Formalizing Bottlenecks in Task-Based OpenMP Applications

Shajulin BENEDICT [a], Michael GERNDT [b] and Diana-Mihaela GUDU [b,1]
[a] *HPCCLoud Research Laboratory, SXCCE, Anna University, India*
[b] *Informatics I10, Technische Universität München, Germany*

Abstract.
Task support was introduced into OpenMP to address irregular parallelism in shared memory architectures. Creating tasks that are extremely fine granular in applications, however, impedes performance. In this paper, we present a methodology for analyzing the performance of task-based OpenMP programs and its implementation in Periscope. We concentrate in this paper on the newly formulated high-level performance properties that formalize typical performance bottlenecks of task-based programs. In addition, we report on the experimental results when applying Periscope to the codes in the Barcelona OpenMP Tasks Suite (BOTS).

Keywords. OpenMP, Parallel Programming, Performance Analysis, Task Parallelism, Tools

1. Introduction

OpenMP, with its simple directives-based approach, facilitates parallelization of applications for shared memory systems. It is receiving more importance in the pace to exascale since current and future machines are composed of nodes with multi- and manycore processors. Even for attached accelerators, e.g., GPGPUs or the Intel Xeon Phi, OpenMP may be in the form of OpenACC and hence, OpenMP is seen as a suited programming interface.

OpenMP 3.0 introduced task-based parallel programming in OpenMP. This extension enables programmers to parallelize applications via defining independent parallel work units (tasks). Execution of such work units aims to facilitate the implementation of application with irregular parallelism. However, in most cases, task-based scientific applications which are typically recursive in nature, have performance issues due to creating many fine granular tasks. Other performance issues are the overhead for task creation and scheduling, load imbalance that might result from variations in the computational complexity of the task, as well as reduction in the data locality when tasks are distributed to processors that are far away from the memory with the required data.

Performance analysis tools are required in tuning task-based applications and for identifying performance deficiencies, thus giving hints to the programmer where the pro-

[1]The research has been partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n 288038 (AutoTune Project, www.autotune-project.eu), the Collaborative Research - InvasIC, and the HPCCLoud Research project under CIM-Returning Experts programme, Germany.

gram can be improved. Periscope, an online-based performance analysis tool using analysis agents, automatically identifies performance issues, points the user to the responsible code region, and indicates the severity of the problem. In this paper, we discuss an extension of Periscope for analyzing OpenMP task-based programs. We concentrate on the formalization of typical performance issues in such programs and show experimental results for the Barcelona OMP Task Suite (BOTS) benchmarks, namely, sort, fft, alignment, nqueens, and fib. The formalized performance properties are not only the base for Periscope's automatic search but can also be used in other performance analysis tools.

The rest of the paper is organized as follows. Section 2 presents existing work. Section 3 explains Periscope architecture, code instrumentation, search strategy, and formalized performance properties. Section 4 discusses experimental results. Finally, Section 5 presents a few conclusions.

2. Related Work

Recently, research on OpenMP tasking has emerged within the parallel programming community.

A few studies have developed applications based on OpenMP tasks. For instance, the authors in [1] have exploited OpenMP 3.0 in some well-known applications as benchmarks. The task-based applications, however, are neither extensively developed nor widely used.

A few proposals are in the context of designing OpenMP tasks [7]. [13] emphasizes the necessity of an error handling mechanism in the OpenMP standard. In [14], the authors have highlighted the importance of transactional memory when OpenMP is run. Similarly, Christian et al [5] has studied the effects of data locality in task-based parallel programs on multi-core architectures.

Apart from the research studies on the design of the task concept in OpenMP, performance studies of OpenMP tasks are also in progress. The work in [2] illustrates the effects of creating and scheduling an excessive number of tasks. By providing some cut-off mechanisms, the authors have controlled the number of tasks created at runtime. [6] has studied the importance of the *if clause* in tasks. A manual performance study for an Unbalanced Tree Search (UTS) application [21], a benchmark suite using OpenMP tasks, demonstrates the load imbalance and the overheads due to the creation of tasks.

