Interfacing Computer Aided Parallelization and Performance Analysis

Gabriele Jost[1*], Haoqiang Jin[1], Jesus Labarta[2], and Judit Gimenez[2]

[1] NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000 USA
{gjost,hjin}@nas.nasa.gov

[2] European Center for Parallelism of Barcelona-Technical University of Catalonia (CEPBA-UPC), cr. Jordi Girona 1-3, Modul D6,08034 – Barcelona, Spain
{jesus,judit}@cepba.upc.es

Abstract. When porting sequential applications to parallel computer architectures, the program developer will typically go through several cycles of source code optimization and performance analysis. We have started a project to develop an environment where the user can jointly navigate through program structure and performance data information in order to make efficient optimization decisions. In a prototype implementation we have interfaced the CAPO computer aided parallelization tool with the Paraver performance analysis tool. We describe both tools and their interface and give an example for how the interface helps within the program development cycle of a benchmark code.

1 Introduction

During the last decades a large amount of time and money has been spent on the development of large-scale scientific applications exposing an insatiable appetite for floating point operations per second. High performance parallel computer architectures have evolved to satisfy this demand. Often the structure of the existing codes can not fully exploit the parallelism provided by the hardware. Considering the enormous investment in these codes and the niche market status of scientific applications, there is a strong incentive not to re-implement the applications from scratch, but rather, employ a conversion process to produce versions of existing codes optimized for the current most powerful computer architecture.

In this paper we focus on shared memory parallel computer architectures which provide multiple processing units and a globally shared address space. Most compilers for shared memory parallel architectures support parallelization in the form of compiler directives such as the OpenMP standard [12]. The program developer has to insert the directives and specify the scope of the variables. The CAPO [6] parallelization support tool was developed at the NASA Ames Research Center to aid the devel

[*] The author is an employee of Computer Sciences Corporation.

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 181–190, 2003.

oper in this task. CAPO automates OpenMP directive insertion into existing Fortran codes and allows user interaction for an efficient placement.

When parallelizing an application, the developer cycles through OpenMP directives placement, performance analysis, and optimization. To support this process we have interfaced CAPO with a performance analysis tool. Paraver [13] was developed at CEPBA-UPC to enable the user to obtain a qualitative global perception of the application behavior as well as detailed quantitative analysis of program performance. Paraver allows the user to graphically inspect trace files collected during program execution.

The CAPO-Paraver interface is a step towards the development of a programming environment for scientific applications which allows the user to jointly navigate through program structure and performance data to optimize critical code segments while repetitive but tedious and error prone tasks are automated. This paper presents the basic motivation and the general idea of how the tools should be integrated. We describe our initial prototype implementation and give a demonstration of its potential use. The rest of the paper is structured as follows: In Sections 2 and 3 we describe CAPO and Paraver. We discuss the interface between the tools in Section 4 and give an example parallelization and optimization session in Section 5. In Section 6 we discuss related work and draw our conclusion in Section 7 where we also elaborate on our future plans.

2 The CAPO Parallelization Support Tool

CAPO was developed to automate the insertion of OpenMP directives with nominal user interaction. This is achieved by use of extensive interprocedural analysis from CAPTools [4], developed at the University of Greenwich, which provides a fully interprocedural and value-based dependence analysis engine. Details on the CAPO parallelization process can be found in [6]. CAPO performs the following steps to exploit loop level parallelism:

1. Identify parallel loops and parallel regions based on dependence analysis.
2. Merge parallel regions and use the NOWAIT clause on successive parallel loops if possible.
3. Place OpenMP directives including variable scope declarations and perform necessary code transformations.

Dependence analysis results from a CAPO session are stored in a data base. This contains the dependence graph and a list of unresolved questions that occurred during the analysis. Unresolved questions indicate conservatively defined dependencies, often due to unknown information about values of variables. The user may browse the list of questions and provide assertions to make a more precise dependence analysis possible.

The CAPO directives browser is designed to display information gathered during the parallelization such as the reasons for loops to be parallel or serial, and the relevant variables. For each subroutine the user can retrieve information about the loops it contains. The browser also provides user interfaces to declare certain variables as shared or private or to explicitly remove dependences.

