An Open64-Based Framework Tool for Parallel Applications

Author1 and Author2

University of X

Abstract

We propose an infrastructure based on the Open64 compiler for analyzing, modeling and optimizing MPI and/or OpenMP applications. The framework consists of four main parts: a compiler, microbenchmarks, a user interface and a runtime library. The compiler generates the *application signature* containing a portable representation of the application structure that may influence program performance. Microbenchmarks are needed to capture the *system profile*, including MPI latency and OpenMP overhead. The user interface, based on Eclipse, is used to drive code transformation, such as OpenMP code generation. And lastly, our runtime library can be used to balance the MPI workload, thus reducing load imbalance. In this paper we show that our framework can analyze and model MPI and/or OpenMP applications. We also demonstrate that it can be used for program understanding of large scale complex applications.

1. Introduction

Analyses of MPI [**?**] and OpenMP [**?**] programs have been done for nearly a decade. There are two major types of analysis which have been used for this purpose: *static analysis* which is performed during compilation time and *dynamic analysis* which is performed during program execution. Some known techniques of dynamic analysis include *simulation*, where the program is executed in an emulated environment, and *program measurement/instrumentation*, where program behavior is captured with special libraries.

Some existing tools use dynamic analysis to perform analysis during the program execution [5, 25, 24, 19, 6, 4]. The dynamic measurement approach through instrumentation is probably the most accurate of all techniques. However, this technique is not without disadvantages. Instrumentation overhead may significantly perturb results and huge trace/event files may be generated. It is important to note that a program counter-based approach [2, 10] does not suffer from these problems but has somewhat limited applicability.

A simulation can also provide an excellent source for analysis [21]. Simulation enables an automated approach for assessing program performance under a variety of conditions. However, it takes an excessive amount of time and has limited accuracy.

Of all analysis approaches, static analysis has the lowest cost and is the fastest to perform since it does not need program execution. However, it must make assumptions about a program's control flow and the values of its data, and may not take the runtime environment properly into account.

Currently, there are several tools which have been developed to analyze MPI or OpenMP applications. For instance, Jumpshot [26] is a tool which can be used to visualize a state-based log trace file for MPI applications. It is unknown if the tool will also support OpenMP in the future.

The closest work on analyzing high performance scientific applications includes the Program Database Toolkit (PDT) [15]. PDT uses compile-time information to create a complete database of high-level program information that is structured for well-defined and uniform access by external tools such as SvPablo [8] and Tau [17]. Tau is an open source tool that provides a general solution to analyze and tune parallel programs. Tau is portable and provides an integrated framework for performance measurement, instrumentation, analysis and visualization based on information collected in a program database (PDB).

The POEMS project [1] is another framework which has been developed by different institutions to analyze, model and optimize parallel programs. It provides an environment for end-to-end performance modeling of complex parallel and distributed systems such as language, compiler, and runtime environment. This project took advantage of compiler analyses to optimize the execution time of its simulator so that some fraction of the program that has no impact on the performance can be skipped during the simulation and replaced by the analytical model.

The objective of our work is to develop an infrastructure to help programmers to analyze large scale parallel applications, to model performance behavior, to optimize MPI and OpenMP interaction and to ensure semantic correctness. In this paper, we describe how our framework can be used for program analysis, program understanding and some of its applications for reducing application load imbalance.

2. Methodology

Our framework adopts a static analysis approach by using the Open64 compiler for gathering the *application signature*. *Application signature* is a portable representation of the application structure that may influence program performance and program control-flow. This *application signature* includes program structure representations, such as call graph and program control flow, and is enhanced with the information concerning the system environment (Figure 1). One of the advantages of this approach is the ability to map between the analysis result and the source code so that programmers can identify the location of the potential problem accurately. Static analysis also has the advantage of not needing a program to be executed. This feature is critical for analyzing a large-scale application that may take days or even weeks to execute.

Our framework, Framework for OpenMP and MPI (FOMPI) (shown in Figure 2), is designed to analyze, model and optimize the parallel code of MPI, OpenMP or hybrid MPI+OpenMP. Our framework is based on Eclipse [23] as the main user

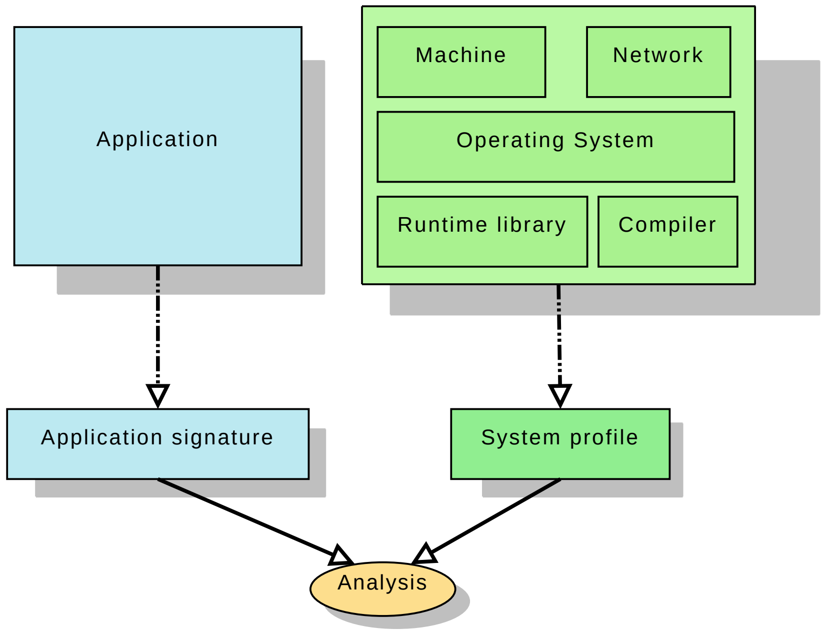


Figure 1: Analyzing applications approach by combining static analysis and runtime information.

interface, Open64 [16] as the source code analyzer and for collecting *application signature*, microbenchmarks for probing *system profile*, and load balancing library (LBL) for reducing load imbalance.

Our framework works as follows:

1. The Open64 compiler analyzes the source code from either MPI, OpenMP or the combination. This analysis is invoked through a special flag: `-apocost`. If more detailed information is needed, additional flags can be added. For instance, in order to retrieve information on the data scope of variables, the additional flag `-autoscope` is needed.