Research work on performance analysis tools emerges with different innovative perspectives, say, trace-based, profile-based, statistics-based, knowledge-based, automatic, centralized or decentralized, and so forth [4,10,11,12,9,15,16]. The proliferation of OpenMP task-based research has urged the tool developers to provide solutions about how to measure and sense performance data for OpenMP tasks. For instance, the ompP tool [8], a profiling tool for OpenMP applications, studied the issues for monitoring OpenMP tasks while instrumenting and measuring the code regions; the Intel Thread Profiler finds where OpenMP support performance analysis and points out code regions where tasking could be used. In addition, [17] has proposed three performance problems related to OpenMP tasks after the programs were instrumented. This paper aims at an online, user-friendly, and automatic detection of performance bottlenecks based on a formal bottleneck specification.

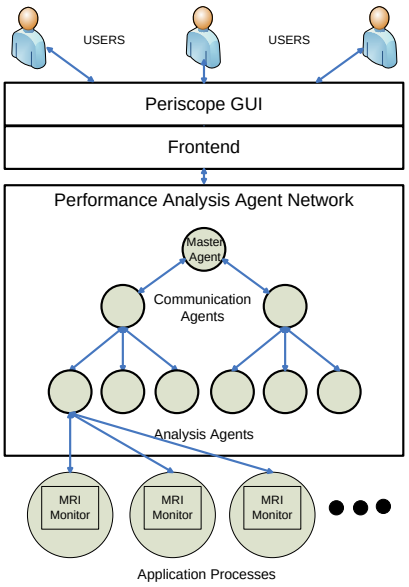

Figure 1. Periscope Architecture

3. OpenMP Task-based Performance Analysis

Performance analysis is an imperative step in the development of large-scale scientific applications, especially when money matters. Because machines and applications can have a variety of run time performance problems, the user opts to investigate into the responsible code regions using tools. In this section, we discuss the modifications carried out in the Periscope architecture along with its brief introduction and the performance analysis methodology for OpenMP tasks.

Periscope, a performance analysis tool, is devised such that the users can investigate performance problems in a distributed manner on the fly [18]. The performance data are measured, evaluated, and reported back to the user at the end of the application run while the application is running. The advantage of the tool is that it need not do any post-processing for the performance data. It can be used in interactive application runs as well as in batch runs.

Periscope architecture consists of four major entities as shown in Fig. 1. They are i) User-Interface, ii) Frontend, iii) Analysis Agent Network, and iv) MRI monitor. The steps involved in while finding the performance problems by the Periscope toolkit and the responsibilities of the entitites are listed as follows:

1. The user has to compile their application with a *psc_instrument* script. The script automatically instruments the source code and compiles them with its corresponding compiler. The instrumented application is then linked with the mrimonitoring library of Periscope to create the application's executable.
2. The analysis can be started either through the normal command-prompt window or using the User-Interface entity of the Periscope architecture. The advantage of using the entity are as follows: i) the user can index the code region when the corresponding performance properties are clicked, ii) highlight syntaxes, iii) cluster the found performance properties, and so forth.

3. For the analysis purpose, the user has to start the *Frontend* entity. The user can specify all the necessary options, such as, the search strategy, the number of processes, and so forth while starting the *Frontend* entity. Otherwise, the default options are automatically initialized. The *Frontend* starts the application and analysis agents; it can also restart them if the search is incomplete.
4. Internally, the *Analysis agent network*, which is a combination of the master agent, communication agents, and the analysis agents, reports on the found properties to the frontend. The search for the performance properties is based on a master agent for management purposes, analysis agents for finding problems on individual processors, and a few other agents responsible for communication. Based on the search strategy domain and its corresponding properties, the analysis agents perform searches using the Monitoring Request Interface (MRI).

Searching for a performance bottleneck with Periscope can be exclusively accomplished for different problem domains, such as, Memory, MPI, OpenMP, or scalability issues. All the performance properties included in the search strategies are bound to the phase regions. The *Stall Cycle Analysis (SCA) Search Strategy* searches for the performance problems on applications due to cache miss stalls, pipeline stalls, or so forth. These stalls might prevent the next instruction in the instruction stream from being executing during its designated clock cycle. The strategy iteratively refines its search if necessary. The *MPI Search Strategy* is dedicated to reveal the communication delays, early/late sending problems, and imbalances on MPI-based scientific applications. This strategy does not iteratively search for the issues. Similar to the MPI strategy, the *OMP search strategy* [19] finds performance problems in a single run of the phase region. With respect to OpenMP, the analysis agents search for performance properties related to i) extensive startup and shutdown overhead for fine-grained parallel regions, ii) load imbalance, iii) sequentiality, and iv) synchronization. The *Scalability-OMP search strategy* [20] is designed for the OpenMP-based applications. This strategy searches for the scalability-based performance properties, such as, Low Speedup, Linear Speedup failed for the first time, and so forth by running the analysis n times with 2^n number of threads for each run. The *ALL Search Strategy* combines all the search strategies to find their respective performance properties iteratively. Currently, this strategy does not support the scalability analysis. Periscope uses OMP search strategy to investigate task-related performance issues. A more detailed description about the search strategy can be seen in [19].