3 The Paraver Visualization and Analysis System

The Paraver system allows performance analysis of task level, thread level and hybrid parallel programs. It has two major components: A tracing package and a graphical user interface to examine the traces, including an analysis module to calculate various statistics.

Accepting a variety of trace formats Paraver optionally provides its own tracing package, OMPItrace [11]. It allows for tracing of programs with multiple processes and multiple threads. Depending on the computer system, OMPItrace dynamically instruments certain runtime libraries. Examples of OpenMP related information dynamically instrumented and traced on our development platform (SGI Origin 3000) are:

- entry and exit of OpenMP runtime library routines,
- entry and exit of compiler generated routines containing the body of parallel loops,
- two hardware counters,
- the state of a thread (running, idle, synchronization, or in fork/join overhead).

User routines are not automatically traced on the SGI Origin, but OMPItrace provides library routines for source code instrumentation.

The trace collected during the execution of a program contains a wealth of information, which as a whole is overwhelming. The information must be filtered to gain visibility of a critical subset of the data. This can be done through timeline graphical displays or by histograms and statistics. Paraver provides flexibility in composing displays of trace data. The Paraver object model is structured in a three level hierarchy: Application, Task, and Thread. Each trace record is tagged with these three identifiers. The timelines can be displayed at each level. At the task level, the values for each thread are combined to produce a value for the task. The Paraver GUI allows to specify the computation of a given performance index from the records in the trace and then save it as a configuration file. These configuration files are used to display a view of the selected performance index.

4 The CAPO-Paraver Interface

An important step in the parallelization process is to determine whether the directives have been placed for efficient parallelization. When working with large applications the developer is confronted with the question of which code sections to focus on. We address this question by the use of trace information and an interface to a performance analysis tool. Although there are many aspects to performance optimization we will focus at this point on the parallelization aspect. The design of the interface between computer aided parallelization and performance analysis is summarized in Figure 1.

4.1 Selective Source Code Instrumentation

We have extended the source code transformation capability of CAPO to automatically insert calls to the OMPItrace library. The OMPItrace module does not automati-

cally trace entry and exit to user routines on our development platform. Even though the automatic tracing of user level routines is possible on some platforms, it may not always be desirable to trace all of the subroutines because of large instrumentation overhead. In our prototype we use the following simple heuristics. At this point we selectively instrument:

- routines that are not contained within a parallel region or a parallel loop,
- routines which contain at least one DO loop,
- outermost serial loops.

Parallelized loops and parallel regions are automatically traced by the OMPItrace package as described in Section 3. During the instrumentation process CAPO generates a file containing symbol information which will be used by Paraver to relate the performance trace data with the routine names. The instrumented, parallelized code can now be compiled and run to obtain a trace file which can be processed by Paraver.

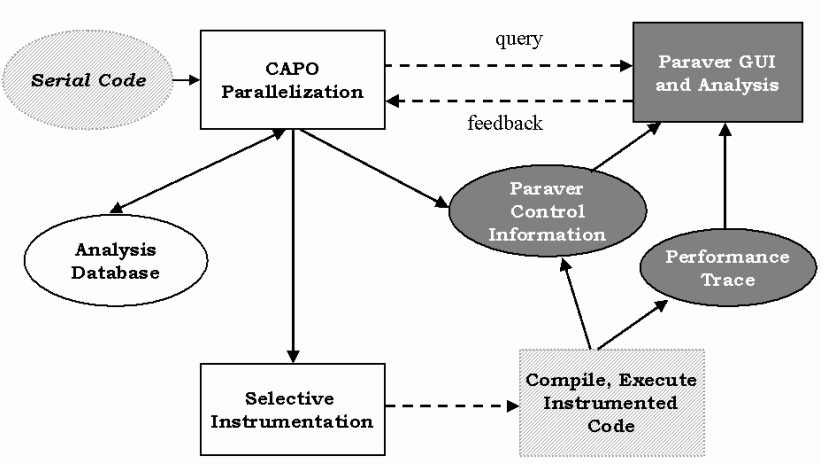


Fig. 1. The CAPO-Paraver interface. Solid lines indicate that data is automatically generated or read by the system components. Dashed lines indicate that user interaction is required. In our prototype querying the performance data is initiated by CAPO via Paraver configuration files. The outcome of the query is visually inspected by the user. Feedback to CAPO on what action to take is also provided by the user. The goal is to automate these steps

4.2 Display of Performance Metrics

We have designed a set of configuration files which show statistics related to the information displayed in the directives browser. The user will need to know:

- Where is the time spent?
- Is the time spent efficiently?

