OpenMP Performance Analysis Approach in the INTONE Project

E. Ayguadé(1), M. Brorsson(2), H. Brunst(3), H.-C. Hoppe(4), S. Karlsson(2),
X. Martorell(1), W. E. Nagel(3), F. Schlimbach(4) , G. Utrera(1) and M. Winkler(3)

(1) European Center for Parallelism of Barcelona (CEPBA), Technical University of Catalunya (UPC), Spain.
(2) Department of Microelectronics and Information Technology, Royal Institute of Technology (KTH), Sweden.
(3) Center for High Performance Computing (ZHR), TU Dresden, Germany.
(4) Pallas GmbH, Germany.

Abstract

In this paper we present the general approach adopted in the INTONE project for performance analysis and optimization of OpenMP applications. The approach considers the following components: runtime interface (instrumentation and threading support) and its library implementation, compilation environments for Fortran90 and C/C++, and an extension of the VAMPIR graphical tool.

The paper also includes an outline of other components that will be developed in the INTONE IST project: support for software DSM systems, including extensions to OpenMP to inform the compiler about performance enablers and a performance assistant tool for OpenMP.

1 Introduction

OpenMP [1] is a highly portable programming model that supports most hardware architectures as well as operating systems including Unix and Windows NT in both programming languages of Fortran and C/C++. In addition, the current definition of thw OpenMP *Application Program Interface* (API) is able to cover a wide range of applications that need of long calculation time.

OpenMP is based on a shared-memory parallel programming paradigm and has been efficiently implemented on a variety of of shared-memory and NUMA vendor platforms. The simplicity of the model together and the possibility of incremental parallelization have widespread the use of this paradigm. Thus, there exists a need for tools that support performance analysis of such parallel programs. VAMPIR [2] has become a well-proven and highly successful tool that enables graphical analysis of parallel programs that use MPI [3] as a parallelization paradigm.

The main purpose of the INTONE project is to produce a new version of VAMPIR that is also useful for the analysis of parallel applications that use OpenMP or that combine MPI/OpenMP. To that end, the project has defined an instrumentation interface for the OpenMP user and for compiler writers and will produce an implementation of such interface to generate VAMPIR traces. OpenMP Fortran and C/C++ compilers will also be produced in the project. These compilers will be source-to-source and inject calls to the instrumentation interface and threading library interface, also designed and implemented in the project. The implementation of the compilers makes use of the experience acquired in the implementation of the NanosCompiler [4] for Fortran77, Odin [5] compiler for C and the NthLib threading library [6].

In this paper we present the two interfaces used by the compiler to generate instrumented parallel code. Although they are designed to be initially used by the INTONE compilers, they will be sufficiently generic to be used by other OpenMP compilation systems. Some extensions to the OpenMP API are proposed to enable the user to control the instrumentation process. The paper also outlines the VAMPIR visualization extensions for the visualization of OpenMP threads and for the analysis of the data available on the trace. Finally, the paper includes an outlook of the project objectives. For additional details about the project, please visit http://www.cepba.upc.es/intone.

2 Run-time Library Interface

In this section we briefly outline the interface that will be offered by the INTONE run-time library to the compilers developed in the project to support the generation of threaded code. The main functionalities provided by the interface are the following: spawning parallelism, support for work-sharing constructs, synchronization constructs as well as the OpenMP intrinsic functions.

INTONE compilers will encapsulate the body of parallel constructs in functions. The encapsulation process eases the management of address spaces (easy implementation of data scoping attributes in OpenMP). It also helps the instrumentation process, as the compiler generate instrumentation probes at thread level for every function entered and exited.

The primitive `intone_spawnparallel` spawns parallelism on the function provided as argument, which encapsulates the body of the parallel construct. This function receives as arguments the identification of the thread, the number of threads in the current team and the identifier of the master thread.

A set of services are provided to implement the loop scheduling algorithms available in OpenMP:

intone_begin_for, which starts the execution of a parallel loop in the current thread; intone_next_iters, which allows a thread to get the next chunk of iterations and indicates whenever the chunk obtained would be the last in the sequential execution; finally, intone_end_for is used to terminate the execution of the loop, entering a barrier if needed not overridden with the OpenMP NOWAIT clause.