The basis for the automatic search of performance problems with Periscope is their formalization. Typical performance problems are entered into the knowledge base of Periscope in the form of performance properties. Performance properties determine runtime measurements required to detect the problem and calculate and report its severity. A detailed description of the performance properties for OpenMP 2.5 can be found in [19,20].

In the following, we present the new performance properties for the task support in OpenMP that were added to Periscope's OMP search strategy.

3.1. Overhead Due To Task Creation

In OpenMP 2.5, each parallel directive creates implicit tasks. But with OpenMP 3.0, users can specify explicit tasks which will be executed by a team of threads. Creating

and scheduling of tasks are done at runtime. Whenever a task construct is encountered, the thread creates a new task from the task region together with its data environment. In addition, the initialization of task private data and clauses is executed. If a programmer mandatorily or recursively creates too fine granular tasks, task creation time can lead to overhead. The *Overhead Due To Task Creation* property is designed to expose overhead due to the creation of tasks.

The significance of the problem is expressed in terms of its *Severity* relative to the entire phase (*phaseCycles*)[2]. To calculate the severity of the *Overhead Due To Task Creation property* in process i, we measure the task creation time (TC_t) in each thread of the process and calculate it using the formula given below:

$$Severity_i(reg) = \frac{\max\limits_{0 \le t \le n-1} TC_t(reg)}{phaseCycles} * 100 \tag{1}$$

where, t indicates the thread number, *reg* indicates the region name, and n denotes the number of threads. We assume that the task creation time for a thread and a region is given in cycles as the phase execution time.

3.2. Too Fine Task Granularity

Writing extremely fine granular task regions in an application, causes overhead due to creating and scheduling tasks. That is, the possibility for overheads is high in a region if there are fine granular tasks in an application. This property finds the task regions that are having very scanty computational time. The threshold for having fine granularity in tasks is contrived based on the empirical experiments conducted by the authors on various OpenMP programs. The severity of the property is calculated as in Equation 1.

3.3. Imbalanced Task Region

To reduce the severity of the *Overhead Due To Task Creation property* and the *Too Fine Task Granularity property*, there are several options:

1. coarsening task regions
2. combining two or more tasks, or
3. executing tasks in a serial fashion.

An application, which incompetently exploits those options, however, suffers from load imbalance. This property reports on the severity of load imbalance for a task region (see Equation 2) when executed within a team of threads.

$$Severity_i(reg) = \frac{\sum\limits_{t=0..n-1} (TBMAX(reg) - TB_t(reg))/n}{phaseCycles} * 100 \tag{2}$$

[2]The term phase is used in Periscope for an iteration of the main loop in the program, typically the time stepping loop in scientific applications.

where, $TB_t(reg)$ is the taskbody execution time of task region *reg* in thread t, $TBMAX(reg)$ is the maximum value of task body execution time TB from all threads, and n is the number of threads per process.

3.4. Imbalance - Number of Tasks smaller than Number of Threads

Imbalance in a task region can happen when the number of tasks is smaller than the number of threads. This property identifies task regions which have less tasks than threads and reports them on to the user with the severity as in Equation 3. However, the severity will not be reported for this property if more tasks than threads were generated.

Thus the property first checks the condition $NT(reg) < n$. Only if the condition is fulfilled the severity is computed by the following formula:

$$Severity_i(reg) = \frac{\sum_{t=0..n-1} (TBMAX(reg) - TB_t(reg))/n}{phaseCycles} * 100 \tag{3}$$

3.5. Imbalance - Uneven Distribution of Tasks

The property, *Imbalance - Uneven Distribution of Tasks property*, identifies the situation where threads do not execute the same number of tasks. The property uses the following condition to check this fact:

$$Condition = \max_{t=0..n-1} (ETMAX - ET_t) > threshold * NT(reg) \tag{4}$$

where $ETMAX$ is the maximum number of tasks executed by any thread, ET_t is the number of tasks executed by a thread, and $NT(reg)$ is the total number of tasks created.

