Profiling of Task-based Applications on Shared Memory Machines: Scalability and Bottlenecks

Ralf Hoffmann and Thomas Rauber

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

Abstract. A sophisticated approach for the parallel execution of irregular applications on parallel shared memory machines is the decomposition into fine-grained tasks. These tasks can be executed using a task pool which handles the scheduling of the tasks independently of the application. In this paper we present a transparent way to profile irregular applications using task pools without modifying the source code of the application. We show that it is possible to identify critical tasks which prevent scalability and to locate bottlenecks inside the application. We show that the profiling information can be used to determine a coarse estimation of the execution time for a given number of processors.

1 Introduction

Due to the dynamic computation structure of irregular applications, task pools have been shown to be a well suited execution environment for these types of applications [1]. Applications decomposed into a large number of fine-grained tasks managed by task pools can be executed efficiently on a wide variety of parallel shared memory machines. This includes large systems like IBM p690 or SGI Altix2 with more than 32 processors but also smaller dual or multi-core SMP systems. The task pool implementation takes care of executing available task and storing newly created tasks to realize a dynamic task structure.

The granularity of the different tasks is a major factor for the resulting performance, since it partially determines the overhead introduced by the task management. It is however difficult to predict the performance and scalability of a specific application. It has been shown in our previous work [1] that this overhead can be reduced by using hardware operations for the task pool management, but the performance also depends on other parameters which cannot be improved easily. A limited number of available tasks, and therefore a larger waiting time for new runnable tasks, is an example of a limiting factor beyond the scope of the task pool implementation.

A detailed analysis of the internal task structure of an application is required to determine bottlenecks, to find scalability problems, and to suggest code improvements. For programs with a static task structures this can be done by analyzing the directed acyclic graphs (DAGs). But for irregular applications the

task graph is usually not known before execution, and even then it may only be known partially. Analyzing the source code and predicting the performance can be difficult for complex and irregular multi-threaded applications.

Applications designed to utilize task pools are completely independent from the actual task pool implementation. Therefore, a profiling task pool can be used to analyze performance characteristics of the application in the background. Possible information gathered in the profiling process include the task creation scheme (i.e., the task graph), statistics about the execution time of each individual task, task stealing operations for load balancing, and so on. In this paper we concentrate on gathering information about the task runtime and the time each processor waits for new tasks to identify bottlenecks in the application. The contribution of the paper is to propose a method to analyze the task structure of arbitrary applications utilizing task pools and to show that it is possible to predict the performance for a large number of processors by using profiling results from runs with a small number of participating processors. As case study the method is applied to the hierarchical radiosity from the SPLASH-2 suite [2].

The rest of the paper is organized as follows. Section 2 introduces the profiling method and Section 3 describes the data analysis. Section 4 presents a detailed case study. Section 5 discusses related work and Section 6 concludes the paper.

2 A profiling task pool

For a task based execution the application defines tasks by providing the operations executed by the tasks as functions. Figure 1 shows the generic structure of a task-based application. After creating initial tasks, depending on the actual input of the application, each processor executes the main loop and asks for tasks to execute until all tasks are finished. Each task is a sequential function, and it can create an arbitrary number of new tasks. The number of executable tasks varies over time, but usually there is a much larger number of tasks than processors. The internal implementation of the task pool is hidden but it is accessible by all threads via an application programming interface.

The granularity of the tasks depends on the actual computations done by the corresponding functions and it can have a large influence on the resulting performance. For example, many small tasks lead to frequent access to the task pool with a potentially larger overhead due to mutual exclusion while a small number of larger tasks can limit the available degree of parallelism.

The actual execution time (in the following referred to as *task size*) of each task instance is hard to predict since it depends on hardware parameters like CPU speed or memory bandwidth as well as input-dependent parameters like loop bounds. Another important factor is the time which threads have to wait for new runnable tasks (referred to as *task waiting time*). It is not easy to predict whether there are enough tasks available at any given time.

