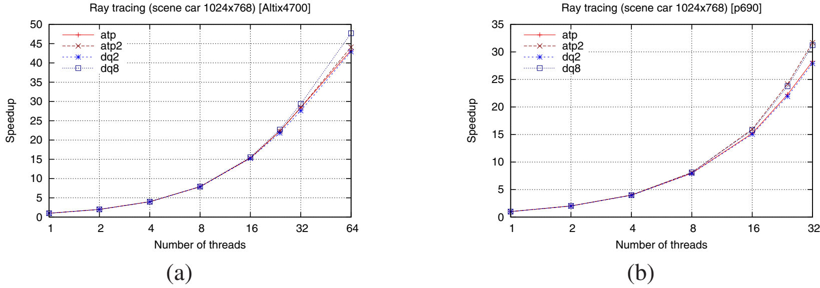


Fig. 7. Speedups for the ray tracing application on the Altix4700 (a) and on the IBM p690 (b)

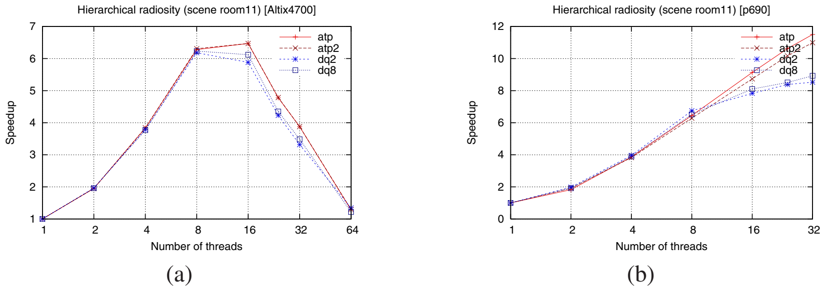


Fig. 8. Speedups for the hierarchical radiosity on the Altix4700 (a) and on the p690 (b)

visibility factors as well as the light distribution between patches. Another task type refines the scenes into smaller patches as required to achieve a specific error bound and a fourth task type performs some post-processing actions. New tasks will be created at runtime to refine the patches so there are read-write accesses from different tasks creating additional dependencies. We use the parallel implementation from the SPLASH-2 suite for evaluation.

Figure 7 shows the results of the ray tracing application on the Altix4700 machine and the p690 machine. The ray tracing application shows good speedups for any task pool implementation. The *dq8* implementation performs best with a speedup of 47.7 for 64 threads on the Altix4700 machine and 31.2 for 32 threads on the p690 system. The low overhead and guaranteed existence of private blocks gives an advantage. However, the adaptive implementations also show good performance and are slightly faster than the *dq2* implementation with very low overhead. In this application the use of private and public areas in the adaptive task pools helps to improve the performance; on the p690 systems the implementation *atp2* is the fastest with a speedup of 31.7.

The speedups for the hierarchical radiosity (Figure 8) are significantly worse than the for the ray tracing due to the additional dependencies and small task sizes. The best speedup is 6.47 for 16 threads for both adaptive task pool implementation on the Altix4700 system while the adaptive task pools can reach a speedup of 11.5 for 32 threads on the p690 machine. The previous results suggested that the non-adaptive implementation *dq8* should be faster because of the lower overhead. But if there are not always

enough task available or more task stealing is required the adaptive implementation can outperform the other implementations. Because of this the ray tracing applications works well for *dq8* as all tasks are created at the beginning with no additional task creation. Radiosity on the other hand spawns new tasks so the adaptive task pools can handle this situation better.

5 Related Work

Dynamic load balancing is often used to execute irregular applications efficiently [6]. There are several libraries and programming languages which utilize dynamic load balancing. Charm++ [7] is based on C++ and offers load balancing by using object migration. TBB [2] is a C++ library and allows the use of different programming paradigms. It offers, for example, parallel loops to describe data parallelism but also enables the programmer to explicitly create tasks for a task parallel execution. TBB uses hardware operations to implement synchronization with low overhead, it is however currently available only for a limited number of architectures. Similar approaches are used in language like Fortress [8] or X10 [9] which focus on data parallelism but task parallelism is also supported.

