Performance Instrumentation and Compiler Optimizations for MPI/OpenMP Applications *

Oscar Hernandez[1], Fengguang Song[2], Barbara Chapman[1], Jack Dongarra[2], Bernd Mohr[3], Shirley Moore[2], and Felix Wolf[3]

[1] University of Houston, Computer Science Department
Houston, Texas 77204, USA
{oscar, chapman}@cs.uh.edu
[2] University of Tennessee, Computer Science Department
Knoxville, Tennessee 37996, USA
{song, dongarra, shirley}@cs.utk.edu
[3] Forschungszentrum Jülich, ZAM
52425 Jülich, Germany
{b.mohr, f.wolf}@fz-juelich.de

Abstract. This article describes how the integration of the OpenUH OpenMP compiler with the KOJAK performance analysis tool can assist developers of OpenMP and hybrid codes in optimizing their applications with as little user intervention as possible. In particular, we (i) describe how the compiler's ability to automatically instrument user code down to the flow-graph level can improve the location of performance problems and (ii) outline how the performance feedback provided by KOJAK will direct the compiler's optimization decisions in the future. To demonstrate our methodology, we present experimental results showing how reasons for the performance slow down of the ASPCG benchmark could be identified.

1 Introduction

Many tools have been created to help find performance bottlenecks and tune parallel codes written using the hybrid MPI/OpenMP programming model. Yet tool use remains labor-intensive and fragmentary. Our goal is to improve the application development and tuning process by creating an integrated environment for parallel program optimization that reduces the manual labor and guesswork of existing approaches. We are developing strategies that enable the application developer, compiler and performance tools to collaborate and that generate code based upon a variety of sources of feedback. To demonstrate our ideas, in this paper we describe the integration of existing, open source software - the OpenUH compiler [10] and the automatic trace analysis tool KOJAK [19] – into a single, coherent environment, called COPPER, for collaborative application

* This material is based upon work supported by the National Science Foundation under grant No. 0444363 and 0444468.

tuning. The integrated application optimization environment permits a variety of user and tool interactions to solve performance problems, such as compiler assistance in the instrumentation of an application, and provision of high-level feedback information by performance tools to direct the compiler in further optimizations.

In Section 2 we briefly discuss other tools that support aspects of performance feedback analysis and optimizations. In Section 3 we give an overview of our system and describe its components. Section 4 describes how OpenUH and KOJAK interact and the APIs used to accomplish this interaction. Section 5 describes the feedback optimization facility in OpenUH and how it can be extended to support OpenMP optimizations. Section 6 describes the initial evaluation of our system and experimental results using the ASPCG kernel. In Section 7, we present our conclusions and plans for future work.

2 Related Work

Profile-guided optimization (PGO), where a program is compiled and run to collect execution profiling information, that is in turn used in a subsequent compilation step to perform optimization, has been exploited by modern static compilers to achieve significant speedups [4]. On the other hand, the exploitation of dynamic feedback has been explored for runtime program improvement [2] [6] [5], dynamic compilation [3] and the Java environment [1], where profiling and sampling information is collected to direct run time optimizations. Unfortunately most of these systems do not provide integrated support for MPI/OpenMP code optimizations and do not take advantage of automatic performance analysis that searches large amounts of performance data to locate inefficiencies on a higher level of abstraction.

Existing state-of-the-art performance tools addressing the combined use of MPI and OpenMP in a single application, such as TAU [11], VGV [7], and VAMPIR [14] deliver valuable performance feedback, albeit on a relatively low level of abstraction. Higher-level information obtained from an automatic analysis of trace data is provided by KappaPi [8], but only for pure message passing applications. Aksum [16], Paradyn [12], and Periscope [15] offer automatic performance analysis features for both MPI and OpenMP applications, but make certain assumptions about the deployment infrastructure that make them less suitable candidates for integration with the OpenUH compiler.

3 Overview

Figure 1 depicts the overall architecture of the envisioned COPPER environment and how it relates to the application optimization process. The process starts with the instrumentation of OpenMP constructs on the source-code level using a preprocessor called OPARI [13]. In the next step, the application is compiled by OpenUH and, at the same time, the compiler inserts instrumentation into the user code to generate traces for KOJAK. After application termination, KOJAK analyzes the resulting trace file and provides higher-level feedback that is returned to OpenUH's feedback optimization module. The initial version of the COPPER environment, as described in this article, integrates OpenUH and KOJAK by using compiler-based instrumentation. The KOJAK feedback is currently returned to the end user instead of the compiler. We hope to close the automatic feedback loop in the near future. Also, in a later step we plan to include the PerfSuite profiling tool [9] into the feedback loop for initial performance assessment and analysis of the OpenMP runtime system.