These three primitives are used to support all work sharing constructs: DO, SECTIONS and SINGLE. The SECTIONS work-sharing construct is converted to a loop with as many iterations as number of SECTION. The SINGLE work-sharing construct is converted to a loop with a single iteration. In both cases, the schedule of the loop is set to dynamic with chunk equals to 1.

The interface also provides support to OpenMP synchronization mechanisms: implicit/explicit BARRIER synchronization (intone_barrier), execution inside a CRITICAL region (intone_set_lock and intone_unset_lock), MASTER execution (intone_is_master) and ORDERED execution (intone_enter_ordered and intone_leave_ordered). ATOMIC memory access (intone_atomic_update) and memory FLUSH (intone_global_flush and intone_selective_flush) are also provided by the interface. Other functions are available to directly support the OpenMP intrinsic functions.

3 Instrumentation

In this section we describe the instrumentation API (directives and runtime library), injection of probes by the compiler and runtime control of the instrumentation.

3.1 General Approach

The INTONE project is concerned with three distinct layers of instrumentation constructs or software:

- *Instrumentation directives* that are inserted by the user to indicate the type and amount of instrumentation data to be collected.
- *Instrumentation probes* that are inserted into the application code by the OpenMP compiler, controlled by the instrumentation directives and a general compile-time option.
- *Instrumentation API* that encapsulates the functions of the INTONE instrumentation library, and are used by the instrumentation probes.

Figure 1 shows the relationship between the instrumentation layers and the rest of the INTONE components. Besides defining the interfaces, a prototype implementation of the instrumentation library will be developed in the project, and the compilation systems will be enabled to carry out the insertion of instrumentation probes. The INTONE compilers are strict source-to-source compilers and so a separate compiler is used to produce object code.

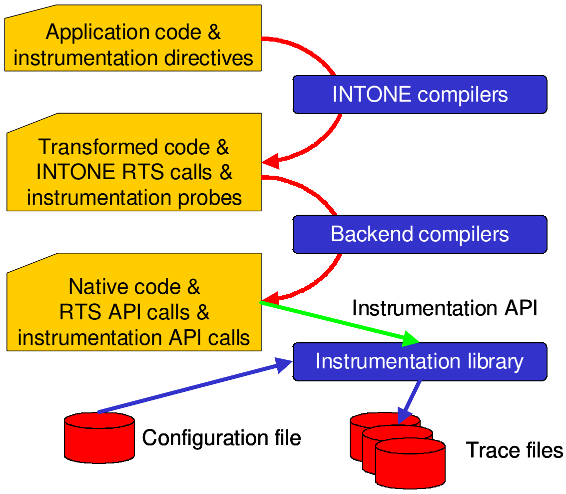

Figure 1: Instrumentation layers in INTONE.

3.2 Instrumentation Directives

The proposed instrumentation directives follow the format of OpenMP directives, being introduced with the prefix $OMPI in Fortran and #pragma ompi in C/C++. These directives will be ignored by non-compatible compilers and will not inflict in any way on the execution of object code produced by such compilers.

The directives have been defined for the instrumentation of function entries and exits, entries and exits to/from parallel regions and other OpenMP constructs. The trace data that can be recorded includes the timestamps for entry and exit, the function/region names and locations in the source code (SCL – Source Code Location), statistics about thread execution in a parallel OpenMP construct, and the values of hardware performance monitor registers, depending on the execution platform. For example,

```
#pragma ompi FUNCTION
#pragma ompi FUNCTION SCLS foo
```

tells the compiler to instrument all functions without SCLs except for function foo which is instrumented with SCLs.

For additional details about the proposed specification for the instrumentation interface, please visit the INTONE project page.

3.3 Instrumentation API

The INTONE instrumentation library inquires and records all instrumentation data, triggered by calls to its *instrumentation API*. These calls will be inserted by the INTONE compilers according to directives and compile time options. Some calls, or *probes*, is conditionally masked and can be enabled or disabled at runtime.