2. The compiler then generates an *application signature* which is a summary of program representations, memory models

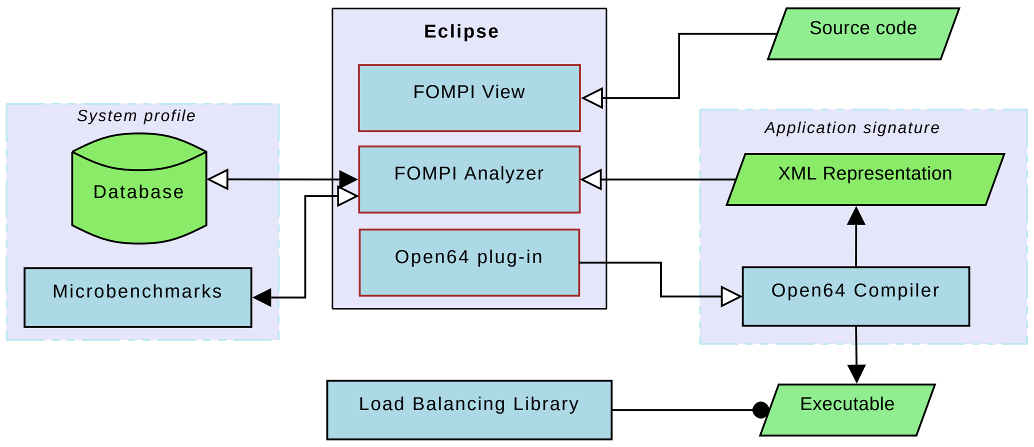


Figure 2: FOMPI framework for analyzing, modeling and optimizing MPI and/or OpenMP applications.

and arithmetic operations in XML format file. This file contains information on MPI communication routines used in the source code, OpenMP directives, control flow, call sites, loop information and parallelization status.

3. By using the *application signature*, it is possible to determine which loop has the potential to be parallelized. FOMPI also assists in parallelizing loops intelligently, and inserting instrumentation such as load balancing assistance.

4. Further, in order to accurately evaluate a hybrid code for a specific target machine, *system profile* such as an overhead measurement and cache analysis is needed. If the overhead measurement does not exist, the interface tool runs Sphinx to perform a new measurement and update the database. If the information of the cache behavior is unknown, the tool will request the information on the target machine through Perfsuite.

5. Using the information from *application signature* and *system profile*, FOMPI is able to construct a *performance modeling*. From this modeling the user will be able to *evaluate* if there is a OpenMP or MPI inefficiency.

6. Finally, the decomposition of MPI process and OpenMP threads for a specific target machine architecture can be determined. By using microbenchmark measurements, it is possible to determine the most efficient decomposition of processes and threads for a given problem size.

3. The Open64 Compiler

The Open64 compiler [16] is a well-written, modular compiler for Fortran, C and C++ which has been made increasingly robust in the last few years. Open64 was the open source version of SGI MIPSPro, a high-end commercial compiler developed by SGI, under the GNU Public License (GPL). Open64 is the final result of research contributions from a number of research compiler groups and industries around the world, such as Rice University, the University of Delaware, Tsinghua University, Intel, STMicroelectronics and Pathscale. As a derivative of Open64, Open64 also inherits advanced analyses and sophisticated optimization.

As shown in Figure 3, the major functional parts of Open64 are the Fortran 77/90, C/C++ and OpenMP front-ends, the inter-procedural analyzer/optimizer (IPA/IPO) and the middle-end/back-end. The back-end is further subdivided into the loop nest optimizer(LNO), auto-parallelizer (APO), global optimizer(WOPT), and code generator (CG). In addition, there is a complete analysis and optimizing framework available to facilitate further analyses and optimizations.

In order to exchange data between different modules, Open64 utilizes its own *intermediate representation* (IR), called Winning Hierarchical Intermediate Representation Language (**WHIRL**). As shown in Figure 3, WHIRL consists of five different levels:

1. **Very High Level** WHIRL (VHLW) serves as the interface between the front-end and the middle-end of the compiler. It contains some Fortran- and C-specific language representations which are still preserved in order to perform language

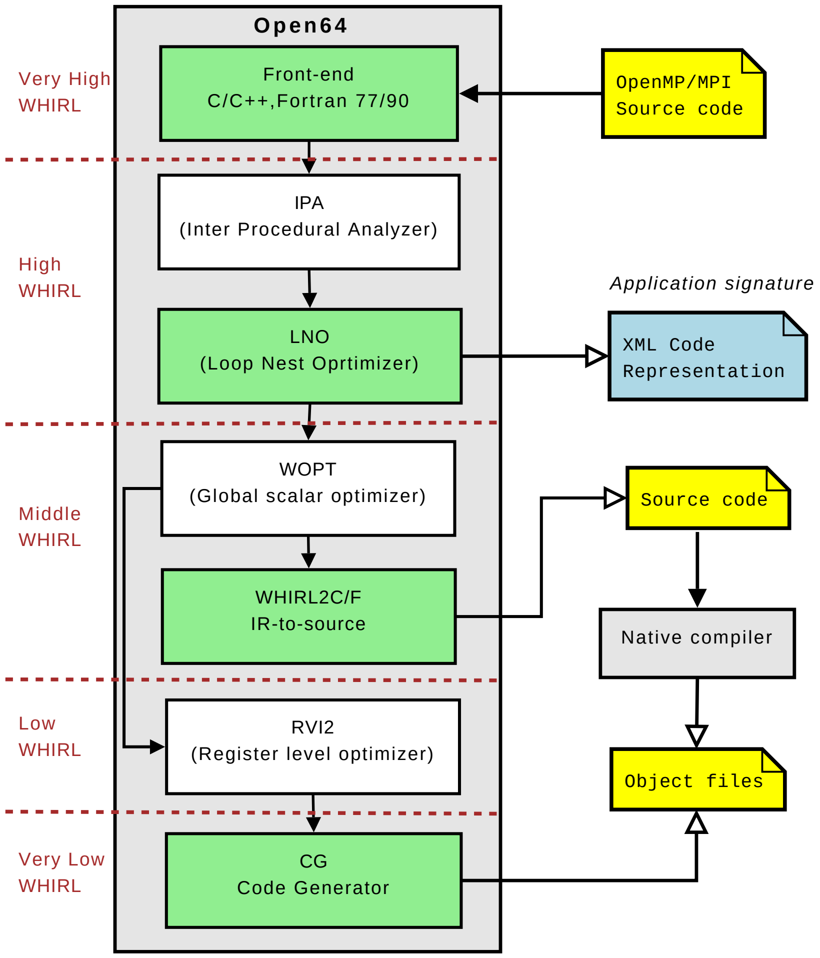


Figure 3: Open64 infrastructure

specific optimization.