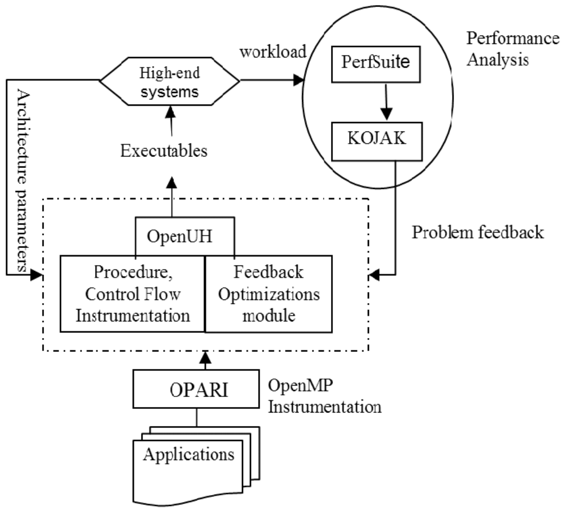


Fig. 1. COPPER architecture

The following sections describe each of the components in more detail.

3.1 OpenUH

OpenUH [10] is a compiler suite that supports C/C++ and Fortran 90/95 with OpenMP and/or MPI on the IA-64 running Linux. OpenUH is based on Open64, originally developed by SGI and subsequently maintained by Intel. It is supported commercially by Pathscale for Opteron architectures. OpenUH is available as an open source compiler. The major functional parts of the compiler are the front ends, the inter-language interprocedural analyzer (IPA), and the middle-end/back end, which is further subdivided into the loop nest optimizer, auto-parallelizer (with an OpenMP optimization module), global optimizer (or whole program optimizer), and code generator. OpenUH currently supports OpenMP 2.0. OpenMP is lowered during different compilation phases. The output code makes calls to a portable run-time library based on Pthreads. The description of how OpenMP is translated in OpenUH can be found in [10].

In the COPPER environment, application program regions are automatically instrumented by the OpenUH compiler. The OpenUH compiler supports the instrumentation of `function`, `loop`, `conditional branch`, and `compare and goto` program units.

3.2 KOJAK

KOJAK is an automatic performance analysis system for MPI, OpenMP, and hybrid applications written in C/C++ or Fortran. It is based on automatic pattern search in event traces and uses different interoperable components to support the analysis cycle from trace generation to visualization of analysis results.

OPARI [13] is a source-to-source translation tool that performs automatic instrumentation of OpenMP constructs according to the POMP profiling interface for OpenMP (see Section 4.1 for more details). The instrumentation of MPI functions is fully automated by interposing an MPI wrapper library based on the PMPI profiling interface.

At runtime, the instrumented executable generates a single trace file that can be searched off-line for inefficiency patterns using the EXPERT analyzer [19], which is part of KOJAK. The analyses concentrate on interpreting wait states resulting from suboptimal parallel interaction. Those can appear in MPI point-to-point or collective communication when processes have to wait for data sent by other processes or in OpenMP when threads reach a barrier at different points in time or when threads compete for the ownerships of locks. Low CPU and memory performance can also be analyzed by adding hardware counter information to event records.

The analysis process transforms the traces into a compact XML representation that maps higher-level performance problems onto the call tree and the hierarchy of system resources, such as nodes, processes, and threads. The XML file can be viewed in the CUBE performance browser (see Figure 4) that is also part of KOJAK or, alternatively, may be bautomatically processed by third-party tools using the CUBE API.

4 Tool Interactions

This section describes how KOJAK and the OpenUH compiler instrument OpenMP constructs and user regions, respectively. The profiling interface and advantages of using KOJAK high-level feedback are also presented.

4.1 OpenMP Instrumentation

Similar to the MPI profiling interface PMPI, Mohr et al. [13] have defined a portable API that exposes OpenMP parallel execution to performance tools. The performance interface is called "POMP". The POMP API consists of callback functions that are called before and after all OpenMP constructs and runtime routines. The callbacks can be inserted into the program during OpenMP compilation, through a source or binary instrumentation tool, or activated by an instrumented OpenMP runtime system.