The management of data buffers, both in–memory and temporary on–disk buffers, and the interaction with the runtime control mechanisms is also handled in the instrumentation library.

The API allows the instrumentation of function entry, exit and change (`ompi_enter_function` and `ompi_leave_function` and `ompi_change_function`), parallel construct entry or exit (`ompi_enter_parallel` and `ompi_leave_parallel`), OpenMP constructs entry, exit and change (`ompi_enter_openmp`, `ompi_leave_openmp` and `ompi_change_openmp`) and record HPM counter values (`ompi_log_counter`). The interface also allows the definition of all elements involved in the instrumentation of the program (`ompi_define_function`, `ompi_define_tag`, `ompi_define_scl` and `ompi_define_counter`).

3.4 Runtime Control

Experience from MPI performance analysis tools shows that the amount of event–based trace data quickly grows to unacceptable levels. For OpenMP, the smaller granularity of constructs, and the more fine–grained partitioning mean even more trace data. To make performance analysis on medium–scale systems possible and usable, it is critical to provide the end–user with a powerful control facility that can be used to tailor trace collection for each application run, without the need for recompilation. This way, instrumentation probes can stay in the compiled code, and be switched on only when their information is requested.

A configuration file will be read in by the INTONE instrumentation library at application start–up. It contains filter settings for functions, OpenMP constructs and HPM registers, for MPI tasks, and for threads. Furthermore, the buffer management parameters can be set, and the level of recorded information can be customized, e.g. recording of all events or only record summaries.

For controlling the tracing while an application is running, a debugger can be attached to the application, and a new configuration can be transmitted to the library, that will then take effect when all "open" parallel constructs have been exited and the application is in a single–threaded phase.

3.5 Instrumentation Probes

As described in Section 2, the INTONE compilation system will transform the OpenMP directives into calls to the INTONE runtime system plus allocation and management of variables. In order to carry out the instrumentation process, it will insert probes to capture entry/exit to/from functions and OpenMP constructs (parallel, work-sharing and synchronization), and to record, at the same points, data from the hardware performance monitoring system available (HPM). These probes will then be compiled into machine code by the backend compilers, and be executed whenever the instrumented construct is executed. To allow for runtime control with minimum intrusion, some of the instrumentation probes will be *masked* by if–statements that inquire from a static variable whether a particular construct should be traced at all.

The compilers will support a number of *instrumentation levels*, generating instrumentation probes for strictly non–modified applications that contain no directives. For instance, automatic instrumentation of function entries and exits, and of parallel OpenMP constructs will be two of the supported instrumentation levels. Which level to use will be specified by a special command–line flag.

Next we present an example in C which shows how a short OpenMP code fragment is automatically instrumented (no instrumentation directives are used in this example) by the compiler. Number on the right refer to source code locations (SCL) which are later used to relate the trace visualized with the source code.

```
25      void foo(){
26
27      #pragma omp parallel sections
28         {
29      #pragma omp section
30            {
31               f1();
32            }
33      #pragma omp section
34            {
35               f2();
36            }
37      #pragma omp section
38            {
39               f3();
40             }
41         }
42      }     /* end foo */
```

For this example, the compiler would emit the following probes into the translated C code assuming we are instrumenting entry/exit of functions, entry/exit of OpenMP constructs and source code locations (SCLs). For performance reasons, each function, construct, SCL, etc., is associated once with a handle using the `ompi_define_tag`, `ompi_define_scl`, etc., functions provided by the instrumentation library. Subsequent functions will use the handles which will significantly reduce the execution time of the instrumentation probes. For brevity, we only present code related to the instrumentation of the first section, i.e., source code locations 29-32.

```
int __intone_handle_1;
int __intone_entry_scl_1;
int __intone_exit_scl_1;

ompi_define_tag("foo_sec_1",
                &__intone_handle_1);
ompi_define_scl("thisfile.c", 29,
                &__intone_entry_scl_1);
ompi_define_scl("thisfile.c", 32,
                &__intone_exit_scl_1);
```