If an application does not scale well there are several issues to consider to improve the performance of the application. A limiting factor may be the task structure of the application, but also the task pool implementation may limit the

```
struct Task { Function, Argument };
// 1. Initialization phase (processor 1)
for (each work unit U of the input data)
   TaskPool.create_initial_task(U.Function,
      U.Argument);
// 2. Working phase
processor 1...p:
   loop:
      Task T ← TaskPool.get();
      if (T = ∅) exit;
      T.execute(); // may create new tasks
      T.free();
```

Fig. 1. Programming interface for task-based application.

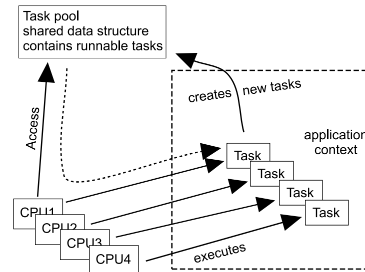

Fig. 2. Parallel execution scheme of a task-based application

scalability. But even if an application scales well on a given machine, it remains unclear whether it will achieve good speedups when using a larger number of processors or faster machines. To obtain the profiling information required to address these issues we select a well performing yet simple task pool implementation from previous work [1] and a modified version is used to gather statistical information about the task execution. Because the task pool appears as a black box to the application this does not require code changes in the application.

Figure 2 illustrates the parallel execution of a task-based application. Each thread accesses the task pool to obtain executable tasks. The code of the task is executed in the application context which can create new tasks for later execution. Besides the application context there is a task pool context which covers the task management, i.e. searching or waiting for new tasks. In this paper we concentrate on profiling the time spent in both contexts and show how this information can be used to identify problems within the application. The time spent in the application context is the execution time of the task code which we call the *task size*. This time represents the work done by the actual application. The larger the task size the more work is done in a single task, so this value helps to identify the task granularity of the application. The task size does not only provide information about the task granularity, it can also be used to obtain information about possible bottlenecks due to parallel execution. For example, if the granularity of a task is independent of the number of threads, but the actual execution times increase with the number of threads then this indicates non-obvious scalability problems. Reasons for such a behavior can be a higher number of cache misses, increased memory access time due to remote access, or higher lock contention. For each single task the profiling mechanism stores which function has been executed along with the corresponding execution time.

The time spent in the task pool context is the time needed for the task management including waiting for new executable tasks. We refer to the sum of these times as *waiting time*. This is the time which is not spent executing the actual application, so it indicates an overhead. An increase in the waiting time also indicates a scalability problem. Reasons can be: the threads access the

task pool too often causing mutual exclusion or there are not enough executable task available, so some threads need to wait. Some of the reasons for scalability problems can be addressed at the task pool level, e.g., by modifying the task pool implementation (as shown, for example, in [1]); other problems need to be addressed at the application level. In any case, detailed information about the waiting time can be used to find bottlenecks. For the waiting time the profiling mechanism measures the time spent in the task pool after finishing a task and before executing a new task. This waiting time is associated with the new task indicating that this task was not available early enough for execution.

3 Profiling Methodology

The execution of a task based application using the profiling task pool generates a large data set which needs to be analyzed for detailed information. In the first step we determine global statistical values which include the number of tasks executed, the total task size (i.e., total time spent in the application), the average task size, the total waiting time (i.e., total time spent outside the application), and the average waiting time.

This information allows first overall conclusions about the task pool usage for a specific application. If the number of tasks is small compared to the number of processors the load balancing effect of the task pool is limited. The waiting time can be considered as overhead as this is the time spent in the task pool and not in the application. If the waiting times are long compared to the task size, then this indicates that too few tasks are available for execution at some times.