OPARI [13] can add POMP instrumentation to C, C++, and Fortran programs. When reading a parallel program containing OpenMP directives, KOJAK automatically invokes OPARI to insert POMP performance calls where appropriate. As a final step of the instrumentation, KOJAK links the application with a library implementing the POMP API to generate appropriate events and write them to the trace buffer. Thus, with the help of the PMPI library and the OpenUH user code-region instrumentation, our approach provides a fully automatic solution to the instrumentation of OpenMP and mixed-mode MPI/OpenMP applications.

4.2 OpenUH Instrumentation and Profiling Interface

Compile-time instrumentation has several advantages over source-level and object-level instrumentation. We can use compiler analysis to detect regions of interest where we can measure and instrument certain events to support different performance metrics. Also, the instrumentation can be performed at different compilation phases, allowing certain optimizations to take place before the instrumentation. These capabilities play a significant role in the reduction of instrumentation points, improve our ability to deal with program optimizations, and reduce the instrumentation overhead and size of performance trace files.

The instrumentation module can be invoked at six different phases during compilation, which are before and after three major stages in the translation: interprocedural analysis, loop nest optimizations, and SSA/DataFlow optimizations. For example, if the user decides to instrument the source code after the interprocedural analysis phase, program transformations such as procedure inlining will reduce the instrumentation points for call sites and the compiler will instrument the body of the procedure being inlined.

The OpenUH compiler provides an interface to enable the instrumentation of three types of program regions: functions, conditional branches, and loops. The instrumentation of other types of regions, such as MPI operations and OpenMP constructs, are avoided so that they can be handled by the profiling libraries of PMPI and POMP. Procedure and control flow instrumentation is essential to relate the PMPI and POMP results to the execution path of the application. We plan to integrate the POMP instrumentation in later versions of OpenUH. For now we are using OPARI to do it. Additionally, the user has the option to instrument the code after OpenMP has been translated to threading code (by setting a special compiler flag). In this case OpenMP runtime system-specific calls are additionally instrumented by the compiler.

Each program region type is further divided into several sub-categories whenever possible. For instance, a loop type may be a `do loop`, `while do loop` or `do while loop`. Conditional branches may be of type `if then`, `if then else`, `true branch`, `false branch`, or `cselect`. The name of the sub-category is communicated back to KOJAK through the profiling interface and later displayed in the call-tree view of the KOJAK GUI. This detailed presentation provides users with a fine-grained control flow graph in which branches taken, branches not taken, and specific loops are all displayed.

The compiler instrumentation is performed by first traversing a program's intermediate representation to locate different program constructs. The compiler locates starting and exit points of constructs such as procedures, branches and loops to insert specific profiling calls at these points. The compiler profiling interface API is defined as follows. Argument `begn_ln` represents the beginning line of the region, and `end_ln` represents the end line of the region. `type` indicates the specific subtype of the region. `pu_name` is the name of a function or subroutine.

- Functions to initialize and finalize the profiling library:

  ```
  void profile_init(void)
  void profile_finish(void)
  ```

- Subroutine entry and exit functions:

  ```
  void profile_invoke(char *pu_name, INT32 begn_ln,
                      INT32 end_ln, char *file_name)

  void profile_invoke_exit(void)
  ```

- Conditional branch entry and exit functions:

  ```
  void profile_branch(BranchSubType type, INT32 begn_ln,
                      INT32 end_ln, char *file_name)

  void profile_branch_exit(void)
  ```

- Loop entry and exit functions:

  ```
  void profile_loop(LoopSubType type, INT32 begn_ln,
                    INT32 end_ln, char *file_name)

  void profile_loop_exit(void)
  ```

The interface has been implemented in KOJAK as part of the trace library to generate appropriate events upon region entry and exit.

4.3 KOJAK High-Level Feedback

To optimize a parallel application based on the MPI/OpenMP programming model, developers usually want to see whether the application is load imbalanced, where the synchronization overhead lies, and where there are opportunities to overlap computation and communication. For each parallel region, KOJAK is able to display the execution time broken down by call path and thread or process. The identification of wait states in combination with the distinction of different OpenMP constructs and program-region types simplifies the comparison of loop scheduling strategies and the validation of the program design. OpenMP *parallel for* and *parallel sections* constructs are two common cases where KOJAK can be used.