The first section will cause the compiler to generate this piece of code. Code generated for other constructs and also functions will have the same structure. One probe will be inserted at the entry and another one at the exit. There might also be additional calls if, for example HPM data is to be included in the instrumentation trace.

```
ompi_enter_openmp(__intone_handle_1,
              & OMPI_SECTION,
              & __intone_entry_scl_1);
f1();
ompi_leave_openmp
              & (__intone_exit_scl_1);
```

4 Visualization Extensions

The performance visualization of OpenMP programs will be integrated in the VAMPIR environment. So far, Vampir was only able to support MPI programs running on a parallel system. The challenge within the INTONE project is to give additional insight into the parallel execution of a thread based OpenMP program within a SMP node.

4.1 Parallel Regions

To support the OpenMP programming model, we first have to show parallel regions. In future, we expect to have parallel MPI programs where each MPI task might have a parallel region on its own where the stream of control is split into several threads. Furthermore, we believe that the level of parallelism within MPI will be of the same order as today. This will lead to many more parallel execution threads. Showing each thread as the first level of information in many cases would lead to many hundreds or even thousands of process bars. For that reason, we have introduced a labeling mechanism. In the timeline display, parallel regions will be labelled in such a way that by only showing the master thread, the user can easily identify where other threads are alive. The labeling will be done showing a colored area within the bars (upper rectangle within each process bar in Figure 2).

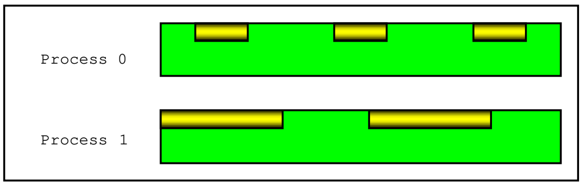

Figure 2. Labelled Parallel Regions

The user then will have the possibility of showing all or several threads, which should lead to a graphical view such as the one shown Figure 3. Interrupted areas in bars mean that threads are waiting for execution or are not alive. Since not every parallel region needs the same number of threads, the maximum number of required threads per CPU has always to be visible.

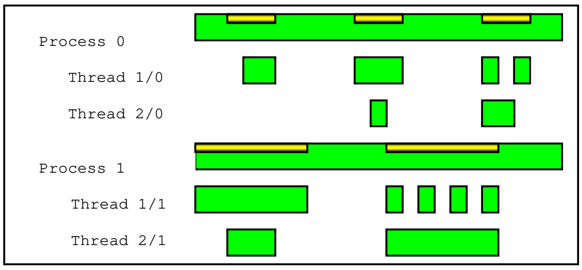

Figure 3. Thread Display in the Global Timeline Window

4.2 Hardware Performance Monitor (HPM) Data

Besides the information of parallel regions, there is a strong request to get information about the performance from the hardware performance monitor counters, which are available on all processor chips these days. Therefore, we will have an option to display data from the hardware performance monitor in a graphical way. Users will have the possibility to choose between the different kinds of HPM data, such as cache misses, MFLOPS, Instructions per second etc. while displaying all or a filtered number of threads. Besides the raw data rates, averaged values will be displayed, allowing to get an overview for larger code fragments. Minimum, maximum and average values can be sorted, thus users will easily get information about e.g. bad cache behavior in a certain part of the program and will know how many threads work properly at this function level. A display of minimum, average and maximum values in the same bar will also be given as shown in Figure 4.

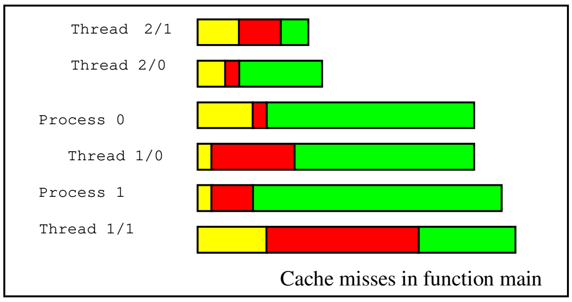

Figure 4. Sorted (by maximum value) HPM Chart

Furthermore, the already existing Summary Chart will be extended to show the sum of HPM data rate per state during the whole program run. This is important to get knowledge about which functions are not critical at all and which of them have to be investigated especially.