For a detailed analysis we create task histograms (see Figure 3 for an example) which count the number of occurrences for every task size. Together with the waiting time these plots allow detailed statements about the performance impacts of the interaction between the application and the task pool. The important observations from the histogram can be summarized as follows:

- Large tasks mean a low overhead but possibly indicate limited parallelism or load imbalance.
- Many medium sized tasks suggest a good balance between high overhead and load imbalance.
- Many small tasks (i.e. a high occurrence of tasks on the left side of the histogram) indicate a large overhead in the task pool.
- An similar shape of the task size histogram for different number of processors indicate a suitable task structure, as the execution time of a task does not depend on the number of processors. Otherwise, memory or lock contention or cache invalidations are possible reason for unsatisfactory performance.
- The waiting time should always be as low as possible, i.e., most of the occurrences should be on the left-hand side of the histogram.
- The majority of the waiting times should be below the task size curve. Otherwise, the waiting time is more significant than the actual computation indicating a serious problem in the parallel application.

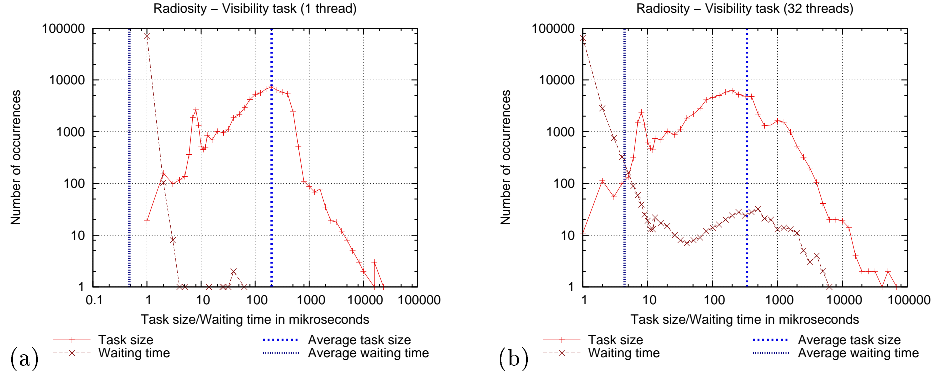

Fig. 3. Task histogram for the "visibility" task of the radiosity application using 1 (a) and 32 (b) threads on the Power4 system.

– Large waiting times, even if they occur rarely, indicate limited parallelism at some time inside the application.

In the following, we discuss the analysis of the profiling information for a selected task type to show how this information can be evaluated for an actual task. The information are gathered from an IBM p690 server which is a symmetric multiprocessor (SMP) with 32 Power4 processors running at 1.7 GHz. A more detailed analysis of the complete application is given in Section 4. Figure 3 shows the results of the data analysis for the hierarchical radiosity application for a specific task type using 1 and 32 threads. For single threaded execution the waiting time for most of the tasks is 1 (0 is counted as 1 due to the logarithmic scale), i.e., there is almost no time spent in the task pool. On average, a task takes $201\,\mu s$ to finish and there are only several hundreds out of about 70,000 tasks with a task size $\leq 5\,\mu s$. Taking all zero waiting times into account, the average waiting time is $\approx 0.5\,\mu s$ so the task size is roughly 400 times larger than the waiting time in the task pool on average. The maximum waiting time of $63\,\mu s$ is also much smaller than the average task size.

For 32 threads, Figure 3b indicates that there are no significant dependencies on the number of threads for this task type as the shape of the task size curve is similar to the single thread curve. All tasks take nearly the same time to complete when executing the application with 32 threads so the execution is not limited by memory bandwidth or cache size. On the other hand, the waiting time increases from 1 to 32 threads. The average waiting time is $4.4\,\mu s$ which is around 10 times more than for the single thread run, but the average task size increases only from $201\,\mu s$ to $340\,\mu s$. Even more important, the largest waiting time is $6319\,\mu s$ which is around 100 times more than for the single thread run. However, the majority of the waiting time is $\leq 1\,\mu s$ and the average task size is still almost 80 times larger than the waiting time on average. On this system, the large increase in the waiting time is not reflected by a performance decrease