When the hardware counter feature is enabled, performance data supplied by the PAPI library is also recorded in the trace file. The hardware counter information will help us to correct CPU and memory anomalies such as unevenly distributed L1/L2 cache miss rate, significantly high TLB misses, and long stalls of the pipeline across a group of threads.

In a later stage of the integration project, the OpenUH compiler will read the analysis results from the XML file and automatically perform optimizations. The feedback mechanism to be used for this purpose is described below.

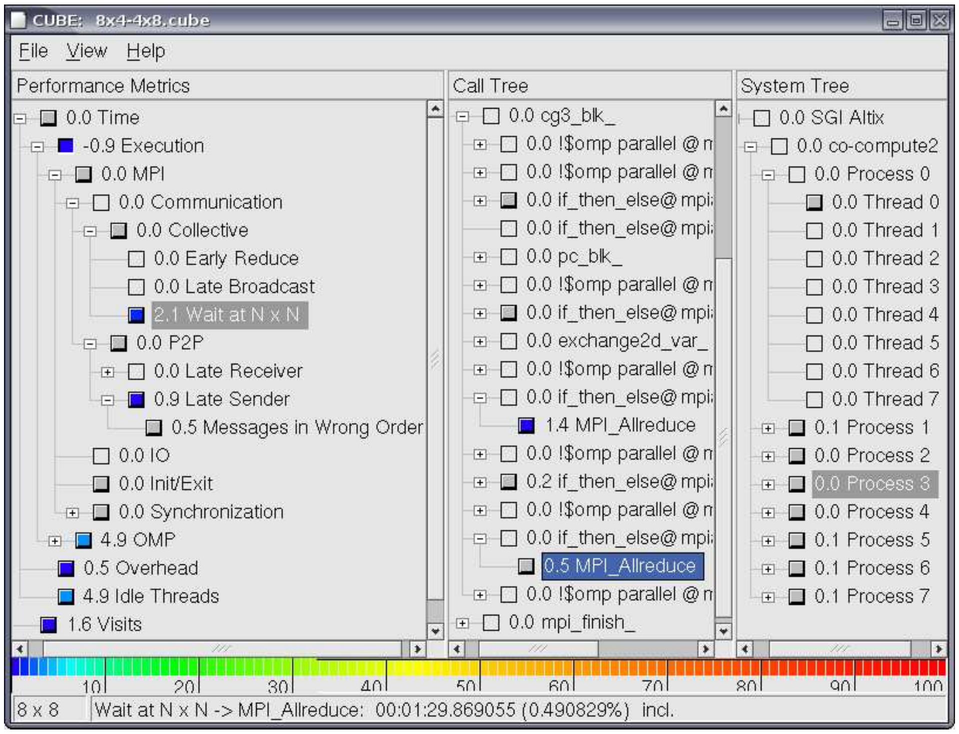
5 Compiler Optimizations for OpenMP

OpenUH has a basic facility for performing feedback-directed optimizations, which are accomplished via the automatic insertion of instrumentation and compiler annotation of the profiling results in the intermediate representation. It is able to use this facility to improve procedure inlining and branch prediction and to improve the cost modeling analysis for loop nests. The current feedback infrastructure supports profiling results at the procedure and control flow level of the procedure. Our main focus has been to extend this infrastructure to support hybrid OpenMP/MPI code optimizations. Our initial focus explores the use of loop transformations to alleviate the problem of parallel loop load imbalances. Feedback information from KOJAK gives the compiler cost modeling information related to the parallel overhead of a given transformation. For example, if KOJAK reports a synchronization problem (e.g. long waits at a barrier) because of load imbalances, the compiler will assign a high cost for parallel synchronization overhead. Based on dependence analysis and interprocedural array region (array data-flow analysis) information, the compiler will try out a set of loop permutations, including loop interchange, loop tiling and outer loop unrolling, to alleviate the problem. The compiler also applies different strategies, including attempting to determine which loop in the nest is most profitable to parallelize and determining an appropriate scheduling strategy from the options provided by the OpenMP standard.