To address these questions the user may invoke Paraver from within CAPO. Paraver will load the previously generated trace file. The first view presents a diagram containing the percentage of time that each routine took. The time is presented on a task level, averaged over all threads. The exclusive time for a function call is reported

where exclusive time is the time for the routine minus the time for nested routines that are also instrumented. To give an indication of the quality of the parallelization we display the percentage of useful time for each thread. We define the useful time of a thread as time not spent in fork/join operations, synchronization or idle time. The Paraver analysis module calculates the percentage of useful time for each thread within the instrumented routines, as well as their mean and standard deviation. Each thread's average useful time should be high and the standard deviation low. CAPO automatically generates control information, in the form of configuration files and symbol information, which is then used by Paraver for calculating this performance metric and their display. Although the simple metrics described above are useful to identify routines with a bad parallelization ratio they do not conclusively point to the cause of the problem. A high standard deviation in useful thread time could for example also be due to an imbalanced workload among the loop iterations. The flexibility of the Paraver analysis module allows identification and addition of further metrics as our system evolves.

4.3 CAPO-Paraver Feedback Mechanism

CAPO supports querying the performance trace by generating control information and invoking Paraver. Our prototype requires the user to visually inspect the performance metrics calculated by Paraver and provide feedback to CAPO as indicated by the dashed lines in Figure 1. Our goal is to have CAPO query Paraver for performance metrics and retrieve the information without user interaction. We are working on a non-GUI based Paraver which will allow retrieving Paraver performance metrics in batch mode. With this, CAPO will be able to automatically query Paraver and correlate the outcome of the query with its loop analysis information. Our approach is to implement a rule based system where CAPO queries the performance trace, compares the metrics to given thresholds, relates the metrics to the program analysis information and then either performs certain transformations or requests further user input. An example for additional user input are assertions necessary to eliminate conservative dependences that prevent parallelization. Our prototype allows us to conduct experiments to gain experience on what the nature of the queries should be and how the outcome should affect the optimization process. The use of Paraver to provide profile data has the advantage that at any point the user can enter the process, inspect the metric requested by CAPO and carry out more analysis using the capability of the Paraver GUI and analysis module.

5 An Example Parallelization and Optimization Session

To demonstrate a parallelization session using the CAPO-Paraver interface we consider the BT benchmark from the NAS Parallel Benchmarks (NPB) [3]. We use the version as described in [5]. The BT benchmark solves three systems of equations resulting from an approximate factorization that decouples the x, y and z dimensions of the 3-dimensional Navier-Stokes equations. The development platform is an SGI

Origin 3000. To develop a parallel version of BT, the user starts a CAPO session loading the source code. Then CAPO performs the parallelization and generates the database. The user saves the CAPO generated source code containing OpenMP directives and the selective instrumentation as described in Section 4. Compiling and running the instrumented OpenMP code generates a performance trace file. In our demonstration the application runs on 4 threads. Paraver is invoked via a link provided by the CAPO user interface. Paraver loads the performance trace and a configuration file which calculates the percentage of time spent in each of the instrumented routines. The fragment of a snapshot is shown in Figure 2.

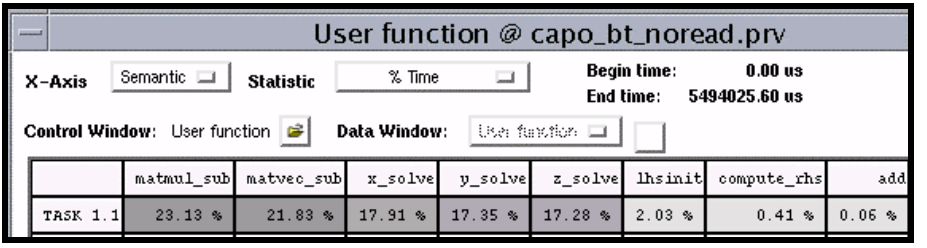


Fig. 2. Time distribution for the non-optimized parallel implementation of BT. Darker shading indicates a higher percentage of time

The useful time for each thread can be displayed by loading a second configuration file into Paraver, shown in Figure 3. The value of 0.96 for thread 1 in routine `z_solve` indicates that thread 1 was running user code 96% of the time. The value of 0.05 for thread 2 in routine `z_solve` indicates that the thread was running only 5% of the time. Routines `matmul_sub` and `matvec_sub` are obviously executed by only one of the 4 threads. The imbalance in useful time of the threads is an indicator for inefficient parallelization. All of these routines are major time consumers as seen in Figure 2.

