SCALEA - A performance analysis system for distributed and parallel programs *

Hong-Linh Truong, Thomas Fahringer
Institute for Software Science
University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
{truong,tf}@par.univie.ac.at

April 30, 2001

Abstract

The current trend of developing hybrid parallel programs requires sophisticated performance analysis tools, that can examine multiple and possibly mixed styles of parallelism in particular shared and distributed memory parallelism. In this paper give an overview of SCALEA, which is a novel performance analysis systems for OpenMP, MPI, HPF, and mixed parallel/distributed programs. SCALEA instruments, executes and measures programs and computes a variety of performance overheads based on a novel overhead classification. Moreover, SCALEA uses a new representation of code regions, called dynamic code region call graph, which enables very accurate overhead analysis for generic code regions (ranging from the entire program to single statements). SCALEA supports profiling and tracing based on source code measurement libraries and HW-profiling interfaces. A prototype of SCALEA has been implemented. Several experiments with realistic codes that cover MPI, HPF, and mixed OpenMP/MPI codes, demonstrate the usefulness of SCALEA.

Keywords: performance, instrumentation, parallel & distributed program

1 Introduction

As hybrid architectures, in particular, SMP clusters become the mainstay of distributed and parallel processing in the market, the computing community is busily developing languages and software tools for such machines. Besides OpenMP [34], MPI [19], as well as HPF [21], also mixed programming paradigms such as OpenMP/MPI are increasingly being evaluated. In this paper we introduce SCALEA which is a new performance analysis system for distributed and parallel programs that covers all of the above mentioned programming paradigms including mixed programs. Most existing performance analysis tools have been developed for specific programming paradigms and are difficult to adjust for new languages. SCALEA is based on a novel classification of performance overheads that covers both shared and distributed memory parallelism including data movement, synchronization, control of parallelism, additional computation, loss of parallelism, and unidentified overheads. Specific instrumentation and performance analysis is conducted to determine each category of overhead for individual code regions. Instrumentation can be done fully automatically or user controlled through directives. Post-execution performance analysis is done based on performance trace-files and a novel representation for code regions named dynamic code region call graph (DRG). The DRG reflects the dynamic relationship between code regions and its subregions and is used to determine a breakdown of performance overheads for every code region. This is in contrast to existing approaches that use the conventional call graph which considers only function calls but not arbitrary code regions.

*The work described in this paper was supported by the Special Research Program SFB F1104 "AURORA" of the Austrian Science Fund.

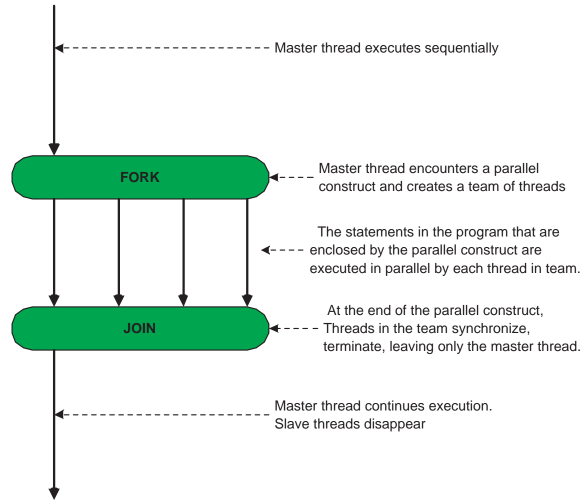


Figure 1: Execution model of OpenMP programs.

The rest of this paper is structured as follows: Section 2 presents the execution model of OpenMP, MPI, HPF+ and hybrid parallel programs. Section 3 describes an overview of SCALEA [31]. In Section 4, we present methodology of automatic instrumentation and overhead determination. Performance analysis is shown in Section 5. In Section 6 we present some experiments. The related works and conclusion are presented in Section 7 and 8, respectively.

2 Execution model

In this section, we present the execution model of OpenMP, MPI, MPI+OpenMP, HPF+, and HPF+ mixing OpenMP programs in our framework.

2.1 Shared memory with OpenMP program

OpenMP is a parallel programming model for shared memory and distributed shared memory on multiprocessor. OpenMP is not new computer language; it works in conjunction with existed languages such as C/C++, Fortran, and Java. It consists of a set of directives that describe the shared memory parallelism in the source codes, a small set of runtime library routines and environment variables.

OpenMP uses the fork-join model of parallel execution[7][11]. Figure 1 shows the execution model of OpenMP programs.

As we can see, an original OpenMP program begins execution as a single process, referred to as the master thread of execution. The master thread executes sequentially until it encounters the first parallel construct. In OpenMP specification, the PARALLEL/END PARALLEL directive pair constitutes the parallel construct. The master thread creates some additional threads, usually called slave threads, forms a team of threads when a parallel construct is encountered and the master thread becomes the master of the team. The statements in the program that are enclosed by the parallel construct, including routines called from within the enclosed statements, are executed in parallel by each thread in the team. Upon completion of the parallel construct, the threads in team synchronize and only the master thread continues execution. Any number of parallel constructs can be specified in a single program so that a program may fork and join many time during execution.

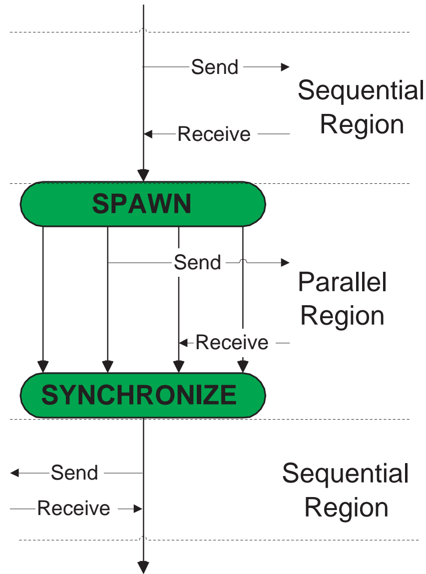


Figure 2: Execution model of a single process in MIMD parallel/distributed program

2.2 Message passing using MPI

In message passing model, each process has a private address space that could not be accessed by other processes. A process must communicate explicitly with other processes to access their data. Communication is performed via send and receive task so that processes in both sides are involved explicitly. The sender knows and specifies whom to send data. Typically, data is packed and unpacked at each end for efficient transfer and for adapted with heterogeneous network.