6 Experimental Results

To illustrate our approach, we performed an experimental analysis of the different process/thread allocation strategies for the ASPCG sparse linear algebra kernel [18]. Large sparse linear systems that are used in simulation of turbulent flow calculation in complex geometries typically have several million unknowns. The ASPCG kernel from Virginia Tech solves such systems with a preconditioned conjugate gradient(PCG) method. The iterative CG method uses a two-level additive Schwarz preconditioner which adopts a domain decomposition approach to compute solutions on subdomains. The code is available in serial, MPI, OpenMP, and hybrid versions. For our experiments, we used the hybrid version and compiled ASPCG with the OpenUH compiler.

The experiments were performed on NCSA's SGI Altix machine which consists of two SMP systems running the Linux operating system. Each system has 512 Intel Itanium2 1.6 GHz processors. Figures 2, 3, and 4 show performance analysis results for the PCG method preconditioned by the additive Schwarz method for the problem size of 8194 × 8194. The GUI windows displayed include three trees. The left tree represents performance properties, the middle tree represents call paths of the source code, and the right tree represents system resources. The numbers represent percentages of the overall execution time.

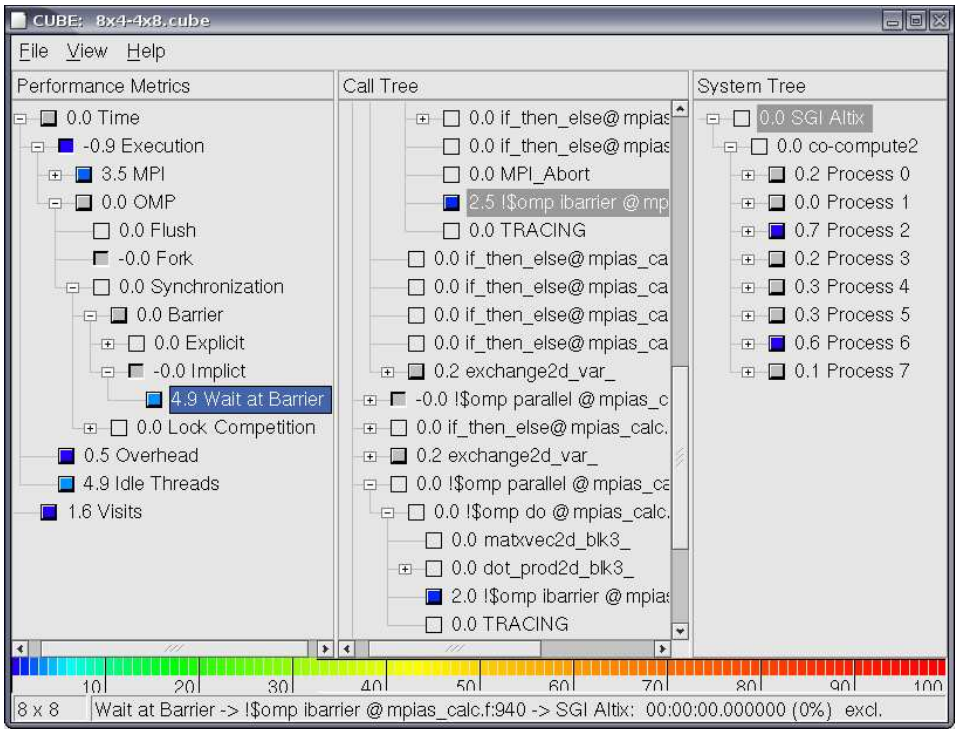


```
Performance Metrics                 Call Tree                          System Tree
0.0 Time                            0.0 cg3_blk_                       0.0 SGI Altix
  -0.9 Execution                      0.0 !$omp parallel @ n             0.0 co-compute2
    0.0 MPI                           0.0 !$omp parallel @ n               0.0 Process 0
      0.0 Communication               0.0 if_then_else@ mpi                  0.0 Thread 0
        0.0 Collective                0.0 if_then_else@ mpi                  0.0 Thread 1
          0.0 Early Reduce            0.0 pc_blk_                            0.0 Thread 2
          0.0 Late Broadcast          0.0 !$omp parallel @ n                 0.0 Thread 3
          2.1 Wait at N x N           0.0 if_then_else@ mpi                  0.0 Thread 4
        0.0 P2P                       0.0 exchange2d_var_                    0.0 Thread 5
          0.0 Late Receiver           0.0 !$omp parallel @ n                 0.0 Thread 6
          0.9 Late Sender             0.0 if_then_else@ mpi                  0.0 Thread 7
            0.5 Messages in Wrong Order 1.4 MPI_Allreduce                  0.1 Process 1
      0.0 IO                          0.0 !$omp parallel @ n               0.0 Process 2
      0.0 Init/Exit                   0.2 if_then_else@ mpi                0.0 Process 3
      0.0 Synchronization             0.0 !$omp parallel @ n               0.0 Process 4
    4.9 OMP                           0.0 if_then_else@ mpi                0.1 Process 5
  0.5 Overhead                          0.5 MPI_Allreduce                  0.1 Process 6
  4.9 Idle Threads                    0.0 !$omp parallel @ n               0.1 Process 7
1.6 Visits                          0.0 mpi_finish_

8 x 8   Wait at N x N -> MPI_Allreduce:  00:01:29.869055 (0.490829%)  incl.
```