2. **High Level** WHIRL (HLW) is used as the common interface among middle-end components such as LNO and IPA. It preserves high level control-flow constructs, such as loops and arrays, as well as OpenMP directives.

3. **Mid Level** WHIRL (MLW) is mainly the representation for WOPT components and the representation at this level starts to reflect the characteristics of the target architecture.

4. **Low Level** WHIRL (LLW) is used by the low-level optimizer and the representation is similar to code sequences generated in the target architecture.

5. **Very Low Level** WHIRL (VLLW) is the lowest level of WHIRL and has a very target-dependent representation; it happens before translation to code generation (CG).

We have added a new feature for Open64, implemented within the LNO module: an XML program representation which stores *application signature* of the source code of the application. There are some advantages for implementing these features in LNO. First, LNO is carried out after IPA, which can take advantage of higher analysis accuracy due to interprocedural analysis (IPA). Second, LNO uses High Level (HL) WHIRL which contains rich information about loops, the loop index and OpenMP directives (or pragmas in C/C++). Lastly, loop-based analysis, such as data dependence and array region analysis, is performed in this level. This enables us to easily manipulate and analyze data at the loop level. The only disadvantage of using this approach is that the analysis is performed after some general optimization, such as constant propagation and code hoisting, which may have slightly modified the original program structure.

4. Information Gathered

4.1. Application Signature

Application signature is a portable representation of the application structure that may influence program performance and program control-flow. We have extended the definition of *application signature* used in the PERC project [18] by including additional features such as program unit dependencies and program control-flow.

Like *intermediate representation*, *application signature* represents an application, but in an abstract fashion. Compared to *intermediate representation*, which contains very detailed information of the code structure, our *application signature* only stores indispensable information of code structure and the summary of the execution time inside the loop. Moreover, our *application signature* is language and architecture independent, so that it can be used for any imperative languages such as Fortran and C. Thus, the advantage of using *application signature*, instead of *intermediate representation*, is that we can maintain scalability for large-scale information without losing critical information.

We found that XML [12] is the most suitable format for storing *application signature* for several reasons. First, XML has an open format, is interoperable and supports hierarchical documents. Secondly, parsers already exist in XML, and some are optimized to handle huge files. However, XML also has disadvantages, since it is known to be slow and big [3]. Most XML files are bigger than any other format, especially the binary format. Moreover, since XML is a text-based format, it is slow to access. However, since XML is so widely used, many free-software developers and commercial vendors have put a lot of time into profiling and optimizing the programs that do low-level XML parsing.

The *application signature* generated by our compiler is stored in an XML format called the *XML program representation*. Our XML program representation is designed to support scalability, portability and extensibility. Although it can be used for purposes other than storing *application signature*, in this dissertation we use the terms *application signature* and *XML program representation* interchangeably.

In our work, we require the *application signature* to contain some information needed for our work, including MPI communications, loop parallelization status, data scope of variables and the estimated execution time as follows:

$$\begin{aligned}\text{Application\_Signature} = &\text{Memory\_Access} + \text{Arithmetic\_Operation} + \\ &\text{Control\_Flow} + \text{MPI\_Calls} + \text{OpenMP\_Directives} + \\ &\text{Parallelism\_Status} + \text{Autoscoping}\end{aligned}$$

The *application signature* is a "specification", and can be stored into any format, including database, binary, text or XML. We decided to use XML format for its openness, ease of access by other tools and the fact that its parser is available in most programming languages. However, since an XML format is based on pure text format, presenting all the details of program representation may affect scalability issues in manipulating the XML file.

To overcome this limitation, we do not store all program representations in XML files. Instead, we only store *blocks* of program sequences which are significant in either control execution or code execution of the application in the XML files.

Figure 5 provides an example of the *application signature*. This application is obtained from the NAS Fourier Transform (FT) source code shown in Figure 4. In Figure 5, we can see that the attribute `line number` is available for every block. This attribute is critical to map between the analysis and the source code so that programmers can modify the code easily. In addition, we can see the detailed insight of the MPI parameters. This information is particularly useful for MPI optimization, including overlapped communication and computation.

The XML file also shows detailed information on loops. This feature is important since the loop is perhaps the most critical block, considering that it is executed several times. For this reason, the XML file provides complete loop information, such as loop bounds (upper bound, lower bound and the estimated number of iterations), estimated execution time (estimated machine cycles, estimated cache cycles and estimated loop overhead) and parallelization status, including whether or not the

```
do j=1,1024
   q = mod(j, nx)+1
   if (q .ge. xstart(1) .and. q .le. xend(1)) then
      r = mod(3*j,ny)+1
      if (r .ge. ystart(1) .and. r .le. yend(1)) then
         s = mod(5*j,nz)+1
         if (s .ge. zstart(1) .and. s .le. zend(1)) then
            chk=chk+u1(q-xstart(1)+1,r-ystart(1)+1,s-zstart(1)+1)
         end if
      end if
   end if
end do
chk = chk/ntotal_f

 call MPI_Reduce(chk, allchk, 1, dc_type, MPI_SUM,
>                 0, MPI_COMM_WORLD, ierr)
```

Figure 4: An extract of NAS FT Benchmark

```
<loop Line="1699" opcode="OPC_DO_LOOP" >
 <header>
  <index>J</index>
  <lowerbound>1</lowerbound>
  <upperbound>1024</upperbound>
  <increment>+1</increment>
 </header>
 <cost>
  <iterations>1024</iterations>
  <average>31.1755</average>
  <machine>3776</machine>
  <cache>3458.25</cache>
  <overhead>4096</overhead>
  <total>11330.2</total>
 </cost>
 <parallel>
  <status>None</status>
  <reason> 1699: Not Parallel
Scalar dependence on CHK. Scalar CHK without unique last value.</reason>
  <scope>
   <private> J Q R S</private>
   <shared> XSTART XEND YSTART YEND ZSTART ZEND U1</shared>
  </scope>
 </parallel>
 <kids>
     . . .
 </kids>
</loop>

<mpicall name="mpi_reduce__" line="1713" args="8">
 <arguments>
  <arg type="SCALAR" name="CHK"/>
  <arg type="SCALAR" name="ALLCHK"/>
  <arg type="SCALAR" name="edef_B"/>
  <arg type="SCALAR" name="DC_TYPE"/>
  <arg type="SCALAR" name="f_B"/>
  <arg type="SCALAR" name="def_B"/>
  <arg type="SCALAR" name="def_B"/>
  <arg type="SCALAR" name="IERR"/>
 </arguments>
</mpicall>
```

Figure 5: An extract of **application signature** of the NAS FT benchmark.

loop is parallelizable.