We use MPICH [18], the portable implementation of MPI [19] for our experiment.

2.3 Hybrid parallel model

SCALEA supports performance analysis for MIMD (multiple-instructions-multiple-data) parallel and distributed programs based on a mixed execution model comprising both distributed and shared memory parallelism. Figure 2 shows the execution model of a generic process with its own address space in a MIMD program. The process executes a program which is subdivided into sequential and parallel regions. A process may dynamically spawn/fork, synchronize, and terminate threads during execution of the program. All threads of the process share the same address space. In a sequential region only one thread within the process is active (executes the code region). In a parallel region several threads may be active and execute simultaneously. The distribution of the computational load among active threads within a parallel region is language dependent. Threads may be spawned at the beginning of the program or at the beginning of the parallel region which is language/implementation dependent. At the end of the parallel region the active threads may be synchronized (e.g. through a barrier synchronization or join operation); moreover, all but one – who continues to execute the following region – will either be terminated or turned passive (do not execute a region but remain alive) at the end of a parallel region. A passive thread can be turned active in a subsequent parallel region or terminated at the end of the program execution. Active threads within the same process exchange data by using the shared memory. Exchanging data between different processes is enabled only between active threads – associated with different processes – through generic SEND and RECV operations which can be executed in both sequential and parallel code regions.

Our current implementation of SCALEA uses mostly OpenMP[34] as the shared memory programming language and MPI [19] for message passing. Message passing operations are currently only allowed in sequential regions due

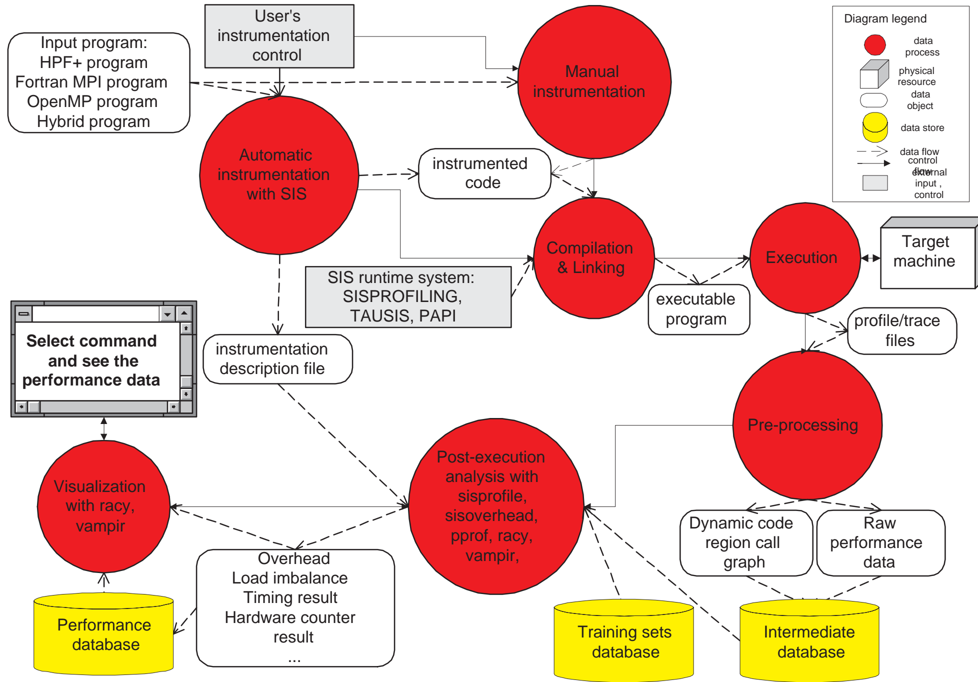


Figure 3: Architecture of SCALEA

to limitations of current OpenMP and MPI implementations.

2.4 HPF+ and HPF+/OpenMP

In our framework, HPF+ [5] , HPF+/OpenMP code will be translated into Fortran90/MPI and Fortran90 MPI/OpenMP by VFC compiler [3], respectively. Thus, the execution model of HPF+ and HPF+/OpenMP follow to the execution model of MPI and OpenMP/MPI, respectively, as discussed in section 2.2 and section 2.3.

3 SCALEA overview

SCALEA is a post-execution performance tool that instruments, measures, and analyses the performance behavior of distributed memory, shared memory, and hybrid parallel (mixed version that combines both distributed and shared memory parallelism) programs.

Figure 3 shows the architecture of SCALEA which consists of two main components: *SCALEA* instrumentation system(SIS) and a post execution performance analysis tool set. *SIS* is integrated with VFC [4] which is a compiler that translates Fortran programs into Fortran90/MPI or mixed OpenMP/MPI programs. The input programs (Fortran90, HPF, MPI, OpenMP, and mixed programs) of SCALEA are processed by the compiler front-end which generates an abstract syntax tree (AST). SIS enables the user to select (by directives or command-line options) code regions of interest. Based on pre-selected code regions, SIS automatically inserts probes (instrumentation code) in the code which will collect all relevant performance information in a set of profile/trace files during execution of the program on a target architecture. SIS also generates an *instrumentation description file* that enables to relate all gathered performance data back to the input program. Basically the instrumentation description file contains a unique probe identifier for every instrumented code region, including information about the performance data gathered during execution of the program. SIS is based on two instrumentation libraries: SISPROFILING and

TAUSIS. SISPROFILING supports profiling (timers, counters, and hardware parameters) and creates the dynamic code region call graph (see Section 4.4). Hardware parameters are determined through an interface with PAPI [8]. TAUSIS is based on the TAU instrumentation library [24] and is mainly used for tracing. The main functionality of SIS is given as follows:

- Automatic instrumentation of pre-defined code regions (loops, procedures, HPF INDEPENDENT loops, I/O statements, etc.) for various performance overheads by using command-line options.
- Manual instrumentation through SIS directives which are inserted in the program. These directives also allow to define user defined code regions for instrumentation and to control the instrumentation overhead and the size of performance data gathered during execution of the program.