For the severity we use formula 2. The severity would be low if the different number of executed tasks does not lead to a significant imbalance, may be, because the thread with the lower number of tasks had more compute intensive tasks to execute.

3.6. Empty Task in Task Region

According to the OpenMP 3.0 standard, a thread can switch from one task to the other if it encounters a scheduling point. A task region with an empty loop body creates empty tasks. Empty tasks, hence, can lead to switches between tasks each and every time the thread creates an empty task. If there are too many empty tasks in a parallel region, the task creation overhead for those tasks is significant.

Thus, the property first checks that the created tasks are empty tasks based on static information about the task region. The instrumenter flags a task region as having an empty task body.

The severity can be computed from the task creation times of the process' threads.

$$Severity_i(reg) = \frac{\max_{t=0..n-1} (TC_t(reg))}{phaseCycles} * 100 \tag{5}$$

The severity does not take into account the overhead induced by task switching due to empty tasks. This overhead cannot be measured by the monitoring system, since it manifests in the pure switching time and other effect, such as a different cache behavior.

4. Experimental Results

This section presents the results from applying Periscope with the extended OpenMP search strategy to the task test codes in BOTS. The measurements were carried out on the SuperMUC's fat nodes to evaluate scalability up to 40 threads.

4.1. Benchmarks

For our tests, we used the Barcelona OpenMP Tasks Suite (BOTS) benchmarks, designed specifically for exploiting irregular and recursive task parallelism. The BOTS suite of benchmarks consists of a collection of applications written in OpenMP C-based language using task constructs. All benchmarks have both tied and untied versions. Some benchmarks provide the option of using a cut-off mechanism to control the task granularity. More details on the BOTS test suite, along with some example performance results, can be found at [3].

In our study, we have selected only a few benchmarks because the trends in the performance measurement are consistent for all benchmarks. We have used the tied versions of the following applications: Alignment (both *for* and *single* versions), N Queens, Sort and Fib.

4.1.1. Alignment

Alignment, one among the BOTS benchmarks, has both the *single* construct and *for* construct versions of OpenMP tasks. The benchmark aligns protein sequences iteratively using the Myers and Miller algorithm. For each pair of sequences in the input file, the algorithm scores the alignments and the one with the best score is finally considered. The benchmark uses a dynamic algorithm to score its results. The benchmark is implemented by a for loop that processes in each iteration a pair of protein sequences inside a single task construct. Depending on the benchmark version, the for loop is inside a *single* construct or parallelized with a *pragma omp for* construct.

Applying Periscope on the Alignment benchmark revealed that the benchmark creates quite coarse tasks. The application reports low or no severities for the fine granularity property (below 4%), with a slight increase with the number of threads. Consequently, the *Imbalance in task region* property is found with high severities.

4.1.2. N Queens

N Queens, a well-known computational problem, provides solutions to place n queens on a $n \times n$ chessboard. The placement should abide the rules of the chessboard standards, such that none of the queens can attack each other. The benchmark creates tasks for each step of the solution. The search for the solution is done by a backtracking search algorithm with pruning. The benchmark has one recursive task construct, *threadprivate* variables and *critical* constructs. The state of the algorithm needs to be copied into each

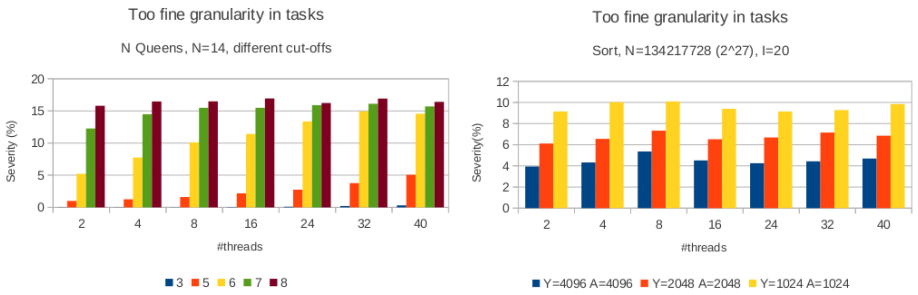

Figure 2. *Too fine granularity in tasks* property for *N Queens* and *Sort*, with different cut-off values. *N Queens'* cut-off mechanism is depth-based (bigger depth means finer tasks), while *Sort*'s is array length based (smaller length means finer tasks).

newly created task so they can proceed, introducing additional synchronization to maintain the parent state alive.