4.2. System Profile

System profile is a collection of information concerning the platform where the program is executed. This information is application-independent and collected from microbenchmarks and then stored in a database, so that any programs and any users can access it. The concept of a microbenchmark, or synthetic benchmark, for collecting specific aspects of system performance has existed in the performance analysis literature for nearly four decades. A program of this form is intended to probe the system in order to reveal a specific performance characteristic such as network latency, bus bandwidth and cache structure.

System profile information consists of two main parts: (1) machine architecture, which is collected by Perfsuite; and (2) MPI latency and OpenMP overhead from Sphinx. Machine architecture is the information of the hardware, including memory hierarchy, cache size, cache line size and CPU speed. This information is collected by Perfsuite [**?**], a suite of tools developed by the National Center for Supercomputing Applications (NCSA). In order to measure MPI communication latency and OpenMP overhead, we use the Sphinx microbenchmark [**?**]. The result of these measurements is stored in the database and can be retrieved by any tools including our Eclipse plug-in (Section 6).

5. Eclipse Plug-in

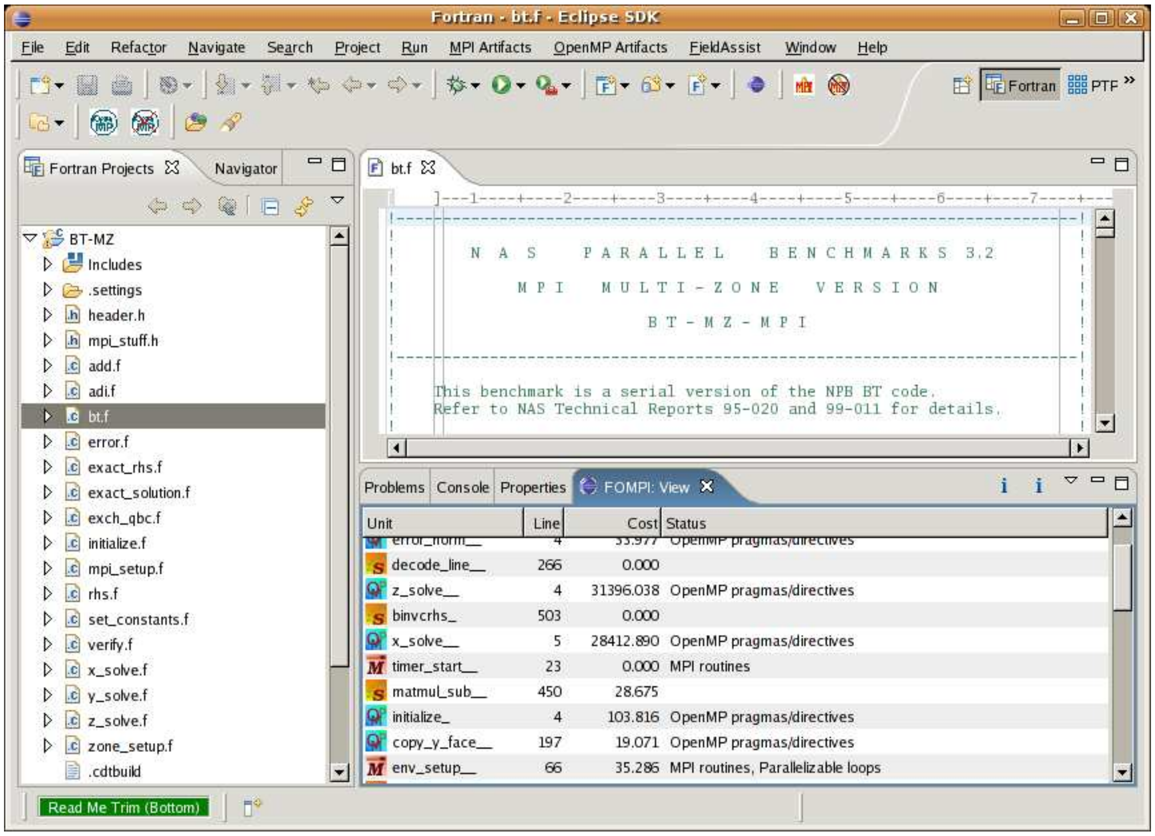
Eclipse is a platform based on the Java programming language. Therefore, it is portable to any operating systems. Eclipse is simply a framework which consists of a set of *plug-in* components for building a development environment. A plug-in is the smallest unit that can be developed and delivered separately. The Eclipse platform is built on a mechanism which can be used for integrating and running independent plug-ins. Usually a small tool is written as a single plug-in, whereas a complex tool has its functionality split across several plug-ins.

We have developed a new plug-in for interacting with the Open64 compiler, manipulating the XML program representation file, and allowing user interaction for application analysis, modeling and optimization. Furthermore, our plug-in provides functionalities to generate call graphs and control-flow graphs.

The FOMPI view plug-in is based on the Eclipse view model. A view is an Eclipse user interface part that is used to display the program property and additional information of the source code. The advantage of implementing an Eclipse plug-in as a view is that our plug-in can be used in any *perspective* [1]. In Figure 6, our FOMPI view is shown at the bottom part of the Eclipse GUI, where it is called within Eclipse Fortran perspective.

