On the Instrumentation of OpenMP and OmpSs Tasking Constructs

Harald Servat[1,2], Xavier Teruel[1], Germán Llort[1,2], Alejandro Duran[1,3], Judit Giménez[1,2], Xavier Martorell[1,2], Eduard Ayguadé[1,2], and Jesús Labarta[1,2]

[1] Barcelona Supercomputing Center
[2] Universitat Politècnica de Catalunya
[3] Intel Corporation
c/Jordi Girona, 31 08034 - Barcelona, Catalunya, Spain

Abstract. Parallelism has become more and more commonplace with the advent of the multicore processors. Although different parallel programming models have arisen to exploit the computing capabilities of such processors, developing applications that take benefit of these processors may not be easy. And what is worse, the performance achieved by the parallel version of the application may not be what the developer expected, as a result of a dubious utilization of the resources offered by the processor.

We present in this paper a fruitful synergy of a shared memory parallel compiler and runtime, and a performance extraction library. The objective of this work is not only to reduce the performance analysis life-cycle when doing the parallelization of an application, but also to extend the analysis experience of the parallel application by incorporating data that is only known in the compiler and runtime side. Additionally we present performance results obtained with the execution of instrumented application and evaluate the overhead of the instrumentation.

1 Introduction

Current supercomputers offer large computational power by connecting multiple computing nodes using a fast interconnection network. According to the TOP500 list [5], most of the supercomputers use multicore processors which increase the inter-node parallelism. It is not surprising to observe that parallelism has currently hit the desktop segment. However, all the compute power offered by these processors needs some guidance to achieve real parallel execution. And not only this, but guaranteeing that the parallel execution is taking real benefit of the system resources is also an important task.

OpenMP* [7] is a widely known shared memory parallel programming model that allows implementing parallel applications by using a set of compiler directives. The OpenMP runtime deals with the parallel thread execution (including fork, execution and join of the slave threads) and offers the user incremental parallel development by adding the compiler directives gradually. Other parallel programming models may require the developer a major adaptation of the

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 414–428, 2013.

application (for instance, MPI* [14]) or deal with the thread management (if using the pthread library). In addition, OpenMP allows to express irregular parallelism through a set of new constructs: the OpenMP Tasking constructs (which includes `task` and `taskwait` since version 3.0 and `taskyield` since version 3.1). A task is a unit of parallel work used to express unstructured parallelism that will be executed by one of the threads at a time, but different parts of a task may be executed by different threads.

OmpSs [6] is a parallel programming model based on the OpenMP standard that extends the OpenMP Tasking constructs to support asynchronous task parallelism and execution on heterogeneous devices. Asynchronous parallelism is enabled by the use of data-dependencies between the different tasks of the program (by extending the OpenMP task construct with `input`, `output` and `inout` clauses with respect to the variables used in the task). These data-dependencies are annotated in a task dependency graph created at runtime. The OmpSs runtime starts scheduling tasks that have their dependencies satisfied (i.e., their required data is ready). Then, as soon as tasks finish, the dependency graph gets updated allowing the dependent tasks to become ready for execution.

Performance tools are pieces of software devoted to analyze the performance of applications by looking for bottlenecks. Among the variety of performance tools, Paraver [22] is a flexible parallel program visualization and performance analysis tool. Its key features include: detailed quantitative analysis of program performance, concurrent comparative analysis of different traces, support for mixed message passing and shared memory applications, and customized semantics of the visualized information.

In this paper, we present the result of the synergy between OmpSs runtime and a Paraver trace generation tool. We not only demonstrate the utility of being capable of emitting the internal values of the OmpSs runtime into tracefiles, but we also improve the performance analysis life-cycle by providing an integrated mechanism to gather the application performance data. We present a new display mechanism for tasking constructs in the visualization tool that facilitates the understanding of the execution of the application from the task perspective. We also evaluate the overhead of our proposal by using a set of benchmarks. And, finally, we demonstrate the usefulness of this work by analyzing an application.

The remainder of this paper is organized as follows: we present a summary of the tools involved in this document in Section 2. Section 3 follows, where we describe our implementation. Section 4 describes the experimental results using several benchmarks. We discuss the related work in Section 5. Finally, we draw some conclusions and present future directions of this work in Section 6.

2 Background

Extrae [1] is a performance extraction tool that generates Paraver [22] tracefiles. Extrae requires a two-stage execution process. The first step gathers data like hardware performance counters and source code references from multiple parallel runtimes (including OpenMP, MPI, CUDA* and pthread) by using either instrumentation or sampling on compiled optimized binaries as long as the

application executes. The second step generates the tracefile and occurs after the application execution. The information gathered by Extrae depends on the parallel runtime and it is only limited by the available documentation.