The functionality to jump directly from the global timeline window – from a special OpenMP block to the source code – is a very helpful feature that will be realized by implementing the source code location in the new trace records describing the blocks.

4.3 The Grouping Concept

The main topic when working with an application and trying to improve its performance is to analyze the data of previous executions, its summed values, to compare it and to conclude where the bottlenecks come from and how they can be prevented. The statistical data have to be collected and be shown in their own display. There should be a way to compare different threads or several CPUs regarding states, activities or OpenMP blocks.

There will be a new StateChart diagram, which will give an overview of the time used per thread (average, minimum, maximum execution time). In the StateChart, one might easily see which thread execution differs a lot from other threads executing the same state. This is a typical performance problem not only on DSM systems (such as SGI O2K). If many threads are involved, this view can even be improved by a sorted chart which will be supported by Vampir.

The current process model has to be changed for several purposes:

- Extending the current functionality of VAMPIR with threads will lead to much more trace data and information which has to be displayed.
- The limited space and resolution of a computer display allows the visualization of at most 10-20 processing entities at any one time when combined with additional information such as performance monitors.
- Using lots of processing entities may also confuse the user, who will not be able to compare too many shown data.

This leads to a new grouping concept as shown in Figure 5.

We will use a hierarchical approach, where the user can navigate through the trace data on different levels of abstraction, such as e.g. cluster nodes, processes, and threads. The display of trace data will start at a high degree of abstraction on the highest level and makes it comfortable going deeper into detail on every subsequent level, until the thread level finally is reached.

5 INTONE Project Outlook

In this section we overview other issues in INTONE closely related with the instrumentation environment: 1) the cluster programming system for Software Distributed Shared Memory and 2) the performance assistant tool.

5.1 Cluster Programming System

Most parallel computers with shared memory feature a relative small number of processors, generally fewer than 10. A cost-effective way of building larger parallel platforms is to cluster several SMP-nodes using a dedicated interconnection network for the communication between the different SMP-nodes. Such clusters can easily scale up to several hundreds of processors. However, this organisation does not readily support OpenMP

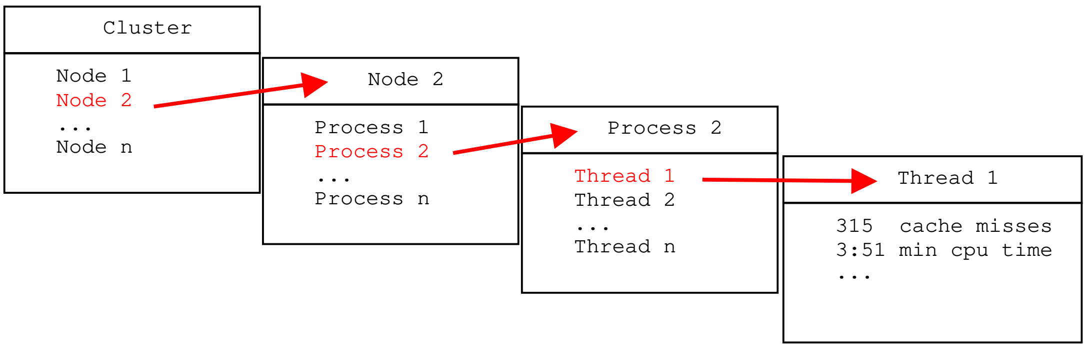

Figure 5. Grouping Model

applications, as the hardware does not permit memory to be shared beyond a node. Software Distributed Shared Memory (DSM) systems can be used to provide the image of shared memory across all the nodes. The two major methods on how to achieve memory coherency in the state-of-the-art in software DSM today – page-based systems which use the memory protection mechanism on page-level granularity [7] and instrumentation-based systems which use instrumentation of memory accesses to enforce coherence [1] – have relatively poor performance for many applications due to either false sharing because of the large coherence granularity of from high overhead in the instrumentation.