FOMPI view is a table consisting of four columns: the program unit, the line number where the program unit is defined, the estimated execution time of the unit and the status of the unit (whether it has MPI communication, OpenMP parallelization

[1]In Eclipse, a perspective defines the initial set and layout of views in the workbench window.



```
]---1----+----2----+----3----+----4----+----5----+----6----+----7----+---
!-------------------------------------------------------------------------!
!                                                                         !
!      N A S     P A R A L L E L     B E N C H M A R K S  3.2            !
!                                                                         !
!          M P I    M U L T I - Z O N E     V E R S I O N                !
!                                                                         !
!                      B T - M Z - M P I                                  !
!                                                                         !
!-------------------------------------------------------------------------!
!                                                                         !
!    This benchmark is a serial version of the NPB BT code.              !
!    Refer to NAS Technical Reports 95-020 and 99-011 for details.       !
!                                                                         !
```

Unit	Line	Cost	Status
error_norm__	4	33.977	OpenMP pragmas/directives
decode_line__	266	0.000	
z_solve__	4	31396.038	OpenMP pragmas/directives
binvcrhs_	503	0.000	
x_solve__	5	28412.890	OpenMP pragmas/directives
timer_start__	23	0.000	MPI routines
matmul_sub__	450	28.675	
initialize_	4	103.816	OpenMP pragmas/directives
copy_y_face__	197	19.071	OpenMP pragmas/directives
env_setup__	66	35.286	MPI routines, Parallelizable loops

Figure 6: FOMPI plug-in within the Eclipse platform. The plug-in is implemented as an Eclipse view and can be seen in the bottom part of the platform.

or contains a parallelizable loop). Currently, FOMPI view provides three menu items:

- **Load XML file**: to open an *application signature* file stored in XML format (as described in Section 4);
- **Create call graph**: to generate a call graph of the program (this will be described in Section 8.1); and
- **Create control-flow graph**: to create a control-flow graph of the selected program unit (see Section 8.2 for more details).

In order to use our plug-in, the user needs to generate the *application signature* either by using the Open64 plug-in or by compiling the source code with the flag `-apocost`. Once the XML file is generated, it has to be loaded via the **Load XML file** menu. Then a list of program properties from the *application signature* is displayed in the FOMPI view (as was seen in Figure 6). In order to generate a call graph and control-flow graph, an external library called *graphviz* is required. This library is open source and available for download at [13].

6 Load-Balancing Library

A load imbalance usually manifests itself as waiting time at synchronization points. An MPI program suffers from load imbalance if it has a small number of processes with less work than the majority, forcing the former to be in an idle state in order to synchronize with other processes.

One of the major benefits of OpenMP is its ability to reduce the negative impact of load imbalance in MPI code by providing a fine-grained decomposition of work [22]. Therefore, in addition to the Open64 compiler, Eclipse and microbenchmarks, our framework contains a load balancing library which is useful for optimizing MPI and hybrid MPI+OpenMP applications.

Some previous work has also adopted this approach, namely Dynamic Thread Balancing (DTB) [20] and Dynamic Processor Balancing (DPB) [7]. Unfortunately, DTB needs programmer intervention to add the library into the main iteration loop in the source code, whereas DPB can transform the code automatically by identifying the main iteration during runtime which causes significant overhead. We have designed a load balancing library which, unlike DTB, can transform the code automatically, and which, unlike DPB, performs the transformation prior to execution, hence reducing large overhead.

Our strategy is based on adjusting the number of OpenMP threads according to the MPI process workload. In contrast, if an MPI process has a higher workload than average, then OpenMP threads were added to reduce the workload. On the other hand, if a process has a low workload, then the number of threads were decreased.

6.1. Detecting Load Imbalance

Detecting load imbalance is performed during the program execution. In order to detect load imbalance in an MPI process, we follow the work of Gabriel *et al.* [11] which is defined as follows:

$$L_i = \frac{|t_i(N) - t(\bar{N})|}{t(\bar{N})}, \tag{1}$$

where $t_i(N)$ is the time spent in the MPI process, i is the Nth iteration, and $t(\bar{N})$ is the average time of all the MPI processes for the Nth iteration. Here, the value of i is varied from 1 to p which is the number of MPI processes. Without loss of generality, we assume the number of MPI processes to be constant.

The value of L_i is normalized from 0, meaning that there is no load imbalance at all, to the theoretically infinite. The bigger the value of L_i, the more load imbalance the process has. Therefore, when L_i exceeds a certain threshold, then the process is allowed to have an additional OpenMP thread to perform computation. Currently, the value of the threshold is still defined by the user. We plan to estimate by computing this threshold automatically based on the information of OpenMP overhead from our microbenchmark.

We have designed our library to minimize perturbation as much as possible. For instance, in order to collect L_i, the local computation time, from other MPI processes, we adopt overlapped communication and computation mechanism by using asynchronous MPI communication: `MPI_Isend` and `MPI_Irecv`.

6.2. Using LBL Library

Once the main iterative loop is identified, the next step is to insert the load-balancing library application programming interface (LBL API). There are three functions to be inserted:

1. `LBL_Init()`. This function prepares the initialization of the load-balancing library and is inserted before the main iterative loop.
2. `LBL_LoadDetection()`. This is the main function that verifies the existence of load imbalance and adds OpenMP threads if necessary. A more detailed description on how the load imbalance is detected is presented in Section 7.1.
3. `LBL_Finalize()`: This function performs the finalization step, such as cleaning up used variables inside the library.

LBL also supports functions to configure some parameters:

1. `LBL_SetSkipIterations(int)` to set the number of iterations to skip;
2. `LBL_GetSkipIterations()` to retrieve the current number of iterations to skip;
3. `LBL_SetThreshold(double)` to set the maximum percentage of load imbalance to be tolerated; and

```
1  #include <lbl.h>
2
3  int MainFunction () {
4
5    LBL_SetSkipIterations(40);
6    LBL_SetThreshold(30);
7    LBL_Init();
8    /* Main iteration */
9    while (iter<MAX_ITER) {
10     LBL_LoadDetection ();
11     . . .
12     Do_Computation ();
13     . . .
14     Do_Communication ( ) ;
15  }
16  /* end of main iteration */
17  LBL_Finalize ( ) ;
18
19 }
```

Figure 7: Example using the Load Balancing Library (LBL)

4. `LBL_GetThreshold()` to retrieve the current maximum percentage of load imbalance to be tolerated.

Figure 7 shows an example of using the library. First of all, the `lbl.h` header file, which contains function prototypes, needs to be included. Lines 5 and 6 set the configuration to determine the number of iterations to skip (line 5) and the threshold for the maximum percentage of load imbalance to be tolerated.