A Paraver tracefile is a container of timestamped records related to the actual execution of an application. The performance information is mapped into the application resources that were assigned in the execution up to three levels (namely: application, process and thread). Among the types of records that can be stored in a tracefile, we focus on two of them: events and relationships. Events are punctual information that occurs in an object (thread or process) of the application. Typical events include entry and exit points of user routines, hardware counter values, callstack information, among others. On the other side, relationships refer to MPI point-to-point communications between two partners of an application. Information contained in a relationship includes: the two objects involved, the timestamps for the point-to-point operations, the size and tag of the communication.

The OmpSs programming model is built on top of the Mercurium compiler [2] and the Nanos++ runtime system [3]. On the one hand, Mercurium is a source-to-source compilation infrastructure aimed at fast prototyping. It is mainly used to implement OpenMP and OmpSs, but since it is extensible, it has been used to implement other programming models and compiler transformations. Extending Mercurium is achieved by using a plugin architecture, where plugins are dynamically loaded by the compiler according to the given user configuration. Code transformations are implemented in terms of source code (that is, there is neither a need to modify nor a need to know the internal syntactic representation of the compiler). On the other hand, Nanos++ is an extensible runtime library designed to serve as a runtime support in parallel environments. Nanos++ supports OmpSs, among other programming models, and it is responsible for scheduling and executing parallel tasks specified by the compiler based on the constrains specified by the user.

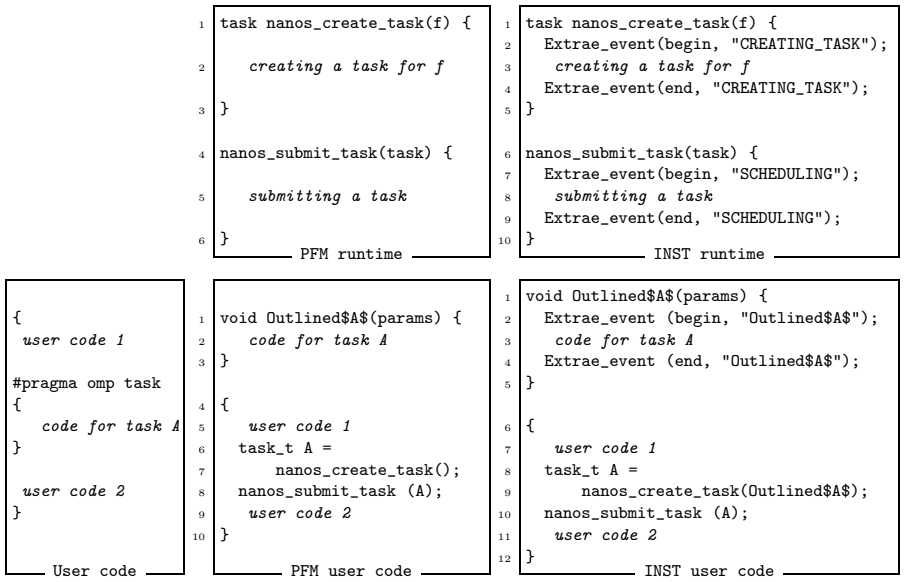
3 Implementation

We discuss in this section the modifications applied to the Nanos++ runtime library and to the Extrae instrumentation package. The modifications discussed below include how to generate instrumentation events, how to manage task switching from the Nanos++ runtime point of view and how to postpone such management to the Extrae post-process phase. We show also the modifications applied when visualizing the tasks without having to modify the display tool.

3.1 Implementation of the Instrumentation Plugin for Nanos++

The Extrae instrumentation library provides an API to emit events into the tracefile. Despite being accessible from Nanos++, Extrae needs a mechanism to identify which thread is executing the function called. We have added a call to the API of Extrae that allows identifying the calling thread by providing a callback

Table 1. Comparison between the performance (PFM) and instrumented (INST) versions of the runtime (top). Comparison between the user source code and compiler transformations for the performance and instrumented versions (bottom).

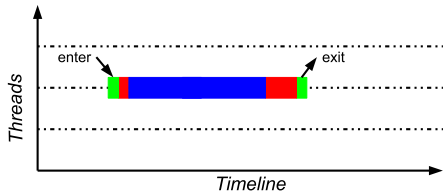


function. Nanos++ is then responsible for initializing the Extrae package, and also to set up the appropriate thread identifier routine by using this new API function so as the events are properly assigned to the executing thread. Once the OpenMP/OmpSs application is running, Nanos++ uses the Extrae API to emit events into the tracefile and also to notify to Extrae whether the application has finished.