Selecting routine `z_solve` in the directives browser and inspecting the parallelized loops reveals that the inner loops of the routine were chosen for parallelization (see Figure 4). Routines `matmul_sub` and `matvec_sub` are called from within routine `z_solve`, outside of the parallelized loop. For the outer loops, dependences had to be assumed. Inspecting the knowledge database (see Figure 4) shows that there are conservative dependences because the values of the entries in `gridpoints` are not known at compile time. These are the number of grid points for each dimension which are provided as input. They determine the loop lengths of the outer loops surrounding the parallelized loop. Since the user knows that the number of grid points is greater than 5 for each dimension, he can provide this information and repeat the analysis. The OpenMP directives are now placed on the outer loops of the solver routines. The new OpenMP code yields a much more balanced useful thread time for the time consuming routines as shown in Figure 5. The improved parallelization is reflected in a decrease of the overall runtime.

X-Axis: Semantic | Statistic: Average value | Begin time: 0.00 us | End time: 5494025.60 us

Control Window: User function | Data Window: useful

	compute_rhs	add	z_solve	y_solve	x_solve	lhsinit	matvec_sub	matmul_sub
THREAD 1.1.1	0.81	0.81	0.96	0.96	0.92	1	1	1
THREAD 1.1.2	0.82	0.84	0.05	0.05	0.05	0	0	0
THREAD 1.1.3	0.68	0.56	0.04	0.04	0.04	0	0	0
THREAD 1.1.4	0.66	0.57	0.04	0.04	0.04	0	0	0
Total	2.97	2.77	1.09	1.09	1.06	1	1	1
Average	0.74	0.69	0.27	0.27	0.26	0.25	0.25	0.25
Maximum	0.82	0.84	0.96	0.96	0.92	1	1	1
Minimum	0.66	0.56	0.04	0.04	0.04	0	0	0
Stdev	0.07	0.13	0.39	0.39	0.38	0.43	0.43	0.43

Fig. 3. View of the useful time per routine for each thread. Darker shading indicates a higher percentage of useful time

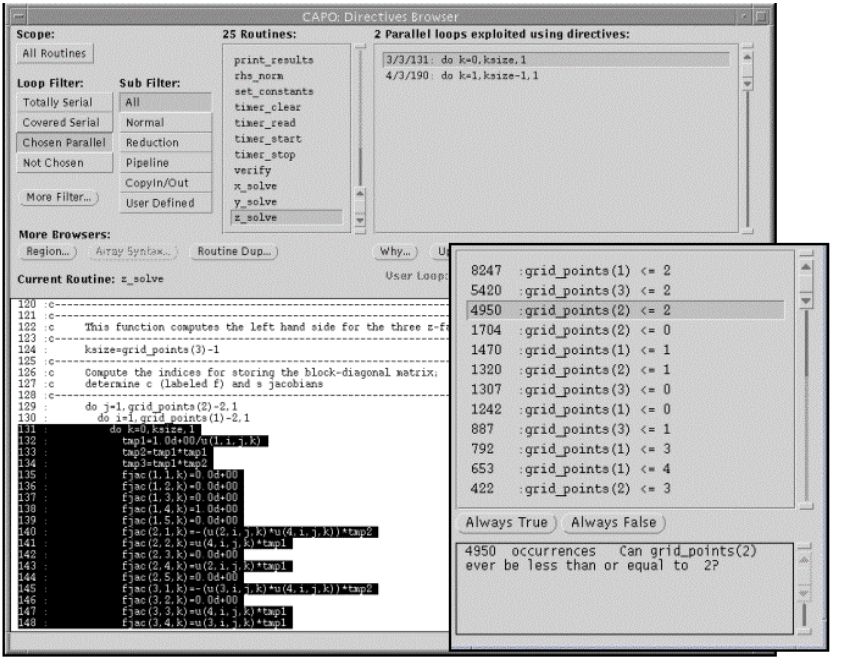
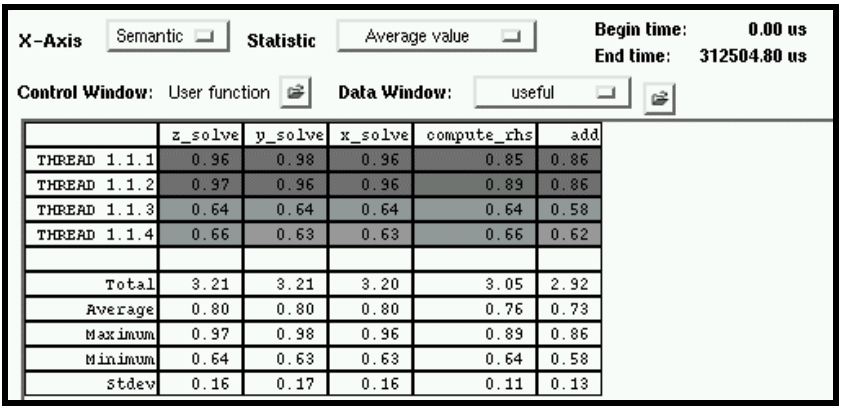