A preprocessing phase of SCALEA filters and extracts all relevant performance information from the profile/trace files which yields filtered performance data and the dynamic code region call graph (DRG). The DRG reflects the dynamic relationship between code regions and its subregions and is used for a precise breakdown of performance overheads for every individual sub-region. This is in contrast to existing approaches that are based on the conventional call graph which considers only function calls but not arbitrary code regions.

SCALEA supports a sophisticated performance overhead analysis based on the DRG and a classification of performance overheads. Post-execution performance analysis also employs a training set method to determine specific information (e.g. time penalty for every cache miss overhead, overhead of probes, time to access a lock, etc.) for every target machine of interest.

4 Automatic instrumentation and overhead determination

In this section, we present the methodology of *SCALEA* that is applied in performance analysis of parallel and distributed programs. Our goal is to determine as much as possible performance problems of programs. To achieve it, we classify sources of performance problems under the temporal overhead classification. Basically overheads of a portion code of a given program depends on characteristics of this code(e.g. number of PARALLEL regions, MPI calls, etc.). Therefore, *a code region analysis* is applied to parallel and distributed programs in order to breakdown overheads of the programs. In our code region analysis method, a program is modeled into set of pre-defined code regions and user-defined code regions(arbitrary code regions). And then correspondent instrumentation methods will be applied for code regions and a dynamic code region call graph(DRG) is built.

In our current implementation, SIS [31] is used to analyze and instrument programs. Different techniques are applied to different types of code regions. SISPROFILING [31] is used to capture raw performance data and create the DRG.

4.1 Classification of Temporal Overheads

According to Amdahl's law [1], theoretically the best sequential algorithm takes time T_s to finish the program, and T_p is the time required to execute the parallel version with p processors. The temporal overhead of a parallel program is defined by $T_o = T_p - T_s/p$ and reflects the difference between achieved and optimal parallelization. T_o can be divided into T_i and T_u such that $T_o = T_i + T_u$, where T_i is the overhead that can be identified and T_u is the overhead fraction which could not be analyzed in detail. Theoretically, T_o can never be negative, it means that the speedup T_s/T_p can never exceed p [22]. However, in practice it is possible for temporal overhead to be negative due to the super linear speedup of the application.

In figure 4 we give a classification of temporal overheads based on which the performance analysis of SCALEA is conducted:

- *Data movement* corresponds to any data transfer within a single address space of a process (local memory access) or between processes (remote memory access).
- *Synchronization* (e.g. barriers and locks) is used to coordinate processes and threads when accessing data, maintaining consistent computations and data, etc.