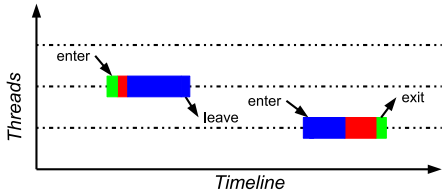
Code in the bottom part of Table 1 shows an example of use of the tasking constructs accompanied by the transformations done by the compiler when using the performance and instrumentation version of the generated code. In the instrumented version the compiler uses the Extrae API to generate two events to delimit the user function task. Also, the instrumented runtime adds events at `nanos_create_task` and `nanos_submit_task` services allowing a further analysis in the Nanos++ runtime, as shown in the top part of Table 1. For example, task creation will change thread/task state to an specific `CREATING_TASK` value, while submitting a task will change the state to `SCHEDULING`.

From the perspective of the runtime and regarding the events generated by the Nanos++, we classify them in four categories:

- *Punctual events*: a stateless event that occurs in a specific timestamp.
- *Burst events*: an event delimited by a starting and an ending timestamps (e.g. function execution).
- *State events*: it is an specific case of a burst event but due its importance it has been treated in a different manner (e.g. whether thread/task is idle).



(a) Task execution without context switches.



(b) Task execution with context switch.

Fig. 1. Execution pattern of OpenMP/OmpSs tasks

– *Point to point events*: it is a relation between two objects at different timestamps (e.g. task *a* sends data to task *b*, or task *b* is blocked because of task *a*).

Most of the generated events during instrumentation are burst or state events, which means that they are delimited by a starting point and an ending point. Task instrumentation of such kind of events requires data bookkeeping along with the cooperation of the runtime library. We refer the saved data as instrumentation context and it contains task instance instrumentation information that needs to be backed up and restored when a task is suspended or resumed, respectively. Figure 1 illustrates this problem.

Figure 1(a) shows a timeline displaying the routine that is being executed by the thread. As long as the application executes, the thread enters in three different nested routines (colored in green, red and blue, respectively), and then leaves them, returning to the previous one. If a task context switch occurs, as in Figure 1(b), it is necessary to save the state of the thread, so as it can be restored whenever it gets resumed (eventually, in a different thread).

Managing such amount of data structures during program execution would incur at additional overhead cost. So as to avoid adding overhead in the instrumentation, we have decided to move the stack management at the trace generation step. Thus the runtime library needs to register in the instrumentation library which events must be considered part of the instrumentation context. During trace generation, the process keeps track of the registered events. If the trace generation process encounters a suspend or resume event, it will backup or restore the instrumentation context in the tracefile.

The nature of the unstructured parallelism due to the usage of the tasking constructs, and more precisely in the case of the untied tasks and the existence of dependencies in OmpSs, makes the task migration and task dependency tracking

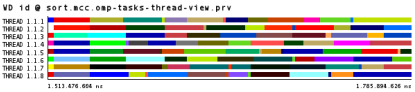
also important to be visualized. So as to fulfill this need, we have extended the API of Extrae by introducing two new event types so as to provide a mechanism that creates a relation between two threads (for example when a task is suspended in a thread and resumed in another thread). The new API calls emit information that is later translated by the tracefile construction. For the tracefile construction, we reuse the matching mechanism that Extrae has for MPI point-to-point operations in order to provide these new task relationships.

We have also implemented a mechanism that performs a deferred emission of events. This mechanism allows queuing events into a task which will be emitted when the task starts executing. Deferred events allow the runtime to place events into the tracefile without knowing when they will actually happen. One example of this usage involves task dependency tracking. During execution, a task only knows its successor tasks but does not know its predecessor tasks. Furthermore, when a task gets started it does not know whether it had predecessors or not. So the predecessor is the responsible for tracking the dependency between them without knowing which thread will execute the successor task. So as to track the dependency, the predecessor issues two events. While the first event is generated in its own instrumentation context, the second event is generated in the instrumentation context of the successor task. These two events are meant to be combined in the Paraver tracefile so as to generate a point-to-point event, and thus, correlate the two threads involved in the dependency.