Algorithm 1 Framework of the hierarchical radiosity application.

```
Phase 1:
  for all input patches P do
    insert P into BSP tree;
    create task REFINEMENT(P);
Phase 2:
  repeat
    for all patches P in BSP tree do
      create task RAY(P);
    execute tasks;
  until error small enough;
Phase 3:
  for all patches P in BSP tree do
    create task AVERAGE(P,average);
  execute tasks;
  for all patches P in BSP tree do
    create task AVERAGE(P,normalize);
  execute tasks;

Task REFINEMENT(P):
  compute form factor and refine recursively;
Task RAY(P):
  for all interactions I do
    if error(I) too large then
      refine element(I) and create interactions;
    else if unfinished(I) then
      create task VISIBILITY(I);
    else
      gather energy;
      if P is leaf then propagate to parent;
      else for each child C do
          create task RAY(C);
Task VISIBILITY(P):
  compute visibility for given interactions;
  continue task RAY(P);
Task AVERAGE( P, mode ):
  if P is leaf then average or normalize values;
  else for each child C do
      create task AVERAGE(C,mode);
```

as the maximum waiting time of $\approx 7\,ms$ is much smaller than the largest task which took almost $70\,ms$ to complete. It can be expected that on a larger or a faster system the problem may limit the scalability especially because such large waiting times occur several times. For this task type we can draw the following conclusions using the profiling information:

1. large task sizes in contrast to waiting times even for 32 processors indicate good scalability;
2. the overhead of the task pool is negligible;
3. there is a small increase in task size and a bigger increase in waiting time when more processors are used, so perfect scalability will not be reached especially for a larger number of processors;
4. the small number of very small tasks also indicates a suitable task structure;

4 A case study for performance prediction

In this section we describe how the task profiling can be used to predict the performance of an application. The case study is done by considering the *hierarchical radiosity application* [3] which is a global illumination algorithm that renders geometrical scenes by computing the equilibrium distribution of light. A hierarchical subdivision of the object surfaces is performed dynamically at runtime, and interactions between the surface elements representing the transport of light are evaluated. Parallelism is exploited across interactions and subdivision elements. The application is subdivided into four task types, see Algorithm 1 for an overview of the application. At the beginning the surfaces (or patches) of the initial scene are divided into sub-patches. This is done by the "refinement" task which can be executed in parallel for different patches. The computation is done by the "ray" task which calculates the energy exchange with other patches. This task can issue new "visibility" tasks and "ray" tasks to evaluate sub-patches. The "average" task post-processes the computed values.

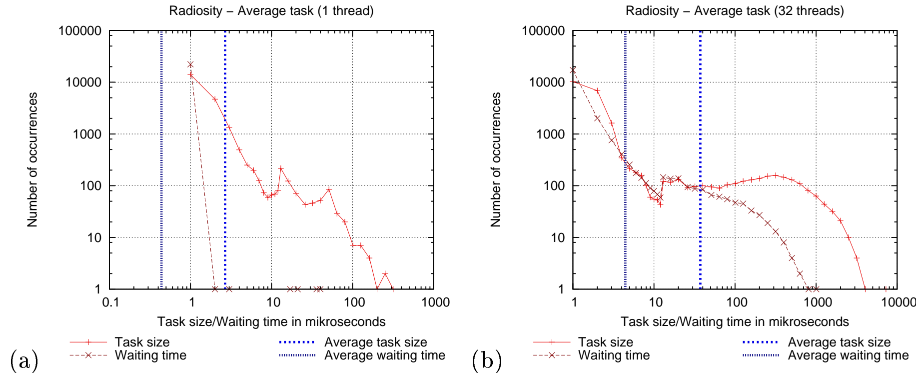

Fig. 4. Task size histogram for the "average" task of the radiosity application using 1 (a) and 32 (b) threads on the Power4 system.