Furthermore, the benchmark implements a depth-based cut-off mechanism. More specifically, after a certain level, the application will not generate more tasks, to avoid the creation of very fine granular tasks.

Figure 2 shows the effect of using different cut-off values in the *N Queens* and *Sort* benchmarks on the severity of the property *Too fine granularity in tasks*. The tests for *N Queens* were performed for $N = 50$ with different manual cut-offs. A higher cut-off depth means that more numerous and smaller tasks are created recursively. Periscope detects an increase in severity with the increase in cut-off depth. No significant variations are observed for the task granularity with respect to the number of threads.

4.1.3. Sort

The Sort benchmark sorts n random numbers using the merge-sort algorithm. The array containing the random numbers is divided in two halves. Then, each independent work unit is sorted recursively. After sorting those numbers, the benchmark uses a divide-and-conquer algorithm to merge them in parallel. The sorting and merging operations are supported by OpenMP tasks.

The application uses several adjustable cut-off values: a sequential quick-sort cut-off, defining the length of the array when the merge-sort switches to a sequential quick-sort; a sequential merge cut-off, defining when the parallel merge stops creating new recursive tasks and switches to a sequential merge; an insertion sort cut-off, defining when the recursive quick-sort uses insertion sort. In our tests, we used a fixed insertion cut-off $I = 20$, while varying the merge cut-off Y and the quick-sort cut-off A. For consistency and simplicity reasons, Y and A are always equal.

Similar to *N Queens*, the analysis for *Sort* unveils a visible increase in severity of *Fine granularity in tasks* property with the decrease in cut-off value, as shown in Figure 2.

4.1.4. Fib

The Fib benchmark is a simple program with nested task constructs within a *parallel* and *single* OpenMP constructs to compute the n-th Fibonacci number. It includes versions that use a depth-based cut-off as well.

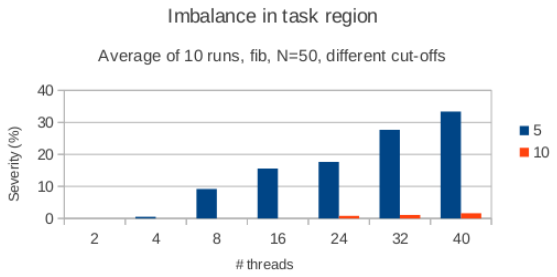

Figure 3. *Imbalance in task region* for *Fib*, with different cut-off values.

The results for *fine granularity* are consistent with *N Queens* and *Sort*: the severity of the property increases with the cut-off depth.

Intuitively, increasing the granularity of tasks decreases the imbalance, and vice versa. Figure 3 illustrates this fact for the *Fib* benchmark. Using a cut-off depth of 5 creates few coarser tasks, leading to a high imbalance, while using a cut-off depth of 10 leads to the creation of many finer tasks, which can be better distributed among the threads. Moreover, in the case of coarse granularity (cut-off 5), the number of tasks is quite small, causing a higher imbalance when increasing the number of threads, as some threads could even remain idle during execution. To tackle the non-deterministic distribution of tasks to threads, we performed 10 runs for each test and averaged the severities of the property *Imbalance in task region*.

5. Conclusions

OpenMP tasks provide new features to the programmer to efficiently handle irregular parallelism in their applications. Performance analysis tools, recently, have started providing solutions to reconcile with OpenMP tasks.

In this paper, we presented the formal definitions of performance problems in task-based parallel programs and the implementation aspects of these definitions in Periscope. The formalized performance properties can be used in any performance analysis tool. Our implementation was demonstrated with the BOTS benchmarks, a benchmark suite for OpenMP tasks. The test results showed the severity of OpenMP task-based performance properties for code regions of applications.

References

[1] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In Proc. of Int. Conf. on Parallel Processing, pages 124-131, 2009.

[2] Alejandro Duran, Julita Corbal, and Eduard Ayguade An adaptive cut-off for task parallelism. In Proc. of the 2008 ACM/IEEE conference on Supercomputing (SC '08). IEEE Press, Piscataway, NJ, USA , pages 1-36, 2008.

[3] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade, Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In proceedings of International Conference on Parallel Processing IEEE, ICPP'09, pages 124-131, 2009.