Once the `LBL_Init()` function in line 7 initializes the library just before entering the main iterative loop, load-balancing detection and optimization is then performed within `LBL_LoadDetection()` located inside the loop (line 10). Lastly, `LBL_Finalize()` marks the end of the LBL library by performing finalization, such as freeing variables.

6.3. Case Study

In order to validate our library, we examined a Jacobi code which has load imbalance where some processes have more workload than others; this is shown in Figure 8.

We ran our experiment on the SGI Altix with 256 Intel Itanium processors, which is a cache coherence Non-Uniform Memory Architecture (cc-NUMA) machine hosted in the National Center for Supercomputing Applications (NCSA). The advantage of using a cc-NUMA machine is that unlike a clustered SMP machine which can only accomodate a certain number of OpenMP threads depending on the number of processors inside the SMP box, a cc-NUMA machine is capable of supporting a flexible number of OpenMP threads.

As shown in Figure 9, our load-balancing library is able to reduce the load imbalance of Jacobi code by modifying the number of OpenMP threads during execution time (Figure 8). Consequently, it results in a marked decrease in overall execution time.

7. Program Understanding

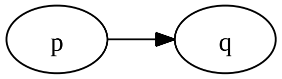
Program understanding is an approach to help programmers to understand software behavior or structure during the development and maintenance phases of the software life cycle. One of our research goals has been to develop a more effective program understanding tool for large-scale parallel applications. Although many program understanding tools already exist, we have noticed that the majority of the program understanding tools have not been adapted for complex large scale parallel applications, especially the MPI and OpenMP programs. In addition, they also do not have high scalability. In this section, we demonstrate that some FOMPI tools can be used for program understanding tailored for large-scale parallel applications.

7.1. Call Graphs

A Call Graph (CG) is a static representation of the dynamic invocation relationships between procedures in a program [14]. A call graph shows not only the interaction among procedures, but also the interaction with external libraries such as `printf`, `malloc`, etc. Our implementation of a call graph differs from the traditional call graph since ours only shows the interaction among procedures, hence increasing its scalability and user visibility.

A call graph $G_c = \langle N_c, E_c, s_c \rangle$ is a collection of a set of vertex N_c which represents a set of analyzable program units (also called procedures), a set of directed edges E_c which represents *call site* or procedure invocation from another procedure, and $s_c \in N_c$ is the initial vertex.

For instance, a call graph $G_c = \langle \{p, q\}, \{p \to q\}, p \rangle$ which represents the invocation of procedure q by procedure p, and is noted as $p \to q$, is graphically represented as:



.

Call graphs have been reported to be useful for program understanding, reverse engineering and for showing interaction between program units. We have designed a call graph to represent not only program dependency, but also the estimated execution time, the estimated number of call sites and the subroutine status. The subroutine status is useful to determine whether a subroutine contains communication or potential parallelization. By using the FOMPI XML code representation, it is possible to construct the call graph of the application shown in Figure 10. This diagram has been generated by our Eclipse plug-in via graphviz library [9], a free and open source graph visualization software package developed by AT&T.

Hence, we have extended the traditional call graph, not only by showing the interaction between *program units*, but also by including the estimated number of invocations of the program unit and the estimated execution time for an invocation. In order to improve scalability and enhance visibility for large-scale applications, the set of vertex N_c is limited only for