Task based execution becomes more important for modern architectures to keep the growing number of processing units busy. The Cell architecture is well suited for task based execution [10] and CellSs [11] proposes a programming environment using annotations similar to OpenMP to schedule tasks to the available SPEs. [12] proposes a hardware accelerated support for dynamic task scheduling.

Adaptive methods have been used in many applications to handle irregularity. [13] shows the effectiveness of dynamic loop scheduling with adaptive techniques. [14] proposes an adaptive scheduler which uses runtime information to dynamically adapt the number of processors used to execute a job. Data structures to adapt the amount of work stolen for load balancing are proposed in [15] which enables the stealing of about half of the tasks from a given queue but the approach needs dynamically executed balancing steps. The 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.

6 Conclusions

We have presented an adaptive data structure which is used to implement adaptive task pools. The forest vector storing the trees of tasks introduces a small overhead in contrast to linear lists but it provides advantages for a task based execution. The tasks are stored in larger groups with a growing number of tasks while the groups shrink as the number of tasks shrinks. The size of the private area of tasks also dynamically depends on the number of task available for execution. Creating many fine-grained tasks actually leads to a benefit as the forest vector owner can access parts of it without locks. Also, the cost for task stealing is reduced as a single operation can move a large group of task at once with only one write access to a remote forest vector. The stolen tree contains, to some degree, related tasks which can reduce the penalty for stealing tasks.

Runtime experiments have shown that the adaptive task pool implementation can cope with a large number of fine-grained tasks but is still able to handle situations with a limited degree of parallelism. Non of the previous implementations could handle both situations equally well.

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

References

1. Hoffmann, R., Korch, M., Rauber, T.: Performance Evaluation of Task Pools Based on Hardware Synchronization. In: Proceedings of the 2004 Supercomputing Conference (SC 2004), Pittsburgh, PA, IEEE/ACM SIGARCH (November 2004)
2. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly, Sebastopol (2007)
3. Singh, J.P., Gupta, A., Levoy, M.: Parallel visualization algorithms: Performance and architectural implications. IEEE Computer 27(7), 45–55 (1994)
4. 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, Santa Margherita Ligure, Italy, pp. 24–36 (1995)
5. Hanrahan, P., Salzman, D., Aupperle, L.: A Rapid Hierarchical Radiosity Algorithm. In: Proceedings of SIGGRAPH (1991)
6. Xu, C., Lau, F.C.: Load Balancing in Parallel Computers: Theory and Practice. Kluwer Academic Publishers, Dordrecht (1997)
7. Kale, L.V., Krishnan, S.: CHARM++. In: Wilson, G.V., Lu, P. (eds.) Parallel Programming in C++, pp. 175–214. MIT Press, Cambridge (1996)
8. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu Jr., S., Steele G.L., Tobin-Hochstadt, S.: The Fortress Language Specification. version 1.0beta (March 2007)
9. 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.) OOPSLA, pp. 519–538. ACM, New York (2005)
10. IBM developerWorks Power Architecture editors: Unleashing the power of the cell broadband engine. Technical report, IBM Systems Group (2005)
11. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: SC 2006. IEEE, Los Alamitos (2006)
12. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. ACM SIGARCH Computer Architecture News 35(2), 162–173 (2007)
13. Banicescu, I., Velusamy, V., Devaprasad, J.: On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. Cluster Computing, The Journal of Networks, Software Tools and Applications 6, 215–226 (2003)
14. Agrawal, K., He, Y., Leiserson, C.E.: Adaptive work stealing with parallelism feedback. In: Yelick, K.A., Mellor-Crummey, J.M. (eds.) PPOPP, pp. 112–120. ACM Press, New York (2007)
15. Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: PODC, pp. 280–289 (2002)