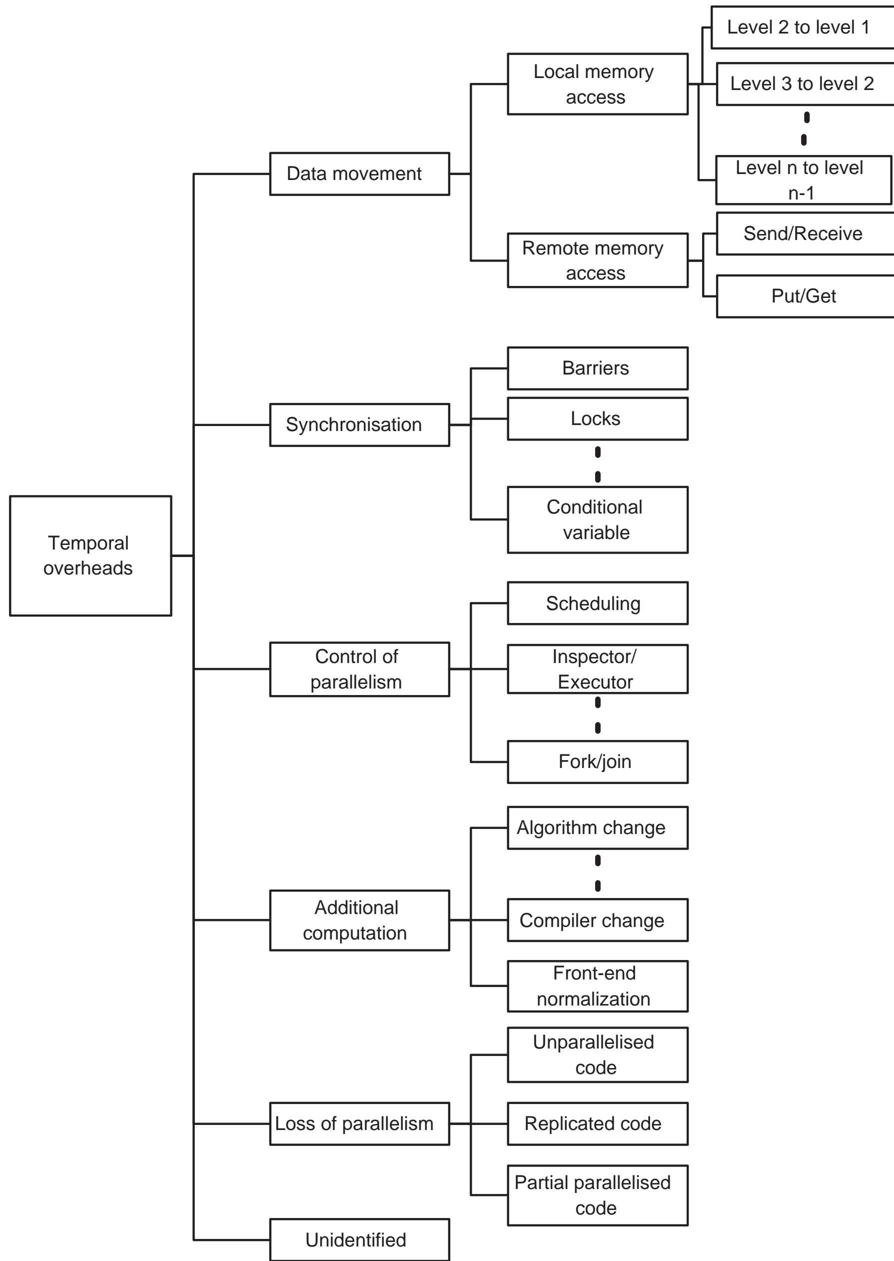


Figure 4: Temporal Overheads classification.

- *Control of parallelism* (e.g. fork/join operations and loop scheduling) is used to control and manage the parallelism of a program and can be caused by user, compiler and runtime library operations.
- *Additional computation* reflects the any change to the original sequential program including algorithmic or compiler changes to increase parallelism (e.g. by eliminating data dependences) or data locality (e.g. through changing data access patterns).
- *Loss of parallelism* is due to imperfect parallelization of a program which can be further classified into: unparallelised code (executed by only one processing unit), replicated code (executed by all processing units), and partially parallelized code (executed by more than 1 but not all processing units).
- *Unidentified overhead* corresponds to the overhead that is not covered by the above categories.

Note that the above mentioned classification has been stimulated by [9] but differs in several respects. In [9], synchronization is part of information movement, load imbalance is a separate overhead, local and remote memory accesses are merged in a single overhead class, loss of parallelism is split into two classes, and unidentified overhead is not considered at all. Load imbalance in our opinion is not an overhead but represents a performance property that is caused by one or more overheads.

4.2 Code regions

When tuning a program, the users focus on some portions of their program that they are interest. The portions chosen by the users can be loops, function calls, or arbitrary codes.

The control flow of a program is represented by a graph $(CFG)G = (N, E, s, e)$ with set of nodes(statements) N and set of edges E. Nodes s and e respectively denote the unique *entry* and *exit* node of G, which are assumed that not to possess any predecessors and successors. An edge $(n_1, n_2) \in E$ indicates transfer of control between nodes(statements) n_1, n_2 $\in N$.

> *a code region* R is a set of nodes(statements) $\{n_1, n_2, ..., n_k\}$ with a subset $\{n_{e_1}, n_{e_2}, ..., n_{e_m}\}$ of entry nodes(statement) and a subset $\{n_{x_1}, n_{x_2}, ..., n_{x_w}\}$ of exit nodes(statements) that satisfies the following conditions:
>
> $\forall n \in \{n_{e_1}, n_{e_2}, ..., n_{e_m}\}$, $\exists\, n'$ such that $n' \in N \; and \, n' \notin R \; and \; (n', n) \in E$. In a special case if $n = s$ then n is a entry node of R.
>
> $\forall n \in \{n_{x_1}, n_{x_2}, ..., n_{x_m}\}$, $\exists\, n''$ such that $n'' \in N \; and \, n'' \notin R \; and \; (n, n'') \in E$. In a special case if $n = e$ then n is a exit node of R.

In order to measure the execution behaviour of a code region, probes will be inserted at entry and exit nodes of the code region.Therefore, the instrumentation system has to detect all entry and exit nodes of a code region and perform instrumentation task by inserting instrumentation code at these nodes. Basically, this task can be done with the support of the compiler. Figure 5 shows an example of a code region with its entry and exit nodes. To select an arbitrary code region, the user marks two statements as the entered and exited statements of the code region. It is clearly that the entered and exited statement of the selected code region are entry and exit node of the code region, respectively. Each node represents a statement in the program. A code region consists of all statements in the program unit that are enclosed by the entered and exited statement of the code region. In figure 5, there are many entry and exit nodes of the selected code region. The instrumentation system will detect all these nodes and perform instrumentation by inserting probes before entry nodes and after exit nodes, respectively.

Code regions can be overlapping by their definition. In SCALEA we don't support to measure code regions that are overlapping. In addition, the current implementation of SCALEA supports mainly instrumentation code regions that have single entry-single exit node. However, code regions that have many exit statements such as STOP, EXIT, RETURN also are supported. We are going to enhance SIS to support fully multiple entry-multiple exit code regions.

In our framework, we have a collection of predefined code regions. The predefined code regions are classified into common code regions and specific code regions. By saying common code regions we mean code regions that are in almost programs regardless to the programming model these programs belong to. For examples, main program,

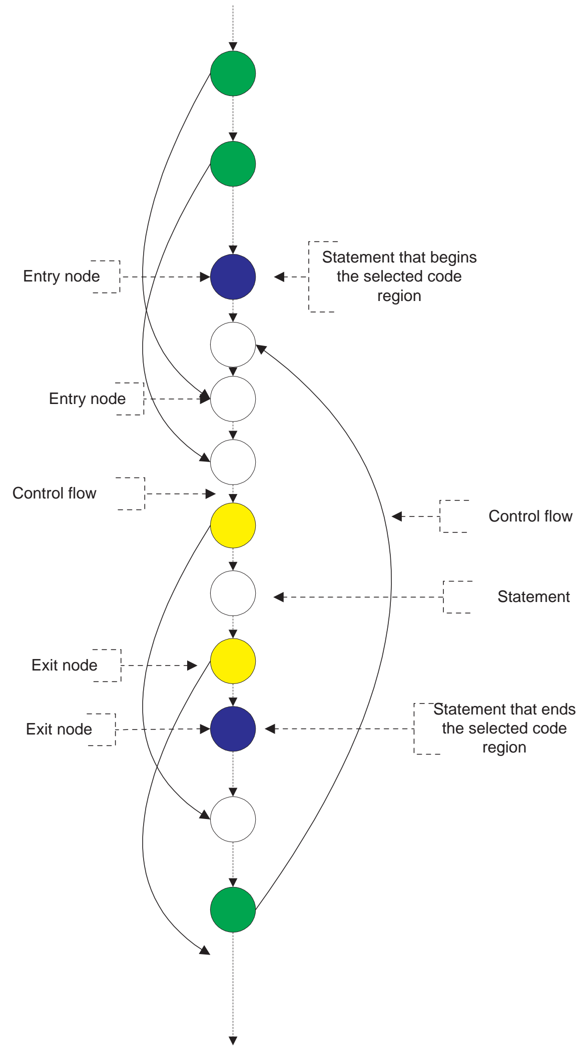


Figure 5: A model of a code region with its entry and exit nodes.