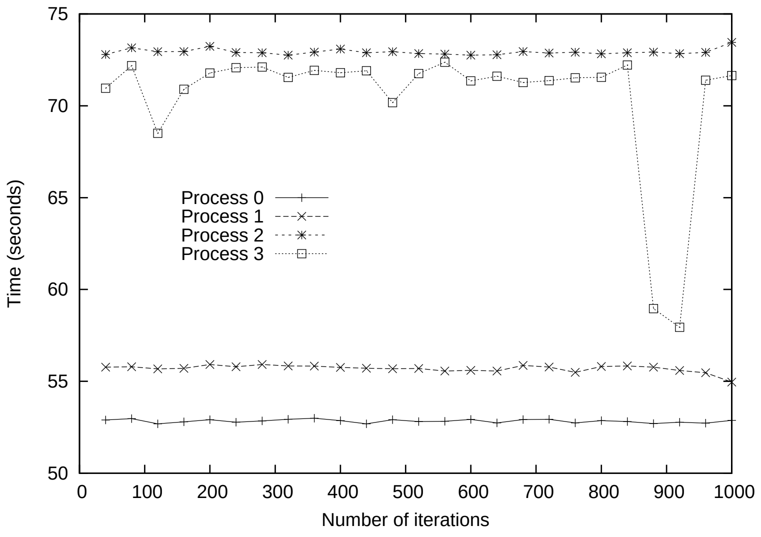


Figure 8: Execution time per process for original code

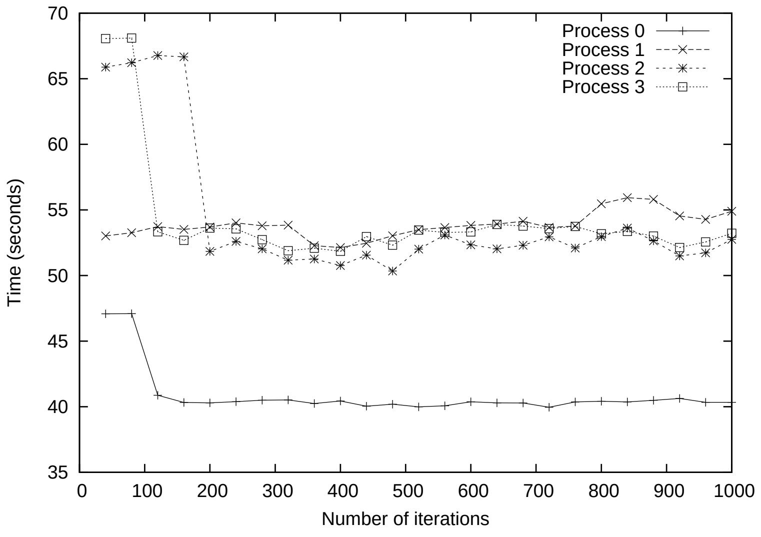


Figure 9: Execution time per process for balanced code

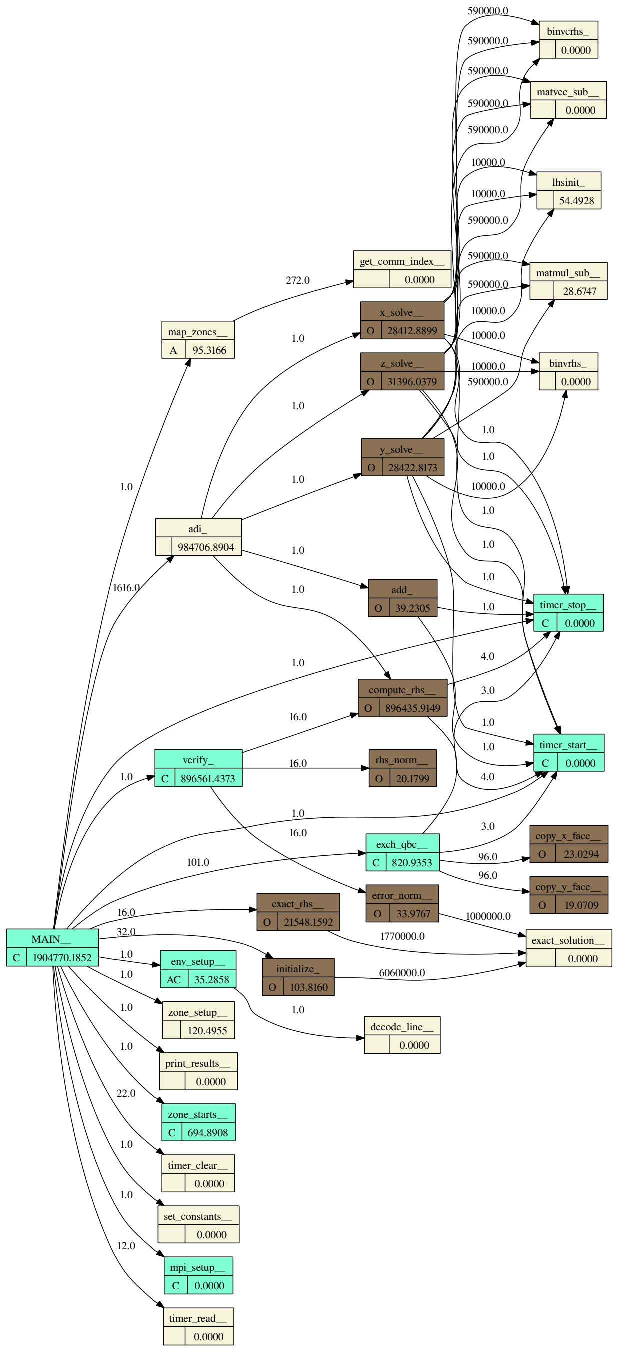


Figure 10: Call-graph of NAS BT Multizone benchmark. The label on the edge denotes the number of calls, while the figure within the node denotes the estimated number of cycles.

program units that are analyzed by our compiler. For instance, if a program unit invokes an external routine, say `printf`, this routine will not be included in our N_c.

Each node N_c in a call graph contains three fields: the program unit's name, the unit status and the estimated execution time of the unit. The cumulative estimated execution time N_c^t is the sum of the local execution time of the node (N_c^l) and the total of the estimated execution time of its call sites, as shown in Equation 2:

$$N_c^t = N_c^l + \sum_{i=0}^{n} \left(N_{c_i}^t \times E_c^i \right), \tag{2}$$

where n is the number of call sites, $N_{c_i}^t$ is the estimated execution time of the call site i and E_c^i is its number of invocations. For instance, it was seen in Figure 10, that the unit `exact_rhs` is executed approximately 16 times with the estimated execution time of 21548 cycles.

The directed edge E_c represents the caller-callee relationship, where the figure on the edge shows the number of invocations in the caller (also called *calling frequency*). Statically, the number of invocations of a call site can be estimated as the sum of the number of occurrences times the number of iterations, provided the call site occurs inside a loop:

$$E_c^t = \sum_i E_{c_i} \times l^t. \tag{3}$$

A summary of program unit status is represented by the letters **C**, **O** and **A** to mark whether the unit contains MPI communications, OpenMP parallelization and potentially parallelizable loops, respectively. For instance, as was shown in Figure 10, the `MAIN` unit, contains MPI communication since it is attributed by the letter **C**. This attribute is critical for some types of analysis, for instance, when an analysis is performed for the purpose of identifying the main iterations in a unit which has the most time-consuming loop containing both MPI communication and OpenMP parallelization.

7.2. Control-Flow Graphs

A control-flow graph (CFG) is a directed graph which represents code representation abstracting the control flow behavior of a program unit. A CFG $G_f = \langle N_f, E_f, s_f \rangle$ contains a set of nodes N_f, a set of directed edges E_f which represent control flow transfer from one node to another, and $s_f \in N_f$ which is the initial node. For instance, a CFG $G_f = \langle \{a, b\}, \{a \rightarrow b\}, a \rangle$ represents a control flow transfer from a to b.

In common with our call graph, our control-flow graph is designed for helping the programmer to understand the program structure of large-scale applications by summarizing some statements; this is shown in Figure 11. We also restrict the set of nodes N_f to only represent *basic blocks* that influence program control flow such as branches, and program performance such as loops. These *basic blocks* are defined in the XML program representation (Section 4).