In the Intone project we intend to advance the state-of-the-art in this field by taking advantage of the fact that we have full control of the OpenMP translation process. In the OpenMP compiler developed in the project, we have almost full information on shared variables and when they are accessed which makes it possible to reduce the overheads associated with calls to coherency routines. For the cases where the compiler cannot determine which accesses are to shared memory – most notably through pointer accesses – we can fall back on a page-based software DSM system.

In addition to the hybrid fine-grained/page-based system described above we intend to investigate the performance and programming implications of using a hybrid shared memory and message-passing programming model. We have indications that a hybrid model which supports a shared memory across all processors of an SMP-cluster but with the possibility of message-passing for improved performance where needed has the possibility of providing the performance benefits of message-passing with the ease of use of shared memory [9].

5.2 Performance Assistant

Another important part of the project will be the support of non-experienced end-users in the performance analysis process. In order to analyze parallel program runs today, users still need a lot of expertise or guidance to find the major bottlenecks. We want to offer the possibility to assist such a non-expert user in this process by introducing a performance assistant as a new component of VAMPIR. The user should get information on how to interpret results, how to find possible performance bottlenecks, and how to solve identified problems.

Basis for the assistant will be a classification of known performance problems. The raw performance data of the parallel run will be analyzed according this classification. User should get a list, sorted by expected performance improvement rates, which will show bottlenecks and explain what to do to avoid it.

On the other hand, the assistant is supposed to give a step-by-step list about the analysis procedure. It should describe how to identify the program phases (usually initialization at the beginning, one or more parts of repeating patterns in the middle, and at the end evaluation, output data, preparation for the next run).

Furthermore this list should give the chance to jump from a part of the trace file of a certain thread to the source code where this part of the trace file is related to. At this level, user will get information e.g. on how to index outer and inner loops best, and the reasons for the observed runtime behaviour.

Finally, the step-by-step list will include information on how to find program parts using most of the time or where most, respectively, fewest cache misses and other HPM rates can be found. The innovative aspect here is the orientation towards non-expert users, plus the largely automatic operation that will save time even for experts.

6 Conclusions

In this paper we have presented the main components that integrate the OpenMP performance analysis and optimization system to be developed inside INTONE. We have presented the interfaces for the instrumentation and threading API, initially designed to be used inside INTONE, they will be sufficiently generic to be used by other OpenMP compilation systems.

Acknowledgements

This work is partially supported by the European IST project INTONE IST1999-20252. We acknowledge the end-users of the project (LMS and Enel.Hydro) for their comments and suggestions to the work presented in this paper.

References

1. OpenMP Fortran/C Application Program Interface, http://www.openmp.org.
2. Vampir: Visualization and Analysis of Parallel Applications. www.pallas.com/pages/vampir.html.
3. Message Passing Interface, www-unix.mcs.anl.gov.
4. M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. Concurrency: Practice and Experience. Special issue on OpenMP. vol. 12, no. 12. pp. 1205-1218. October 2000.
5. C. Brunschen and M. Brorsson, Odin/CCp – A Portable Implementation of OpenMP for C. Concurrency: Practice and Experience. Special issue on OpenMP. vol. 12, no. 12. pp. 1193-1203. October 2000.

6. X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta. "Thread Fork/join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors". 13th International Conference on Supercomputing (ICS'99), Rhodes (Greece). pp. 294-301. June 1999.
7. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel. *TreadMarks: Shared Memory Computing on Networks of Workstations*, IEEE Computer, Vol. 29, no. 2, pp. 18-28, February 1996.
8. D. J. Scales, K. Gharachorloo, and C. A. Thekkath, Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grained Shared Memory, In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96), October 1996, pp. 174-185.*
9. A. Rodman and M. Brorsson, Programming Effort vs. Performance with a Hybrid Programming Model for Distributed Memory Parallel Architectures, in *Proceedings of EuroPar'99*, September 1999.