Fig. 2. Difference experiment of ASPCG between 8×4 run and 4×8 run. Unlike Figure 3, this figure focuses on the difference in the MPI communication.

We conducted two experiments on 32 processors, 8 processes with 4 threads each and 4 processes with 8 threads each, respectively (i.e., 8 × 4 and 4 × 8



```
Performance Metrics            Call Tree                               System Tree
0.0 Time                       0.0 if_then_else@mpias                  0.0 SGI Altix
  -0.9 Execution               0.0 if_then_else@mpias                    0.0 co-compute2
    3.5 MPI                    0.0 MPI_Abort                               0.2 Process 0
    0.0 OMP                    2.5 !$omp ibarrier @mp                      0.0 Process 1
      0.0 Flush                0.0 TRACING                                 0.7 Process 2
      -0.0 Fork              0.0 if_then_else@mpias_ca                     0.2 Process 3
      0.0 Synchronization    0.0 if_then_else@mpias_ca                     0.3 Process 4
        0.0 Barrier          0.0 if_then_else@mpias_ca                     0.3 Process 5
          0.0 Explicit       0.0 if_then_else@mpias_ca                     0.6 Process 6
          -0.0 Implict       0.2 exchange2d_var_                           0.1 Process 7
            4.9 Wait at Barrier  -0.0 !$omp parallel @mpias_c
        0.0 Lock Competition     0.0 if_then_else@mpias_calc.
    0.5 Overhead                 0.2 exchange2d_var_
    4.9 Idle Threads             0.0 !$omp parallel @mpias_ca
  1.6 Visits                       0.0 !$omp do @mpias_calc.
                                     0.0 matxvec2d_blk3_
                                     0.0 dot_prod2d_blk3_
                                     2.0 !$omp ibarrier @mpias
                                     0.0 TRACING

8 x 8   Wait at Barrier -> !$omp ibarrier @mpias_calc.f:940 -> SGI Altix:  00:00:00.000000 (0%)  excl.
```

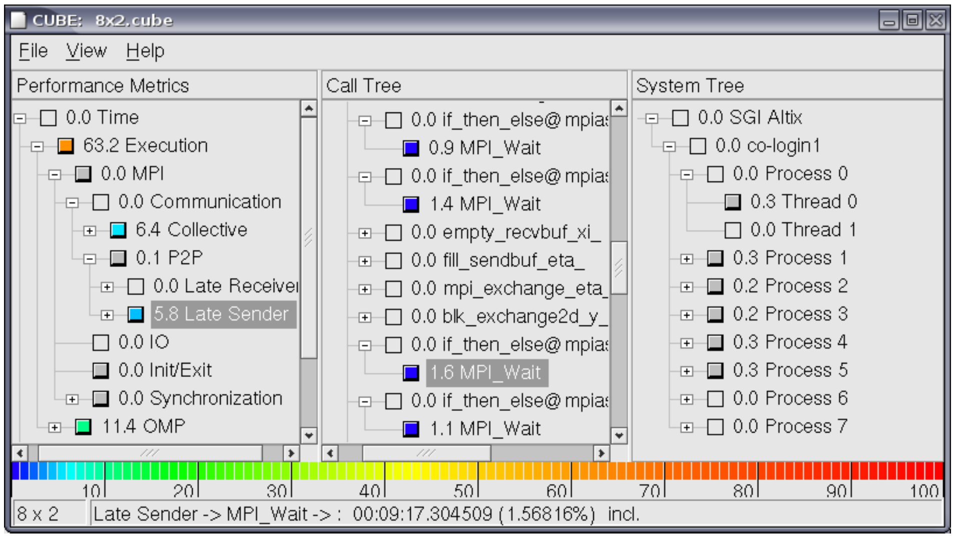
Fig. 3. Difference experiment of ASPCG between 8×4 run and 4×8 run. Unlike Figure 2, this figure focuses on the difference in the OpenMP constructs.