loops, procedures, and so on are in all program. In contrast, the second category of code regions stays within a given programming model. Such code regions are INDEPENDENT loops of HPF+ programs, OpenMP parallel constructs, or MPI calls. A shortened list of code regions can be found at [31]. In shared memory programming model with OpenMP, OpenMP specific code regions are OpenMP constructs [7] such as parallel constructs, work sharing construct, synchronisation constructs, etc. In message passing programming model with MPI, specific code regions are MPI routine calls.

Although, pre-defined code regions cover almost code of programs but it doesn't fulfill the requirements of users. Therefore, directives that are used to describe a arbitrary code regions are supplied.

Depending on the user's requirements, different instrumentation technique is applied for different code region. To show it clearly, let take an example how SCALEA instruments OpenMP PARALLEL region. If the user wants to measure fork/join overhead(control of parallelism overhead) SCALEA will divide OpenMP parallel code region into three parts: the first part consists of PARALLEL directive, the second one consists of the statements enclosed by PARALLEL and END PARALLEL directive and the last one consist of END PARALLEL directive. On the other hand, if the user wants to measure only wall clock time of PARALLEL region in other to see the load balance of work in all thread SCALEA just instruments the code enclosed by PARALLEL directives.

4.3 SCALEA Instrumentation System

Based on user-provided command-line options or directives, SIS inserts instrumentation code in the program which will collect all performance data of interest. SIS supports the programmer to control profiling/tracing and to generate performance data through selective instrumentation of specific code region types (loops, procedures, I/O statements, HPF INDEPENDENT loops, OpenMP PARALLEL loops, OpenMP SECTIONS, OpenMP CRITICAL, MPI barrier statements, etc.). SIS also enables instrumentation of arbitrary code regions. Finally, instrumentation can be turned on and off by a specific instrumentation directive.

4.3.1 Using command line options

Instrumentation of code regions can be controlled through SIS directives and command-line options. Command-line options are used to specify one or several classes of predefined code regions, for which probes should be inserted.

With command line options SIS will analyze *entire program* and perform instrumentation task, for example if the user choose an option to instrument loops in his program, SIS will find all loops in the program and instrument them. Basically, he could not *choose a specified loop* via command line options. However he can use SIS directives (see section 4.3.2) to specify portions of program that SIS should analyze.

By using command line options, user can not specify the performance metrics he wants to measure (e.g execution time, floating point instructions, or overhead of loss parallelism) at the instrumentation step. To allow the user choosing measurement metrics(timing or hardware metrics), SIS runtime system provides execution environment variables. Thus, it affects entire program, for instance user could not measure one code region with timing metrics and the other one with hardware metrics (both code regions must be measured the same metrics). Besides, temporal overheads depend on selected code regions so that it requires user the knowledge of code regions. Therefore, we are going to support more flexible way to specify code regions and temporal overheads by using SIS directives (see section 4.3.2). This is in contrast with the approach as usual to support only the same timing and/or hardware metrics for every instrumented code regions in a program.

4.3.2 Using SIS directives

SIS directives [31] can be manually inserted in a code that allow:

(1)*Specify arbitrary code regions*: directives are used to describe arbitrary code regions.

(2)*Dynamic switch on/off tracing instrumentation*: directives are used to disable and enable instrumentation process at the runtime.

(3)*Specify code regions & related performance metrics*: directives allow users to specify both code regions and performance metrics that they want to measure. Performance metrics can be timing metrics

(e.g. wall clock time, CPU time, ...), hardware metrics (such as floating point operations, cache misses,...) or temporal overhead (for instance loss of parallelism, additional overhead, ...).

With SIS directives, the user not only can specify different performance metrics for different code regions but also can restrict portion of code to be analyzed, profile/trace data size. In the current prototype of SCALEA, we have supported (1) and (2) whereas We are going to implement (3).

In order to measure arbitrary code regions SIS provides the following instrumentation:

```
!SIS$ CR_BEGIN
      code region
!SIS$ CR_END
```

The directive !SIS$ CR_BEGIN and !SIS$ CR_END must be, respectively, inserted by the programmer before and after the region starts and finishes. S. Note that there can be several entry and exit nodes for a code region. Appropriate directives must be inserted by the programmer in every entry and exit node of a given code region. Alternatively, compiler analysis can be used to automatically determine these entry and exit nodes.

Furthermore, SIS provides specific directives in order to control tracing/profiling. The directives MEASURE_ENABLE and MEASURE_DISABLE allow the programmer to turn on and off tracing/profiling of a program.

```
!SIS$ MEASURE_ENABLE
      code region
!SIS$ MEASURE_DISABLE
```

For instance, the following example instruments a portion of an OpenMP pricing code version (see Section **??**), where for the sake of demonstration, the call to function *RANDOM_PATH* is not measured by using the facilities to control profiling/tracing as mentioned above.

```
!SIS$ CR_BEGIN
      !$OMP DO PARALLEL PRIVATE(PATH) REDUCTION (+:V)
      DO I = 1, N
!SIS$   MEASURE_DISABLE
          PATH = RANDOM_PATH(0,0,N)
!SIS$   MEASURE_ENABLE
          V = V + DISCOUNT(0,CASH_FLOW(B,1,N),
                           FACTORS_AT(PATH))
      END DO
      PRICE = V/N
!SIS$ CR_END
```

Note that SIS directives are inserted by the programmer based on which SCALEA automatically instruments the code.