4.1 Application evaluation

The implementation uses four different task types for different stages of calculating the resulting images of a given scene.

Visibility task. We already have investigated the "visibility" task in the previous section, see Figure 3. We have seen that this particular task performs well also for a large number of processors (32).

Average task. Figure 4 shows the task histogram for the "average" task. This task performs a post-processing step for averaging and normalizing the calculated radiosity values. The majority of the task sizes is very small, less than $10\,\mu s$. For one thread (Figure 4a), the average task size is $2.67\,\mu s$ and there are only a few tasks larger than $100\,\mu s$. The average waiting time is $0.44\,\mu s$.

For 32 threads, the shape of the task size graph changed much more than for the "visibility" task shown in Figure 3. There is now a significant number of task sizes larger than $100\,\mu s$. The average task size is with $37.35\,\mu s$ almost 14 times larger than in the single threaded run. Because the number of participating threads does not influence the granularity of this task type, this significant increase indicates a performance problem. The average waiting time is 10 times larger ($4.44\,\mu s$) in contrast to a single thread execution. This is 12% of the average task size which is a significantly larger fraction than for the "visibility" task considered in the previous section.

The conclusion is that this particular task type does not scale very well. The increase in waiting time is not extremely large but the fraction of the waiting time on the task size is large enough to influence the performance. More important is the significant increase of the task size which needs attention to improve the performance of the application.

Ray task. The "ray" task actually calculates the energy of the patches. The shape of the task histogram is similar to the histogram for the "average" task. When using a single thread (Figure 5a) the majority of the tasks are small (less than $10\,\mu s$), but there are also several larger tasks leading to an average task

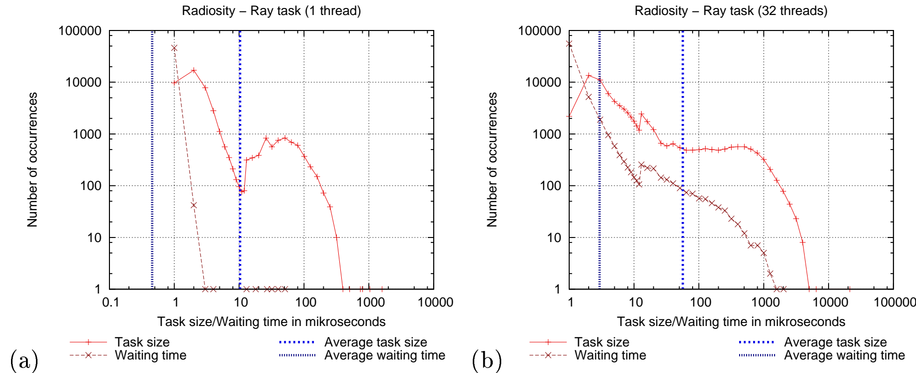

Fig. 5. Task histogram for the "ray" task of the radiosity application using 1 (a) and 32 (b) threads on the Power4 system.

size of $\approx 10.3\,\mu s$. The average waiting time is very small ($\approx 0.46\,\mu s$, $\approx 4\%$ of the task size). For 32 threads we see the expected increase in the average waiting time which is with $\approx 2.94\,\mu s$ only around 6.5 times larger. The average task size is ≈ 5.47 times larger ($56.43\,\mu s$) which is less than the increase in the task size for the "average" task, but it indicates a similar problem. The conclusion is that this task type works slightly better than the "average" task, but still has scalability problems due to the majority of small tasks and an increase in the task size when using more processors.

Refinement task. The "refinement" task is used to divide the patches of the scene into smaller sub-patches. We observe a different behavior (Figure 6) than for the other task types. For a single thread, no task is smaller than $11\,\mu s$ and the majority is $11 - 13\,\mu s$ but there are several larger tasks ($50 - 500\,\mu s$). The average task size is $12.24\,\mu s$. The waiting time is small, mostly less than $1\,\mu s$. The peak at around $10\,\mu s$ and the few larger waiting times up to $4500\,\mu s$ represent overhead in the task pool implementation. This task type is created and executed first, so the internal data structures to store the large number of tasks need to be created. The average waiting time is very small ($0.61\,\mu s$).