configurations). The walltime is about 35 minutes to execute the 8×4 experiment and about 25 minutes for the 4×8 run. A virtual difference experiment was computed by subtracting the 4×8 experiment from the 8×4 experiment using our performance algebra tool [17]. Positive values represent performance losses of the 8×4 experiment, and negative values represent gains of 8×4. By comparing the two experiments, we find the 8×4 run is slower than the 4×8 run by 12.8% (where 4.9% lies in OpenMP constructs and 3.5% in MPI communication). In order to show the difference experiment in more detail, we use two figures, Figure 2 and Figure 3, to display the difference located in MPI communication and OpenMP constructs, respectively. As shown in Figure 3, the 8×4 experiment spent 3.5% more time in MPI communication than the 4×8 experiment did. After expanding the tree node of "MPI", we can see detailed information in Figure 2. The reason why the 8×4 experiment is slower is because the greater number of processes induced bigger overhead in "Wait at $N\times N$" and longer blocking time in "Late Sender" and "Messages in wrong order". In addition, the increased loss in the metric "Idle Threads" (4.9%) is a consequence of the higher MPI time when the MPI calls were made during sequential phases of execution.

Prolonged sequential executions cause slave threads to remain idle, which is a typical problem of hybrid programs.

As we further investigate the reason for the greater cost in OpenMP (shown in Figure 3), we note that the selected property in the left tree reveals a performance problem: 4.9% of the total execution time was spent waiting in front of OpenMP barriers. The waiting time is significant and it is a common indicator of load imbalance. The scheduling strategy we used was `static`. The 4×8 experiment has better load balance than the 8×4 one, since the workload of the former is more evenly distributed among 8 threads than that of the latter which uses only 4 threads to compute the same amount of work.

Figure 4 displays the results for the 16×2 ASPCG experiment and demonstrates that more fine-grained control flow information can help users distinguish different instances of function calls within a region. As an example, the "Late Sender" property shown in Figure 4 takes 5.8% of the total execution time. In examining the "Late Sender" problem, we hope to identify which `MPI_Wait` in P2P communication spent most of the time blocking. With the help of conditional branch regions, we are able to identify the locations of the most expensive calls. The four expanded nodes with blue boxes in the middle tree account for 86% of the "Late Sender" inefficiency. In the CUBE performance browser, a user can click the right-button of her mouse on an interesting node to look at the corresponding source code and modify it to eliminate the problem.



```
Performance Metrics              Call Tree                          System Tree
0.0 Time                         0.0 if_then_else@mpias             0.0 SGI Altix
  63.2 Execution                   0.9 MPI_Wait                       0.0 co-login1
    0.0 MPI                      0.0 if_then_else@mpias                 0.0 Process 0
      0.0 Communication            1.4 MPI_Wait                           0.3 Thread 0
        6.4 Collective           0.0 empty_recvbuf_xi_                    0.0 Thread 1
        0.1 P2P                  0.0 fill_sendbuf_eta_                  0.3 Process 1
          0.0 Late Receiver      0.0 mpi_exchange_eta_                  0.2 Process 2
          5.8 Late Sender        0.0 blk_exchange2d_y_                  0.2 Process 3
      0.0 IO                     0.0 if_then_else@mpias                 0.3 Process 4
      0.0 Init/Exit                1.6 MPI_Wait                         0.3 Process 5
      0.0 Synchronization        0.0 if_then_else@mpias                 0.0 Process 6
    11.4 OMP                       1.1 MPI_Wait                         0.0 Process 7

8 x 2   Late Sender -> MPI_Wait -> :  00:09:17.304509 (1.56816%)  incl.
```

Fig. 4. Tracking down locations of significant late senders for ASPCG with 8×2 processors by taking advantage of the detailed control flow information in the middle tree.