3.2 Displaying OpenMP and OmpSs Tasking Traces

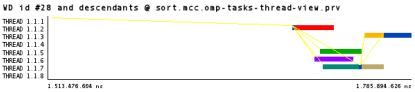
The Paraver resource mapping has fulfilled the needs of many types of executions, including multiple hybrid executions like MPI and OpenMP at the same time. However, adding support for tasks poses a difficulty to understand the logical execution of the threads because tasks may express irregular parallelism which is executed at any thread. Also, for the particular case of the untied tasks, a task can migrate from one thread to another, which turns that we would need an additional resource level to support representing a full-fledged execution. An example of this difficulty is shown in Figure 2(a), where the timeline shows information of the sort benchmark included in the Barcelona OpenMP Task Suite compilation [11]. In this Figure we see more than a hundred tasks (each colored differently) when the benchmark has been executed using eight threads. This Figure shows the reader the difficulty of following a set of tasks and establish their hierarchy. However, it is possible by using the visualization tool to track a particular set of tasks by selecting them by hand, as we depict in Figure 2(c). In the Figure we show in the execution of a particular thread including its migrations and its children tasks using the original display mode. From the picture we observe that the blue task was created at the first thread, it first executed at the second thread, then moves to the seventh thread and it finished its execution at the third thread. The blue task creates six tasks, four to execute the sort phase and two to execute the merge phase of the benchmark.

In order to mitigate this problem, we are experimenting with a new visualization mechanism to help on the task analysis. While the original timeline displays

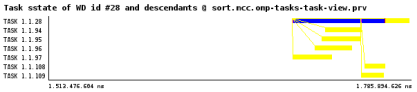


(a) Timeline showing the actual distribution of the OpenMP tasks within the threads.

(b) Timeline showing the state of every OpenMP task created during the execution.



(c) Timeline showing the actual distribution of an OpenMP task and its descendants within the threads.



(d) Timeline showing the state of a selected set of OpenMP tasks.

Fig. 2. Comparison of the two timeline display modes for the very same execution

information in terms of the application resources (i.e. every line in the Y-axis represents a combination of application, process and thread), the new visualization renders information in terms of tasks within a process (i.e. every line in the Y-axis represents a combination of application, process and task). For example, the whole execution of the sort benchmark by using the new display mechanism is shown in Figure 2(b). In this Figure we see the evolution of every task in the whole execution and their state. Considering that yellow means running and blue means blocked, we observe that there are two main running regions. First region is shown as a diagonal from the upper left to half-bottom right, which represents the sort phase in the benchmark. Second region is shown as a small diagonal (almost horizontal in the Figure) from the half-bottom right to bottom right, representing the merge phase in the benchmark. When both phases finalize, the main task (shown in the first row) finalizes. Figure 2(d) shows the state of the blue task referred previously in Figure 2(c) (identified as 1.1.28) and its descendants (1.1.94 to 1.1.97 for the sort phase and from 1.1.108 to 1.1.109 for the merge phase). As opposite Figure 2(c) we observe in the Figure 2(d) that task 1.1.28 spawns tasks 1.1.94 to 1.1.97 (colored in dark green, light green, purple and red, respectively, in Figure 2(c)), then waits for their completion, then spawns tasks 1.1.108 and 1.1.108 (colored in light brown and orange, respectively, in Figure 2(c)) and finally waits for their finalization.

We think that both display modes complement each other. While the original display mode is useful to determine the proper usage of the threads and task migrations, the new visualization mechanism can display more naturally the logical execution of tasks and their dependencies. A detailed application analysis would probably need using both of them so as to understand exactly which is the application behavior.

4 Experimental Results

We have done a twofold experiment of the environment we have presented. In the first experiment we evaluate the performance penalty of using the

instrumentation mechanism in Nanos++. To perform this evaluation, we use a set of benchmarks that use unstructured parallelism by using the OpenMP task clause and an application named Hydro [16]. In the second experiment we do a performance analysis of the Hydro application to show which kind of analyses can be done by using the integrated environment.

All the experiments described in this section have been executed on a system containing four hexa-core Intel® Xeon® E7450 processors running at 2.4GHz and 48 GBytes of RAM. The applications have been compiled using a development branch of the Mercurium version 1.3.5.8 and Nanos++ version 0.7a, which relies on the GNU C compiler version 4.3.4, using `-O3` as optimization flags. For the particular case of Hydro, we have compiled the application using OpenMPI version 1.4.4.