Fig. 4. The CAPO directives browser and knowledge database. The source code of the selected loop is highlighted



	z_solve	y_solve	x_solve	compute_rhs	add
THREAD 1.1.1	0.96	0.98	0.96	0.85	0.86
THREAD 1.1.2	0.97	0.96	0.96	0.89	0.86
THREAD 1.1.3	0.64	0.64	0.64	0.64	0.58
THREAD 1.1.4	0.66	0.63	0.63	0.66	0.62
Total	3.21	3.21	3.20	3.05	2.92
Average	0.80	0.80	0.80	0.76	0.73
Maximum	0.97	0.98	0.96	0.89	0.86
Minimum	0.64	0.63	0.63	0.64	0.58
stdev	0.16	0.17	0.16	0.11	0.13

Fig. 5. Useful thread time after optimization. Darker shading indicates a higher percentage of useful time. The routines `matmul_sub`, `matvec_sub`, and `lhsinit` that appear in Figure 3 are not instrumented, since they are now enclosed by parallel loops within the solvers `x_solve`, `y_solve`, and `z_solve`. The remaining difference in useful thread time is caused by workload imbalance due to the loop length

We have chosen a small example because it allows us to focus on explaining the components of our system, rather than a particular application. The motivation for building our system, however, is based on the need for dealing with large application packages. We have used our prototype successfully on large scientific codes and will report on the results elsewhere [7].

6 Related Work

There are a number of commercial and research parallelizing compilers and tools for automatic parallelization and performance analysis tools that have been developed over the years. We can only name a few.

The SUIF Explorer [8] developed at Stanford University is an interactive parallelization tool based on the SUIF [14] compiler. It performs extensive static dependence analysis and also includes a set of dynamic analyzers to provide runtime information. Runtime information includes checking dependencies and time profiles for loops and routines. The user can provide assertions via a graphical user interface. The most notable difference to our approach is that by interfacing to a full performance analysis system like Paraver, we have more performance metrics available than just the time. This allows us a more flexible and detailed analysis and greater opportunity to detect performance problems. A commercial product for interactive parallelization is the ForgeExplorer [1]. It provides means to display dependence analysis information but does not allow extensive user interaction the way CAPO does.

An example of a commercial product for performance analysis is Vampir [16] which allows tracing and analysis of MPI codes. The CAPTools system provides an interface for generating VAMPIR traces and invoking VAMPIR from within CAP-

Tools for performance analysis of message passing applications. Two research projects on performance analysis are Paradyn [9], which is developed at the University of Madison, and Aksum, which is part of the ASKALON [2] project conducted at the University of Vienna. Both aim at the automatic detection of performance bottlenecks. A general performance model for OpenMP is proposed in [10]. Performance libraries based on this model have been developed for the TAU (Tuning and Analysis Utility) performance analysis framework ([10], [15]) and the EXPERT automatic event trace analyzer [17]. The advantage Paraver offers is a high level of flexibility in computing performance indices and statistics. This allows exploration of metrics of interest and their corresponding influence on the parallelization choices.