7 Conclusions and Future Work

Large-scale parallel applications on advanced architectures with deep memory hierarchies rarely achieve even a moderate fraction of the theoretical peak performance. Our work provides a framework to automatically analyze and optimize the performance of hybrid MPI/OpenMP applications through integration of an optimizing compiler (OpenUH) with a performance analysis tool (KOJAK). While KOJAK automates the process of instrumenting MPI functions and OpenMP constructs, OpenUH automatically instruments user regions down to the flow-graph level at different compilation stages. The compile-time approach allows the application of a variety of optimizations before the instrumentation. Further, the results of static analysis can be used to help reduce the instrumentation overhead and the amount of trace data. KOJAK gives performance feedback at a significantly higher level than traditional tools. Such high-level information will enable the OpenUH compiler to adjust the performance model parameters and determine the most effective optimizing strategies to optimize parallel loops. The instrumentation of loops can produce a huge amount of event trace data when confronted with a deep loop nest. An approach to turn the instrumentation of loops on and off or use sampling mechanisms is being considered. The source-to-source translation approach of OPARI has some limitations: we hope to integrate the POMP instrumentation into later versions of the OpenUH compiler. Our future research will mainly focus on how to provide useful compiler-oriented feedback and on how the compiler can adapt optimization strategies accordingly. In this context, we plan to extend the current set of KOJAK performance properties to support advanced loop optimizations and scheduling, which might also include the monitoring of OpenMP runtime events beyond those specified in POMP.

8 Acknowledgments

We would like to thank Rick Kufrin at NCSA for giving us access to the SGI test platform and the team of Danesh Tafti at Virginia Tech for providing the ASPCG benchmark.

References

1. Ali-Reza Adl-Tabatabai. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7:19–31, 2003.
2. Eduard Ayguad, Bob Blainey, and Alejandro. Is the Schedule Clause Really Necessary in OpenMP? *Lecture Notes in Computer Science*, 2716:147–160, 2003.
3. Mihai Burcea and Michael J. Voss. A Runtime Optimization System for OpenMP. *Lecture Notes in Computer Science*, 2716:42–53, 2003.
4. W. Chen, R. Bringmann, and S. Mahlke et al. Using Profile Information to Assist Advanced Compiler Optimization and Scheduling. *Lecture Notes in Computer Science*, 757, 1993.

5. Francis H. Dang and Lawrence Rauchwerger. Speculative Parallelization of Partially Parallel Loops. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 285–299, 2000.
6. D. J. Hancock and J. Mark Bull et al. An Investigation of Feedback Guided Dynamic Scheduling of Nested Loops. In *ICPP Workshop*, 2000.
7. W. Nagel J. Hoeflinger, B. Kuhn. An Integrated Performance Visualzer for MPI/OpenMP Programs. In *Proc. of WOMPAT 2001*, West Lafayette, July 2001.
8. Josep Jorba, Tomas Margalef, and Emilio Luque. Automatic Performance Analysis of Message Passing Applications Using the KappaPI 2 Tool. *Lecture Notes in Computer Science*, 3666:293–300, 2005.
9. R. Kufrin. Perfsuite: An Accessible, Open Source Performance Analysis Environment for Linux. In *Proc. of the Linux Cluster Conference*, Chapel Hill, North Carolina, April 2005.
10. Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *12th Workshop on Compilers for Parallel Computers*, January 2006.
11. A. D. Malony and S Shende. Performance Technology for Complex Parallel and Distributed Systems. In P. Kacsuk and G. Kotsis, editors, *Quality of Parallel and Distributed Programs and Systems*, pages 25–41. Nova Science Publishers, Inc., New York, 2003.
12. B. Miller, M. Callaghan, and J. Cargille et al. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.
13. B. Mohr, A. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, August 2002.
14. W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 63, XII(1):69–80, 1996.
15. PERISCOPE. wwwbode.cs.tum.edu/ gerndt/home/research/periscope/periscope.htm.
16. C. Seragiotto Júnior, M. Geissler, G. Madsen, and H. Moritsch. On Using Aksum for Semi-Automatically Searching of Performance Problems in Parallel and Distributed Programs. In *Proc. of PDP 2003*, Genua, Italy, February 2003.
17. F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analysis. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.
18. G. Wang and D.K. Tafti. Performance Enhancement on Microprocessors with Hierarchical Memory Systems for Solving Large Sparse Linear Systems. *Int. J. of Supercomputing Applications and High Performance Computing*, 13(1):63–79, 1999.
19. F. Wolf and B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003.