4.1 Overhead Analysis

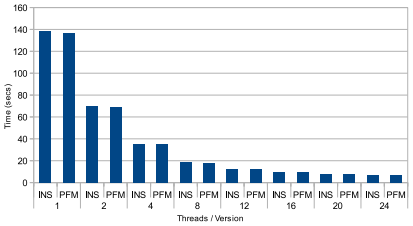
So as to analyze the instrumentation overhead of our integrated solution, we have chosen some benchmarks from the Barcelona OpenMP Task Suite (BOTS). This compilation of benchmarks is focused in testing such unstructured parallelism in combination with other OpenMP worksharing constructs or recursive techniques. Most of them include a cut-off mechanism that is used not to generate more tasks and continue executing in a serial fashion. The cut-off allows to handle task granularity (i.e. the amount of work executed by each task) allowing users to manage a trade-off between application parallelism and the runtime overhead.

Due to space restrictions we have selected a subset of the BOTS benchmarks. This subset includes:

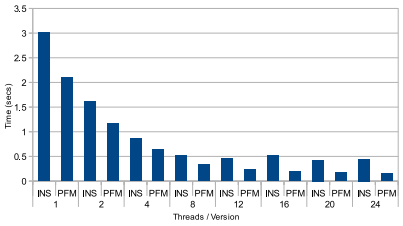
- *SparseLU* computes a LU matrix factorization over sparse matrices. A first level matrix is composed by pointers to smaller submatrices (blocks) that may not be allocated. In each of the SparseLU phases, a task is created for each block of the matrix that is allocated. We have used a matrix of $50 * 50$ blocks, and several block sizes from $25 * 25$ up to $200 * 200$. Due to space restrictions, we show the results for blocks of $50 * 50$ and $200 * 200$.
- *Floorplan* kernel computes the optimal floorplan distribution of a number of cells (each cell with its own shape description) which has been provided as a parameter. The algorithm calculates the minimum area size which includes all cells through a recursive branch and bound search. Floorplan application implements a cut-off based on the depth of the tree. In our experiments we have tested 20 cells input with three cut-off values (4, 5 and 6).

Table 2. TLB miss ratio of NQueens with different cut-off values

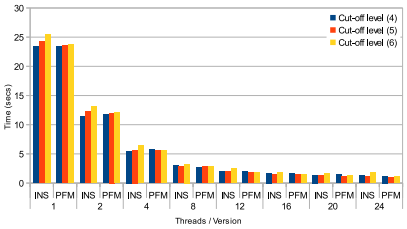
	.00−.01	.01−.02	.02−.03	.03−.04	.04−.05	.05−.06	.06−.07	.07−.08	.08−.09	.09−.10
Cut-off set to 2	99.99%	0.01%								
Cut-off set to 3	99.99%	0.01%								
Cut-off set to 4	99.99%	0.01%	0,00%							
Cut-off set to 5	44.02%	8.51%	8.55%	10.76%	11.52%	6.94%	4.30%	2.75%	1.69%	0.96%



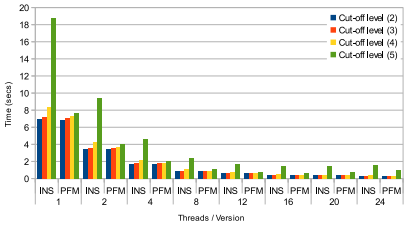
(a) SparseLU (M=50, N=200).



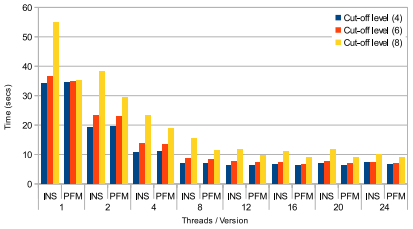
(b) SparseLU (M=50, N=50).



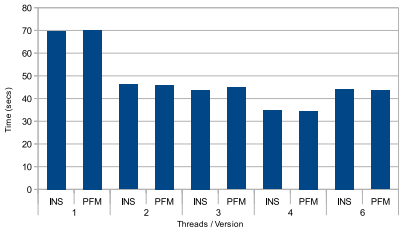
(c) Floorplan (20 cells).



(d) NQueens $13 * 13$ cheessboard.



(e) Strassen $4K * 4K$ matrix.



(f) Hydro $2K*2K$ matrix with block size of $256 * 256$ elements.

Fig. 3. Execution time for instrumented (INS) and performance (PFM) version of different applications

- *NQueens* computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an $n * n$ chessboard such that none of the queens attack any other. The benchmark uses a backtracking search algorithm with pruning, creating a task for each step of the solution. The algorithm has a cut-off mechanism based on the recursion level of the kernel. In our experiments we have tested a chessboard of $13 * 13$ and four cut-off values (2, 3, 4 and 5).
- *Strassen* algorithm uses hierarchical decomposition to multiply large dense matrices. Decomposition is done by dividing each dimension of the matrix into two sections of equal size and a task is created for each decomposition step. In our experiments we have used a matrix $4K*4K$ of complex numbers and three cut-off levels (3, 4 and 5).