For 32 threads we observe major scalability problems. The average task size is more than 14 times larger ($172.77\,\mu s$), and the average waiting time is almost 27 times larger ($16.45\,\mu s$). The absence of very small tasks should be a good sign for good scalability, but the significant increase of the task size and waiting time indicates serious scalability problems for this task.

4.2 Performance prediction

For a coarse estimation of the execution time of the application for a specific number of processors, we use the profiling information from measurements with a smaller number of processors to extrapolate the task sizes and waiting times for a larger number of processors. The information for each pair of consecutive num-

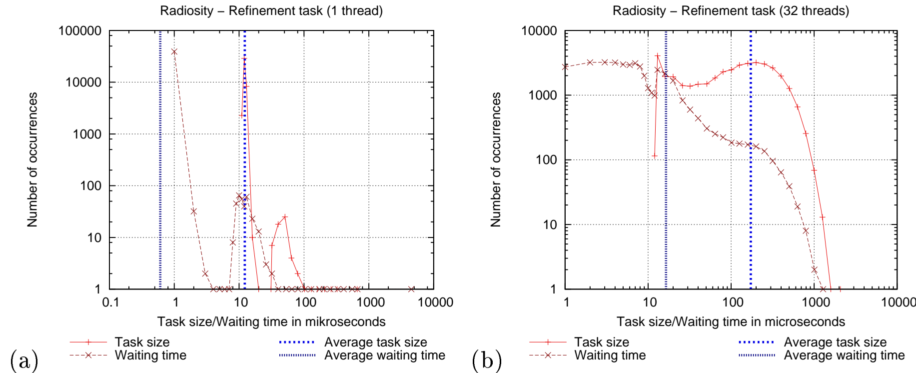

Fig. 6. Task histogram for the "refinement" task of the radiosity application using 1 (a) and 32 (b) threads on the Power4 system.

ber of processors is used to linearly extrapolate the task size and waiting time to the target number of processors. Several extrapolation values are combined to the final value by weighting each value by its distance to the target number of processors. The prediction is done by using only information gathered from the profiling process without considering details about the application. As tasks are arbitrary complex functions, the actual performance depends on many parameters unknown to the profiling task pool (like lock contention inside the task, memory bandwidth requirements, and so on). The estimated performance cannot be expected to be exact, but it can give an indication how well the application will scale.

Table 1 shows the estimated results and the actual results on the Power4 system for the computation time and the waiting time both for 32 processors. The estimated values are calculated by taking measured results for 1, 2, 4, 8 and 16 processors into account. Most values are underestimated or slightly overestimated, and the only specific value which is significantly overestimated is the execution time for the refinement tasks. It is around 66 percent above the actual value which is still acceptable considering the low assumptions made and the few information available.

Using these values the predicted speedup for 32 processors is 9.75 whereas the actual speedup is 7.87. The speedup for 16 processors is ≈ 9.5 so the estimation suggests that using twice as many processors does not give a significant benefit for the execution time. The predicted increase of the task sizes, waiting times and number of tasks helps pointing out which tasks need more attention than others.

5 Related work

Application profiling is a well known technique to analyze the performance of an application. Modern compilers support adding profiling code to the application

Table 1. Estimated combined execution and waiting time for 32 threads.

Radiosity Task type	Computation time		Waiting time	
	Estimated time in μs	Actual time measured in μs	Estimated time in μs	Actual time measured in μs
RAY	2,922,294.15	5,794,241.25	125,116.07	296,063.50
REFINEMENT	20,249,703.33	12,141,060.70	299,718.71	703,022.05
AVERAGE	1,751,265.41	1,556,440.05	158,188.50	123,334.20
VISIBILITY	22,136,753.20	38,105,390.00	51,363.556	361,316.50
Sum	47,060,016.06	57,597,132.00	634,386.84	1,483,736.25

to be able to find time consuming parts of an application. Tools like VAMPIR [4] allow the evaluation of such information.