4.3.3 Instrumentation Description File (IDF)

A crucial aspect of performance analysis is to relate performance information back to the original input program. During instrumented of a program, SIS generates an *instrumentation description file* (IDF) which correlates profiling, trace and overhead information with the corresponding code regions. The IDF maintains for every instrumented code region a variety of information (see Table 1).

A code region type describes the type of the code region, for instance, entire program, outermost loop, read statement, OpenMP SECTION, OpenMP parallel loop, MPI barrier, etc. The program unit corresponds to a subroutine or function which encloses the code region. The IDF entry for performance data is actually a link to a separate repository that stores this information. Note that the information stored in the IDF can actually be made a runtime data structure to compute performance overheads or properties during execution of the program. IDF also helps to keep instrumentation code minimal, as for every probe we insert only a single identifier that allows to relate the associated probe timer or counter to the corresponding code region.

IDF Entry	Description
id	code region identifier
type	code region types
file	source file identifier
unit	program unit identifier that encloses this region
line_start	line number where this region starts
column_start	column number where this starts
line_end	line number where this ends
column_end	column number where this ends
performance_data	performance data collected or computed for this region
aux	auxiliary information

Table 1: Contents of the instrumentation description file (IDF)

4.4 Dynamic code region call graph

SCALEA can be used to instrument OpenMP, MPI, HPF+ and hybrid parallel programs compiled with VFC compiler [3] automatically. SIS [31] classifies code regions of a parallel program into *common code regions* and *specific code regions*. Through set of execution commands, the instrumentation system of *SCALEA*(SIS) is controlled to instrument any code region such as MPI point-to-point communication routine, OpenMP PARALLEL region (specific code regions), loop, function call (common code regions), etc[31]. An example of code regions of an OpenMP/MPI program is shown in figure 6.

HPF+ programs are only translated to only Fortran 90 MPI programs following to the SPMD (Single Program Multiple Data) model. When the execution of the instrumented program finishes, a dynamic code region call graph can be obtained by processing profile/trace files.

4.4.1 What is the Dynamic code region call graph?

Every program consists of a set of *code regions* which can range from a single statement to the entire program. A code region can be, respectively, entered and exited by multiple entry and exit control flow points (see Figure 5). In most cases however, code regions are single-entry-single-exit code regions. SCALEA has a set of predefined code regions which are classified into common (e.g. program, procedure, loop, function call, statement) and programming paradigm specific code regions (MPI_calls, HPF INDEPENDENT loops, OpenMP parallel regions, loops, and sections, etc.). Moreover, SIS provides directives to define arbitrary code regions in the input program. Based on code regions we can define a new data structure called dynamic code region call graph (DRG):

> A dynamic code region call graph (DRG) of a program Q is defined by a directed flow graph $G = (R, E, s)$ with a set of nodes R and a set of edges E. A node $r \in R$ represents a code region which is executed at least once during execution of Q. An edge $(r_1, r_2) \in E$ indicates that a code region r_2 is called inside of r_1 during execution of Q and r_2 is a dynamic sub-region of r_1. The first code region executed during execution of Q is defined by s.

The figure 7 shows a sample of OpenMP program and its dynamic code region call graph.

We have five code regions: R_1 is *entire program*, R_2 is *sequential code region*, R_3 is the *OMP PARALLEL* region, R_4 is *OMP DO* code region and R_5 is *OMP CRITICAL* code region. The dynamic code region call graph that is created after the execution of the application finished is shown in figure 7. Notice that dynamic code region call graph is different from syntactic structure of code regions. Only code region has been instrumented and executed is in dynamic code region call graph. Notice that the timing overhead of a code region r with n sub-regions $r_1, ..., r_n$ is given by:

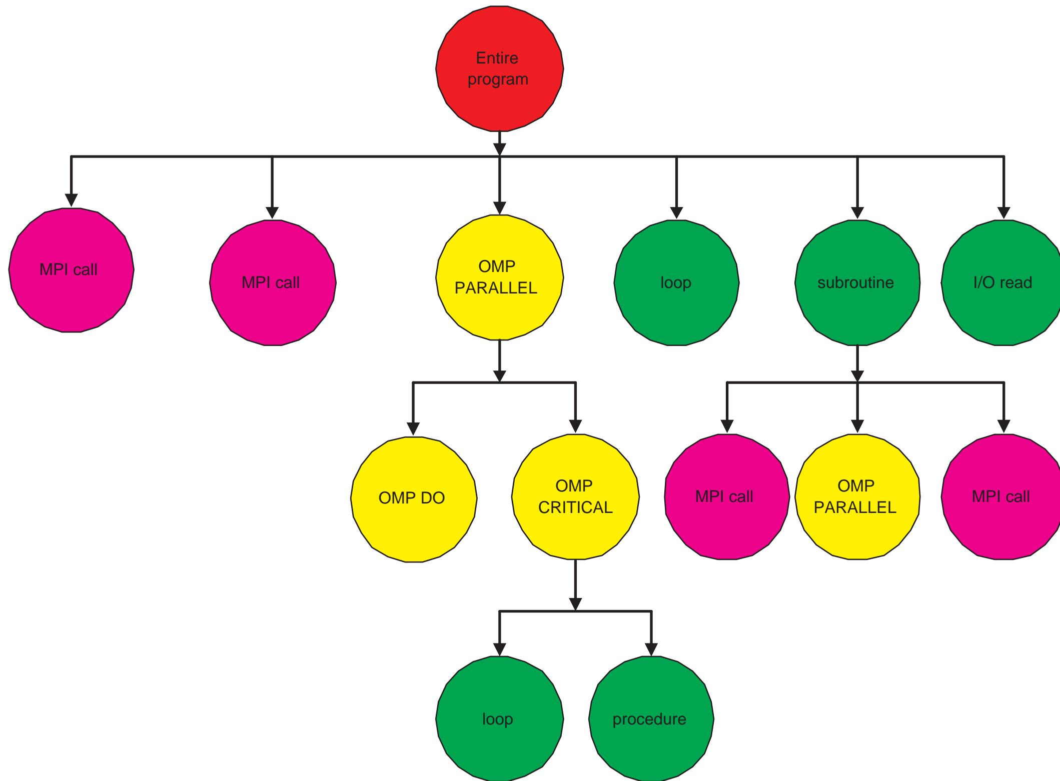


Figure 6: An example of code regions of an OpenMP/MPI program.