Figure 3 shows the results obtained when executing the aforementioned BOTS benchmarks plus the Hydro application. The Figure shows the comparison of

the application times by using the standard performance and the instrumented versions. The X-axis shows the number of threads and also the runtime version, while the Y-axis shows the execution time. The plots also show the execution time with different cut-offs where applicable.

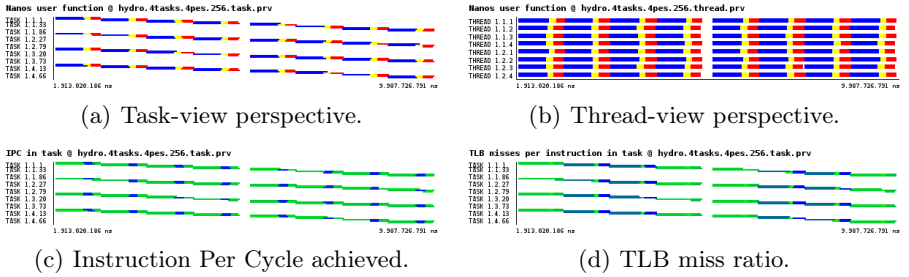
In all cases, the reader can see that the instrumented version adds some overhead with respect the performance one. Overhead depends on the task granularity, which is related also with cut-off values as in Floorplan, NQueens and Strassen, or block size like in SparseLU. While on large task granularity the overhead is less than 1% we find that reducing the task granularity increases the overhead. For example in Figure 3(b) we see that the overhead for SparseLU is about 43% with 1 thread, in Figure 3(e) we observe up to a 56% and Figure 3(d) shows the largest overhead by using 1 thread (243%).

So as to understand the reasons of the large overhead experienced in the NQueens benchmark, we have executed an instrumented version of the benchmark using one thread with the different cut-off values and adding hardware performance counters metrics into the tracefile. Table 2 shows on each row the TLB miss ratio analysis (i.e. number of TLB misses per instruction) for each cut-off value shown in Figure 3(d). Each column in the table represents the TLB miss ratios ranging from 0 to 0.1 in buckets of 0.01. The value in each cell is the percentage of time that the thread has been experiencing that TLB miss ratio. Using a cut-off level of 2 we observe that 99.99% of the total time the number of TLB misses has been between 0 and 0.01, whereas the remaining time a TLB miss ratio between 0.01 and 0.02. By comparing cut-off levels between 2 and 4 we do not observe a fluctuation on this behavior. This is no longer true when the cut-off value is set to 5 where 44.02% of the total time experiences a TLB miss ratio between 0 and 0.01. More than 25% of the execution time shows a TLB miss ratio between 0.03 and 0.06. This large proportion of TLB misses per instruction is clearly increasing the overhead of the instrumentation. Our preliminary analysis of the situation indicates that the issue may appear because the large number of tasks created consumes more memory pages because of the generation of the events.

4.2 Hydro Analysis

The Hydro application is a proxy benchmark of the RAMSES [4] application, that solves a large scale structure and galaxy formation problem using a rectangular 2D space domain split in blocks. Hydro is a mixed application that combines OmpSs tasking constructs with data dependencies and MPI. The application input requires to execute using 4 MPI processes and works with matrices of $2K * 2K$ elements split in blocks of $256 * 256$ elements.

First, we have analyzed the overhead of the instrumentation which is plotted in Figure 3(f). The diagram shows a minimal overhead, typically close to 1% and that the application does not scale linearly in respect to the number of threads used. In fact, the application does not even scale when using more than four threads.



(a) Task-view perspective. (b) Thread-view perspective.

(c) Instruction Per Cycle achieved. (d) TLB miss ratio.

Fig. 4. Different time-line perspectives of the Hydro execution

Due to space restrictions, we study the case that gives better performance from those shown in Figure 3(f). In Figure 4(a) we show a portion of the whole application execution in a timeline using the task view. Within the Figure three different types of OmpSs tasks appear colored in blue, yellow and red, which correspond to three outlined routines generated by the Mercurium compiler that are executed as OmpSs tasks. We observe that the yellow tasks do not start until the blue tasks finish and that the red tasks do not execute until the yellow tasks finish, this is because of the data dependencies in the source code. Although data dependencies limit the concurrency of the application because blue, red and yellow tasks do not overlap in time, we observe in Figure 4(b) that 4 instances of tasks are executed in parallel. This parallelism exists because the data dependencies in the loop are intra-iteration but not inter-iteration, giving the runtime the possibility to execute up to four iterations at a time.