The method presented in [5] does not depend on compile time instrumentation but uses the binary code to profile the application and predict the performance. The method profiles memory accesses to predict the estimated execution time for larger inputs for sequential execution. [6] tries to predict the performance of a selected application for parallel execution even on future architectures but requires detailed analysis of the actual application.

[7] proposes a framework to automatically tune an application. Similar to ATLAS [8], but more generic, the framework is able to select an efficient implementation of certain library functions used by the application. A small amount of source code changes are required and the optimizations are application specific while we are trying to optimize generic task-parallel applications.

As a similar approach to analyze an application and identify performance problems, [9] proposes a method to use a simulator to obtain memory access information (cache misses etc.) and suggest improvements. In our work we are trying to avoid the overhead of a simulator and source code modifications and we also consider the impact of contention which are not modeled by cache statistics. [10] proposes a method to profile parallel applications using TAU. Similar to our work, TAU profiles different contexts (phases) but this requires instrumentation of the source code.

6 Conclusions

The profiling methods proposed in this paper allow to study the behavior of irregular applications and identify scalability problems without code changes or even recompilations of the actual application. Splitting the execution time into the application context and the task pool context makes it possible to evaluate different task types separately and even to consider single tasks. The application context models the actual computations but also covers possible contention inside single tasks. The task pool context covers the available parallelism and overhead of the task based execution.

The proposed method allows the investigation of specific tasks for a given number of processors to identify possible scalability problems inside the task and

to indicate missing parallelism inside the application or a too large overhead from the task pool implementation. The isolated examination of single tasks allows us to point out specific tasks which indicates problems and also to propose changes to improve the scalability.

Similar profiling information are otherwise only available by changing the application or recompile it to use profile information from compilers or utilize hardware counters which is not always available or wanted.

Acknowledgments. We thank the NIC Jülich 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'04), Pittsburgh, PA, IEEE/ACM SIGARCH (November 2004)
2. 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 (1995) 24–36
3. Hanrahan, P., Salzman, D., Aupperle, L.: A Rapid Hierarchical Radiosity Algorithm. In: Proceedings of SIGGRAPH. (1991)
4. Brunst, H., Kranzlmüller, D., Nagel, W.E.: Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz. In Juhasz, Z., Kacsuk, P., Kranzlmüller, D., eds.: DAPSYS. Volume 777 of Kluwer International Series in Engineering and Computer Science., Springer (2004) 93–102
5. Marin, G., Mellor-Crummey, J.: Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In: Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems - Sigmetrics 2004, New York, NY (June 2004) 2–13
6. Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., Gittings, M.: Predictive performance and scalability modeling of a large-scale application. In: Proceedings of the 2001 Supercomputing Conference (SC'01), IEEE/ACM SIGARCH (2001) 37
7. Tapus, C., Chung, I.H., Hollingsworth, J.K.: Active Harmony: Towards Automated Performance Tuning. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (2002) 1–11
8. Whaley, R.C., Dongarra, J.J.: Automatically Tuned Linear Algebra Software. Technical report, University of Tennessee (1999)
9. Faroughi, N.: Multi-Cache Profiling of Parallel Processing Programs Using Simics. In Arabnia, H.R., ed.: Proceedings of the PDPTA, CSREA Press (2006) 499–505
10. Malony, A., Shende, S.S., Morris, A.: Phase-Based Parallel Performance Profiling. In Joubert, G.R., Nagel, W.E., Peters, F.J., Plata, O.G., Tirado, P., Zapata, E.L., eds.: Proceedings of the PARCO. Volume 33 of John von Neumann Institute for Computing Series., Central Institute for Applied Mathematics, Jülich, Germany (2005) 203–210