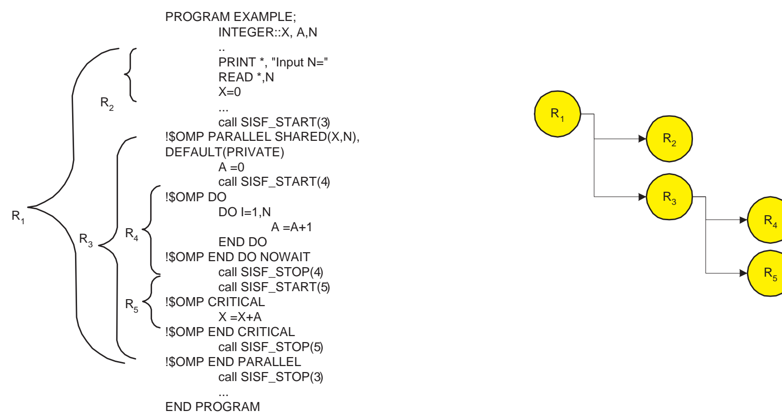


Figure 7: Example of dynamic code region call graph.

1. In initialization step, an abstract code region is set as the root of sub DRG of the thread.
2. IF a code region starts its execution then
   (a) Determine the identifier of its parent code region and then generate an entry that represents the parent-children relationship between this code region and its parent code region and records execution information of the code region. This entry is maintained by instrumentation library.
   (b) Update the parent code region: for all instrumented code regions that will be called directly inside this code region, their parent core region is the current code region.
3. IF a code region finishes its execution then
   (a) Record execution information of this code region and update performance metrics of the code region
   (b) Update the parent code region: the next instrumented code region that will be executed has the same parent as the current code region

Figure 8: The algorithm for generating a sub DRG for a thread.

$$T(r) = T(Start_r) + T(r_1) + ... + T(r_n) + T(End_r) + T(Remain)$$

where $T(r_i)$ is the timing overhead for a code region r_i $(1 \leq i \leq n)$. $T(Start_r)$ and $T(End_r)$ corresponds to the overhead at the beginning (e.g. fork threads, redistribute data, etc.) and at the end (join threads, barrier synchronization, process reduction operation, etc.) of r. The factor $T(Remain)$ is the timing overhead of sub-regions that aren't instrumented and/or executed.

Techniques based on call graph have been widely used in performance analysis of a program. Performance analysis tools such as Vampir[25], gprof[17][16] , CXperf [20] supply the call graph that shows how much time was spent in each function and its children and who calls who. In [10] a call graph is used to improve search strategy for automated performance diagnosis. However, nodes of call graphs in these tools usually represent for routine calls [16] [20]. In our dynamic code region call graph, a node presents for a code region that can be a routine call, a loop, or a arbitrary code region. Therefore, our DRG supplies more fine grain in analysis than the usual function call graph. In addition, the DRG gives overheads related to nodes that could be received with usual call graph in other tools.

4.4.2 Generating and building dynamic code region call graph

Calling code region r_2 inside r_1 during the execution of the program establishes a parent-children relationship between r_1 and r_2. Instrumentation library will generate such relationships, maintain them during the execution of the program. If r_2 is called inside r_1, an entry data that represents the relationship between code region r_1 and r_2 is generated and saved in profile/trace files. For an instrumented code region r that isn't child of any other instrumented code regions, an abstract code region is assigned to its parent. Code regions are distinguished by their identifiers. An identifier of a given code region is determined by the value of identifier of the probe that is used to measure this code region.

The DRG data structure maintains the information of code regions. The algorithm for generating a DRG of a thread is shown in figure 8.

Performance data generated are maintained thread based. In contrast to generating DRG we need to build a DRG for each thread by processing the profile/trace files that contains performance data of respective thread. The algorithm for building DRG of a thread from profile files is shown in figure 9. An application will have many processes with multiple threads for a given process. Each thread has its own DRG.

1. Create the root node of the DRG for the program
2. FOR each process DO
   (a) We start to build sub DRG for current process by creating root node of sub DRG for current process
   (b) FOR each thread of the current process DO
      i. Start to build sub DRG for current thread by creating root node of sub DRG of current thread
      ii. Assume that $currentNode$ is the root node of sub DRG of current thread
      iii. WHILE still exists an entry in trace/profile file of current thread that represents a parent-children relationship (r_1, r_2) where r_1 calls r_2 and determined by $currentNode$.
         A. Build a node $childNode$ represents for information of code region determined by r_2, an edge $e(currentNode, childNode)$ represents for the parent-children relationship of the two code regions.
         B. Add the node $childNode$ and the edge e into the sub DRG of current thread.
         C. Set $currentNode = childNode$
      iv. Add the sub DRG of current thread into the sub DRG of the process
   (c) Add the sub DRG of current process into the DRG of the program.

Figure 9: The algorithm for building DRG for program.