Some *basic blocks*, such as **loops** and **branches**, can contain others called *children*. Furthermore, each *basic block* N_f

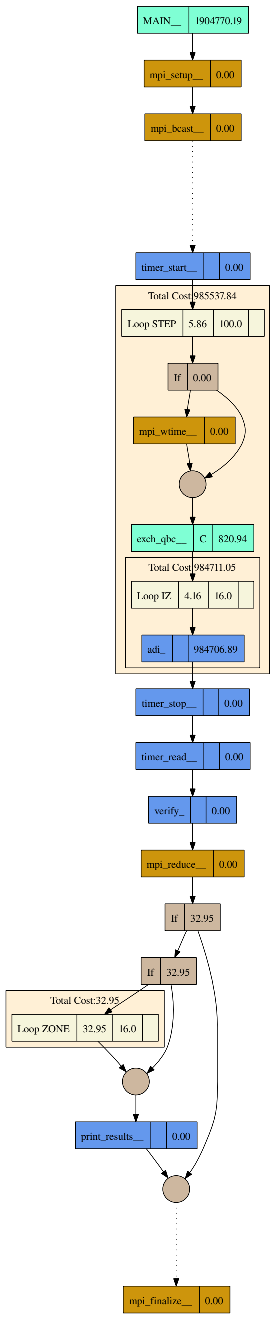


Figure 11: Control-flow graph of NAS BT Multizone benchmark in the main procedure. Each node contains an estimated execution time in cycles. A loop is represented by a subgraph.

has an estimated execution time N_f^t which is the sum of its own exclusive estimated execution time N_f^l and the total of the estimated execution time of its children (if they exist) as follows:

$$N_f^t = \begin{cases} N_f^l + \sum_{i=0}^{n} E_{f_i} \times N_{f_i}^t, & \text{inside loop;} \\ N_f^l + \max_{i=0}^{n}(\sum_{j=0}^{m_i} N_{f_{i_j}}^t), & \text{conditional branch;} \\ N_f^l, & \text{otherwise,} \end{cases} \tag{4}$$

where E_{f_i} denotes the number of invocations of the i^{th} child *basic block* $N_{f_i}^t$ and n is the number of children. For conditional branches such as the ones with the **if-then** statement, N_f^t is the maximum of the total estimated execution time of its children ($N_{f_{i_j}}^t$). In the case of other conditional branches such as the **if-then-else** statement, the maximum value of n is 2.

In Figure 11, we showed that we were able to abstract the control-flow graph of the main procedure in an NAS BT multizone benchmark. In this figure, each node represents a *basic block* such as branch, loop and call site, and each block is denoted by its estimated execution time. However, since a loop can be executed hundreds or even thousands of times, and therefore can have significant execution time, we used a *subgraph*, instead of a node to represent the loop body.

Finally, we designed our control flow graph so it does not show all available information. By eliminating detailed information such as assignment statements, not only can our control-flow graph support scalability, but it is also capable of generating a clearer graph compared to a traditional control-flow graph. This, combined with additional information, such as parallelization status and estimated execution time, enables the programmer to identify accurately which part of the code has to be taken into consideration for optimization.

8. Summary and Conclusions

We have presented FOMPI, the framework for analyzing MPI and/or OpenMP applications based on Open64 compiler. The *application signature* generated by our compiler has been proved to be critical for this purpose. This *application signature* is stored in an open format-based XML file which supports portability and interoperability so it be used for other external tools. It contains the complete and detailed information to analyze and optimize MPI or OpenMP applications, while maintaining scalability.

We have developed an Eclipse plug-in as the prototype of the user interface. The prototype can list all the program units, including the estimated execution time and the unit status needed to identify if a unit contains MPI communication, OpenMP regions or potentially parallelizable loops. From the user interface, different applications of FOMPI such as call graph, control-flow graph, OpenMP generation and MPI load imbalance reduction can be accessed. We have shown that the call graph and control-flow graph can improve program understanding thus enhancing reverse engineering.

Finally, we have also shown that our framework can be used to reduce MPI load imbalance using our load-balancing library

(LBL). The advantage of this approach is that a transformation can be performed automatically without adding significant overhead needed in other approaches.

References

[1] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, 2004.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM.

[3] P. Anderson. The performance penalty of xml for program intermediate representations. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 193–202, Washington, DC, USA, 2005. IEEE Computer Society.

[4] T. Brandes, H. Schwamborn, M. Gerndt, J. Jeitner, E. Kereku, M. Schulz, H. Brunst, W. E. Nagel, R. Neumann, R. Müller-Pfefferkorn, B. Trenkler, W. Karl, J. Tao, and H.-C. Hoppe. Monitoring cache behavior on parallel smp architectures and related programming tools. *Future Generation Comp. Syst.*, 21(8):1298–1311, 2005.

[5] H. Brunst, D. Kranzlmüller, and W. E. Nagel. Tools for scalable parallel program analysis - vampir vng and dewiz. In *DAPSYS*, pages 93–102, 2004.

[6] H. Brunst and B. Mohr. Performance analysis of large-scale openmp and hybrid mpi/openmp applications with vampirng. In *International Workshop on OpenMP (IWOMP)*, Eugene, Oregon, June 2005.

[7] J. Corbaln, A. Duran, and J. Labarta. Dynamic load balancing of mpi+openmp applications. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 195–202. IEEE, 2004.

[8] L. A. de Rose and D. A. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, page 311, Washington, D.C., USA, 1999. IEEE Computer Society.

[9] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2004.

[10] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM.

[11] E. Gabriel, M. Lange, and R. Ruhle. Direct numerical simulation of turbulent reactive flows in a metacomputing environment. *International Conference on Parallel Processing Workshops*, 00:0237, 2001.

[12] C. F. Goldfarb and P. Prescod. *The XML handbook*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[13] Graphviz. http://www.graphviz.org.

[14] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Lett. Program. Lang. Syst.*, 1(3):227–242, 1992.

[15] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, F. Juelich, R. Rivenburgh, C. Rasmussen, and B. Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[16] Open64. http://sourceforge.net/projects/open64/.

[17] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[18] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[19] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 63–72, Washington, D.C., USA, 2004. IEEE Computer Society.

[20] A. Spiegel, D. an Mey, and C. H. Bischof. Hybrid parallelization of cfd applications with dynamic thread balancing. In *PARA*, pages 433–441, 2004.

[21] J. Tao, M. Schulz, and W. Karl. A simulation tool for evaluating shared memory systems. In *ANSS '03: Proceedings of the 36th annual symposium on Simulation*, page 335, Washington, D.C., USA, 2003. IEEE Computer Society.

[22] H. W. and T. D. K. A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. In *Parallel CFD99, Williamsburg, VA*, May 1999.

[23] A. Weinand. Eclipse - an open source platform for the next generation of development tools. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, page 3, London, UK, 2003. Springer-Verlag.

[24] F. Wolf and B. Mohr. Automatic performance analysis of MPI applications based on event traces. *Lecture Notes in Computer Science*, 1900, 2001.

[25] F. Wolf and B. Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *J. Syst. Archit.*, 49(10-11):421–439, 2003.

[26] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with jumpshot. *Int. J. High Perform. Comput. Appl.*, 13(3):277–288, 1999.