7 Conclusions and Future Plans

We have interfaced the CAPO computer aided parallelization tool with the Paraver performance analysis system to support the scientific programmer when parallelizing existing sequential codes. We have used the static analysis information available from CAPO to instrument the code. By employing the statistical analysis module from Paraver, performance metrics for critical parts of the code can be calculated and displayed. We have shown how the CAPO user interface displays loop analysis information and the performance statistics available from Paraver to point the user to parallelization problems.

As mentioned, we are working on the design of a non-graphical interface to the Paraver analysis module to be used by CAPO directly, without user interaction. Many common problems could be detected by CAPO automatically and correlated to the loop analysis information and the knowledge data base. A high standard deviation, for example, in the useful thread time is often an indicator that an inner loop within a loop nest has been parallelized. In many cases CAPO might be able to determine that the reason for this is a conservative dependence in the outer loop. If the reason is missing information for a variable value, CAPO can prompt the user for an assertion. Another opportunity is the automatic detection of workload imbalance within a parallel loop which can be improved by changing the scheduling of the loop iterations from static to dynamic. We also plan to address scalability issues by automatically comparing traces for different numbers of threads.

Not all performance problems in full-scale applications will be solved automatically. At this point it is still not clear where to draw the line between automated and user guided responsibilities. Experiments with our prototype will allow us to identify more tasks that can be automated. Our approach is to successively develop an environment where the repetitive, cumbersome and error-prone tasks are left to the tools, allowing the user to focus on the tasks that require ingenuity. This will help reduce the development time and increase the quality of the generated code.

Acknowledgements. The authors wish to thank the CAPTools team (C. Ierotheou, S. Johnson, P. Leggett, and others) at the University of Greenwich for their support on

CAPTools. This work was supported by NASA contract DTTS59-99-D-00437/A61812D with Computer Sciences Corporation/ AMTI, by the Spanish Ministry of Science and Technology, by the European Union FEDER program under contract TIC2001-0995-C02-01, and by the European Center for Parallelism of Barcelona (CEPBA).

References

1. Applied Parallel Research Inc., "ForgeExplorer," http://www.apri.com/.
2. ASKALON, http://www.par.univie.ac.at/project/askalon/.
3. D. Baily, T. Harris, W. Saphir, R. Van det Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0", RNR-95-020, NASA Ames Research Center, 1995.
4. C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Leggett, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163–195. http://captools.gre.ac.uk/
5. H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance", NAS Technical Report NAS-99-011, 1999.
6. H. Jin, M. Frumkin and J. Yan. "Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes," *in Proceedings of Third International Symposium on High Performance Computing (ISHPC2000),* Tokyo, Japan, October 16–18, 2000.
7. G. Jost, H. Jin, "Computer Aided Optimization and Performance Analysis of Hybrid MPI/OpenMP Applications: A Case Study", NAS Technical Report, to appear.
8. Liao, S., Diwan, A., Bosch, R. P., Ghuloum, A., Lam, M., "*SUIF Explorer: An interactive and Interprocedural Parallelizer*", 7th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Atlanta, Georgia, (1999), 37–48.
9. B.P. Miller, M.D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K.L. Karavanic, K. Kunchithhapdam and T. Newhall, "*The Paradyn Parallel Performance Measurement Tools",* IEEE Computer 28, 11, pp.37–47 (1995).
10. B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "*Design and Prototype of a Performance Tool Interface for OpenMP*", Internal Report FZJ-ZAM-IB-2001-09, Research Centre Juelich, Germany, 2001.
11. OMPItrace User's Guide, https://www.cepba.upc.es/paraver/manual_i.htm
12. OpenMP Fortran/C Application Program Interface, http://www.openmp.org/.
13. Paraver, http://www.cepba.upc.es/paraver/
14. SUIF Compiler System, http://suif.stanford.edu/.
15. TAU: Tuning and Analysis Utilities, http://www.cs.uoregon.edu/research/paracomp/tau.
16. VAMPIR User's Guide, Pallas GmbH, http://www.pallas.de.
17. Wolf, F., Mohr, B., "Automatic Performance Analysis of SMP Cluster Applications", Tech. Rep. IB 2001-05, Research Centre Juelich, Germany, 2001.