Figures 4(c) and 4(d) show the achieved Instructions Per Cycle (IPC) and the TLB miss ratio, respectively. The data in these timelines are colored by a gradient that goes from green to blue, where green represents the lowest value and blue represents the highest value. We can correlate Figure 4(a) with Figure 4(c) and we can note that the tasks coded in yellow are the task achieving the highest IPC (0.57) whereas the rest of the code task runs at much slower IPC (0.13). By comparing Figures 4(a) and 4(d) we notice also that the yellow task has the lowest TLB miss ratio (less than 1 TLB miss per kilo-instruction). It is also noteworthy that the red and blue tasks present two different behaviors with respect to TLB misses. Half invocations of these tasks experience a high number of TLB misses (about 80 TLB misses per kilo-instruction in the red task, for instance) whereas the other do not miss so much in the TLB (less than 1 TLB miss per kilo-instruction again in the red task). These tasks execute the functions named `updateConservativeVars` and `gatherConservativeVars`, which depending on one of its parameters traverses a matrix by its leading dimension or not. Those invocations of the routine that do not traverse the matrix by its leading dimension present the highest miss rates in the TLB.

5 Related Work

POMP [20] is an extension to OpenMP proposed by Mohr *et al.*. POMP extensions include different mechanisms to obtain information from the OpenMP runtime, mostly using callback functions. POMP issues these callbacks whenever the OpenMP runtime reaches a selected specific code locations like: fork and join threads, invoking a parallel outlined function, entering and leaving a barrier, among others. To our knowledge, POMP has been fully implemented by Sun Microsystems[15] and partially implemented by IBM [10]. POMP is currently incomplete [10], *i.e.* not everything needed for a working implementation is specified. Compared to the work we present, it differs from POMP in the sense that the parallel programming runtime is responsible for emitting events through the instrumentation library.

Scalasca [24], Vampir [21] and TAU [23] are performance tools that collect data from OpenMP applications, among other types of applications. OpenMP data collection in these tools can be done through OPARI [19], DynInst [8]. OPARI implements POMP by performing a source to source translation of the user code, and thus limits the optimization opportunities of the application and also presents some other drawbacks as making implicit barriers explicit and not being able to instrument the worksharing constructs. DynInst is a binary rewriter library that allows adding monitors in specific locations of the code. The developer of the tool needs to know the OpenMP runtime calls, and this is not always possible. Also, each OpenMP manufacturer has its own runtime, which limits the chances of instrumenting all the OpenMP applications. The solution we propose relies on a tight cooperation between the parallel programming model and the performance analysis tool, this way the tool does not only know information about when events (like entering or leaving parallel constructs) occur but also information of the internal runtime state and also migration information of untied tasks. Also, the usage of Paraver allows a more flexible and precise analysis of the performance data.

Fürlinger *et al.* have written about a performance profiling for OpenMP 3.0 in [13]. They use a more recent version of OPARI, OPARI2 [18], to instrument OpenMP applications that use OpenMP tasking constructs. In their work, they provide a summary of the time spent on each task construct, the function executed as a such, in addition to imbalance, overhead and synchronization time. Because of the nature of the profile, it provides a summary of the execution and it cannot provide the rich details of the execution we can show by using a tracefile and a visualization tool as Paraver. Moreover, although OPARI2 provides support for OpenMP tasks, to our knowledge other performance tools have not extended their visualizations tools so as to display the execution of the OpenMP tasks.

Finally, ROSE [17] is an open source compiler infrastructure to build source-to-source program transformations and analysis tools. Authors proved their system by providing a transformation interface for the OpenMP programming model that tested lock misuse by replacing runtime library calls. Although the work is only developed for runtimes of the GNU and Omni compilers, it could

probably be extended to support other runtimes and to generate event trace based performance data. However, we believe that the emitting performance information of the runtime within it, can provide richer analysis rather than information gathered at runtime library calls.

6 Conclusions and Future Work

In this paper we have summarized the integration of a parallel runtime library that supports tasking constructs and an instrumentation library. Mercurium and Nanos++ work together to support OpenMP Tasking model and they also can produce all the information needed by instrumentation which will be used by Extrae to generate a Paraver trace. The close cooperation between them allows to generate highly detailed information about the program execution including information from the programming model perspective and also from the runtime internal information.