[4] Andreas Knupfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Muller and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Proc. of the 2nd Int. Work. on Parallel Tools for HPC, HLRS, Stuttgart, pages 139-155, Springer Publications, July 2008.

[5] Christian Terboven, Dirk Schmidl, Tim Cramer, Dieter an Mey, Assessing OpenMP Tasking Implementations on NUMA Architectures, in Lecture Notes in Computer Science, Volume 7312, pp 182-195, 2012.

[6] Eduard Ayguade, James Beyer, Alejandro Duran, Roger Ferrer, Grant Haab, Kelvin Li, and Federico Massaioli. An extension to improve OpenMP tasking control. In Lecture Notes in Computer Science, IWOMP 2010, Springer Berlin / Heidelberg, pages 56-69, 2010.

[7] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Xavier Teruel, Priya Unnikrishnan, Federico Massaioli, and Guansong Zhang. The Design of OpenMP Tasks. In IEEE Trans. on Parallel and Distributed Systems, Vol. 20, No. 3, pages 404-417, 2010.

[8] Furlinger, Karl and David Skinner Performance Profiling for OpenMP Tasks In Lecture Notes in Computer Science, Volume 5568, Springer Berlin / Heidelberg, pages 132-139, 2009.

[9] Luiz DeRose, Bernd Mohr, and Seetharami Seelam. Profiling and Tracing OpenMP Applications with POMP Based Monitoring Libraries. In Euro-Par 2004 Parallel Processing, Volume 3149 of LNCS, pages 39 to 46. Springer, 2004.

[10] Markus Geimer, Felix Wolf, Brian J.N. Wylie, Erika Abraham, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. In Concurrency Computation:Practice and Experience, Wiley InterScience, Volume 22 Issue 6, Pages 702-719, 2010.

[11] Michael Gerndt and Edmond Kereku. Search strategies for automatic performance analysis tools. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, Euro-Par 2007, Volume 4641 of LNCS, pages 129–138. Springer, 2007.

[12] Michael Gerndt and Edmond Kereku. Automatic memory access analysis with Periscope.In ICCS 2007, Volume 4488 of LNCS, pages 847 to 854. Springer, 2007.

[13] Michael Wong, Barna L.Bihari, Bronis R. de Supinski, Peng Wu, Maged Michael, Yan Liu, and Wang Chen A Case for Including Transactions in OpenMP. In Lecture Notes in Computer Science, IWOMP 2010, Springer Berlin / Heidelberg, pages 149-160, 2010.

[14] Michael Wong, Michael klemn, Alejandro Duran, Tim Mattson, Grant Haab, Bronis R. de Supinski, and Andrey Churbanov, Towards an Error Model for OpenMP. In Lecture Notes in Computer Science, IWOMP 2010, Springer Berlin / Heidelberg, pages 70-82, 2010.

[15] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K.Kunchithapadam, and T.Newhall. The Paradyn parallel performance measurement tool. IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995.

[16] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. International Journal of High Performance Computing Applications, Volume 20 Issue 2, pages 287 - 311, May 2006.

[17] Schmidl D., Phillippen P., Lorentz D., Rossel C., Geimer M., Dieter an Mey, Mohr B., Wolf F., Performance Analysis Techniques for Task-based OpenMP Applications, in Proceedings of the 8th international conference on OpenMP in a Heterogeneous World, IWOMP 12, pp. 196 - 209, 2012.

[18] Shajulin Benedict, Matthias Brehm, Michael Gerndt, Carla Guillen, Wolfram Hesse and Ventsislav Petkov. Automatic Performance Analysis of Large Scale Simulations. In PROPER 2009, LNCS, Springer Publishers, Vol 6043, pages 199-207, 2009.

[19] Shajulin Benedict and Michael Gerndt. Automatic Performance Analysis of OpenMP Codes on a Scalable Shared Memory System using Periscope, Applied Parallel and Scientific Computing, Lecture Notes in Computer Science, Volume 7134, pp 452-462, 2012.

[20] Shajulin Benedict and Michael Gerndt. Scalability and Performance Analysis of OpenMP codes using the Periscope Toolkit submitted in Computing and Informatics 2011.

[21] Stephen L. Oliver and Jan F. Prins. Comparison of OpenMP 3.0 and other Task parallel frameworks on Unbalanced Task Graphs. International Journal of Parallel Programming, Vol 38, pages 341-360, 2010.