We have also presented a new trace display based on task, which complements the classic view by showing the application execution from the task perspective. This new view mode allows us to follow the liveness of a certain task through timeline without taking into account how it switches from one thread to another. Using task view mode we can easily follow task dependencies and we can also combine it with other existing events (e.g. hardware counters) to correlate them, as we have shown in the Hydro application study.

We have analyzed the overhead and we have studied the impact of our instrumentation in several application kernels and using different configurations. We are aware about the current limitations on terms of overhead of our implementation. As of today, we are gathering a huge amount of information from the runtime including: the number of ready tasks, the number of tasks blocked, the number of yields executed and the idle loop executions among others. We think that gathering all such information will easily excess the capabilities of the analysis tool for large runs and we would like to tackle this issue by rethinking which of the generated events are really required and which could be optional. Additionally, we are also working on a version of Nanos++ for distributed-memory systems [9] and accelerators [12]. We plan to extend the instrumentation mechanism in order to support these versions of the runtime.

Acknowledgments. We thankfully acknowledge the support of the European Commission through the Mont-Blanc project (FP7-288777), and the coordinated twin project HOPSA (FP7-ICT-2011-EU-Russia grant number FP7-277463, and Russian Ministry of Education and Science number 07.514.12.4001), and the support of the Spanish Ministry of Education (TIN2007-60625, CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980).

References

1. Extrae instrumentation package, `http://www.bsc.es/paraver` (accessed April 2012)
2. Mercurium C/C++ source-to-source compiler, `http://pm.bsc.es/projects/mcxx` (accessed May 2012)
3. Nanos++RTL. `http://pm.bsc.es/projects/nanox` (accessed May 2012)
4. RAMSES, `http://web.me.com/romain.teyssier/Site/RAMSES.html` (accessed May 2012)
5. Top 500 supercomputing sites, `http://www.top500.org` (accessed June 2012)
6. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009)
7. OpenMP Architecture Review Board. OpenMP Application Program Interface v 3.0 (May 2008)
8. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. Int. J. High Perform. Comput. Appl. 14(4), 317–329 (2000), `http://www.dyninst.org`
9. Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R.M., Ayguade, E., Labarta, J.: Productive Cluster Programming with OmpSs. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 555–566. Springer, Heidelberg (2011)
10. De Rose, L., et al.: An Implementation of the POMP Performance Monitoring Interface for OpenMP Based on Dynamic Probes, `http://www.research.ibm.com/actc/projects/pdf/T16p.pdf` (accessed May 2012)
11. Duran, A., et al.: Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: International Conference on Parallel Processing, ICPP 2009, pp. 124–131. IEEE (2009), `https://pm.bsc.es/projects/bots` (accessed May 2012)
12. Ferrer, R., Planas, J., Bellens, P., Duran, A., Gonzalez, M., Martorell, X., Badia, R.M., Ayguade, E., Labarta, J.: Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 215–229. Springer, Heidelberg (2011)
13. Fürlinger, K., Skinner, D.: Performance Profiling for OpenMP Tasks. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 132–139. Springer, Heidelberg (2009)
14. Hempel, R.: The MPI Standard for Message Passing. In: Gentzsch, W., Harms, U. (eds.) HPCN-Europe 1994. LNCS, vol. 797, pp. 247–252. Springer, Heidelberg (1994)
15. Itzkowitz, M., et al.: An OpenMP Runtime API for Profiling, `http://www.compunity.org/futures/omp-api.html` (accessed May 2012)
16. Lavallea, P.F., et al.: HYDRO, `http://www.prace-ri.eu` (accessed May 2012)
17. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 15–28. Springer, Heidelberg (2010)

18. Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., Wolf, F.: How to Reconcile Event-Based Performance Analysis with Tasking in OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 109–121. Springer, Heidelberg (2010)
19. Mohr, B., et al.: Design and Prototype of a Performance Tool Interface for OpenMP. The Journal of Supercomputing 23, 105–128 (2001), doi:10.1023/A:1015741304337
20. Mohr, B., et al.: A Performance Monitoring Interface for OpenMP. In: Proceedings of the Fourth Workshop on OpenMP, EWOMP 2002 (2002)
21. Nagel, W.E., et al.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer 12(1), 69–80 (1996)
22. Pillet, V., et al.: Paraver: A tool to visualize and analyze parallel code. Transputer and Occam Developments, 17–32 (April 1995), `http://www.bsc.es/paraver` (accessed April 2012)
23. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006)
24. Wolf, F., et al.: Usage of the SCALASCA for scalable performance analysis of large-scale parallel applications. In: Tools for High Performance Computing, pp. 157–167. Springer, Heidelberg (2008)