4.4.3 Determining overhead based on the DRG.

In this section we present methods that are used to compute overheads related to the code region. Overhead analysis is based on the equation $T_o = T_p - T_s/p = T_i + T_u$ and our goal is to compute overheads of code regions of a program automatically and determine as much as possible sources of overheads with fine grain way. We model the execution of instrumented sequential and parallel version in DRGs and compare code regions in both versions. For a given program P, we model the equivalent between two DRGs of the sequential and the parallel version.

> For a given program P, DRGs of sequential and parallel version are $DRG_s(V_s, E_s, s_s)$ and $DRG_p(V_p, E_p, s_p)$, respectively.

In some cases, we need to build two DRGs in order to compute the overhead. Figure 10 shows an examples of two DRGs of a program with OpenMP and sequential version.

For a code region R_i^p in DRG_p, we determine the equivalent code region of R_i^p in DRG_s. Assume that this code region is R_i^s. Let call $T_s(R_i^s)$, $T_p(R_i^p)$ and $T_o(R_i)$ are the execution time in sequential version, the execution time in parallel version and the observed overhead. We have:

$$T_o(R_i) = T_p(R_i^p) - T_s(R_i^s)/p$$

Notice that R_i^s can be *empty* in some cases. For examples, R_i^p is a MPI call, the equivalent R_i^s is *empty*. This means that, for a code region R_i^p, we don't always need to know the performance data of R_i^s in sequential version in order to compute the overhead of R_i^p in parallel version. We also notice that the value of $T_o(R_i)$ is determined based on the DRG of sequential version and parallel version. It is the total overhead of code region R_i. Notice that if we assume:

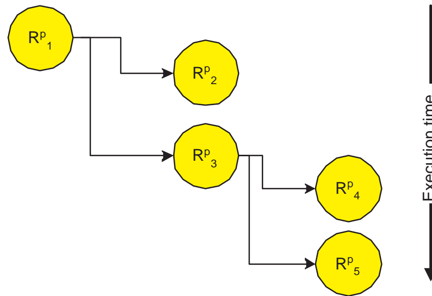
$$T_p = T_p(seq) + T_p(par)$$

where $T_p(seq)$ and $T_p(par)$ are execution time of sequential and parallel part of parallel version. Then $(p-1)T_p(seq)/p$ is the overhead of unparallelised codes (because if this code is executed in parallel, each processor will

```
PROGRAM OMPEXAMPLE;
          INTEGER::X, A,N

          PRINT *, "Input N="
          READ *,N
          X=0

!$OMP PARALLEL SHARED(X,N),
DEFAULT(PRIVATE)
          A =0

!$OMP DO
          DO I=1,N
                    A =A+1
          END DO
!$OMP END DO NOWAIT

!$OMP CRITICAL
          X =X+A
!$OMP END CRITICAL

!$OMP END PARALLEL

END PROGRAM
```

```
PROGRAM SEQEXAMPLE;
          INTEGER::X, A,N

          PRINT *, "Input N="
          READ *,N
          X=0

!!!$OMP PARALLEL SHARED(X,N),
DEFAULT(PRIVATE)
          A =0

!!!$OMP DO
          DO I=1,N
                    A =A+1
          END DO
!!!$OMP END DO NOWAIT

!!!$OMP CRITICAL
          X =X+A
!!!$OMP END CRITICAL

!!!$OMP END PARALLEL

END PROGRAM
```

OpenMP version and its dynamic code region call graph



Sequential version and its dynamic code region call graph

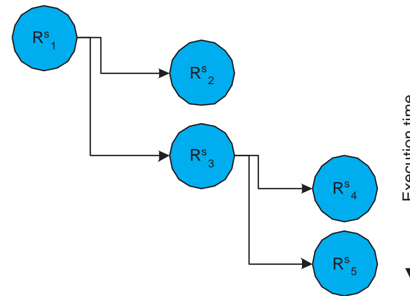


Figure 10: An examples of two code region call graphs of a program.

1. Start from root node s of DRG
2. Call routine **overhead_computing**(s) to compute overheads of node s. This routine acts as follows
   (a) FOR all edges $e(s, currentNode)$ in DRG
      i. Computing total overhead of $currentNode$ based on its type of code region and its equivalent code region in sequential version by applying the equation $T_o = T_p - T_s/p$
      ii. Computing each kind of overheads of $currentNode$ and updating these kinds to its parent
         A. IF $currentNode$ is a leaf node than computing each kind of overheads of the code region represented by $currentNode$ based on its type of code regions, measured data, training sets data if possible. In this step, we determine each kind of overhead associated with the code region. In addition we compute identified overhead T_i and unidentified overhead T_u.
         B. ELSE call routine **overhead_computing**($currentNode$) to compute overheads of $currentNode$.
      iii. Updating overheads of s based on overheads of $currentNode$ and characteristics of code region represented by $currentNode$.
   (b) Updating overhead of s based on its type of code regions, measured data, training sets data if possible.

Figure 11: A computing overheads algorithm.

take $T_p(seq)/p$ in theoretically). It is clearly that if $T_p(seq)$ is fixed, it means that the unparallelised code doesn't change, the overhead of unparallelised code will increase when the number of processors is increased[1].

The question that can be generated in this case is how many percents of total overhead is control of parallelism overhead? Or how many percents of the total overhead is synchronisation overhead ? To solve this problem, we need to determine each kind of overheads of individual leaf node in the dynamic code region call graph. An then, each kind of overhead of intermediate node in the dynamic code region call graph is computed based on its type of code regions and each kind of overhead of its children. Figure 11 shows the algorithm that is used to compute overhead based on DRG. To clarify this model, we turn back to the above example of OMP code region call graph in figure 10. Based on two DRGs, we can compute $T_o(R_3)$, $T_o(R_4)$ and $T_o(R_5)$. How to answer above questions? Assume that we have the control of parallelism overhead $T_{controlofpar}(R_4)$ of OMP DO and the loss of parallelism overhead $T_{lossofpar}(R_5)$ of OMP CRITICAL, then we can compute the control of parallelism overhead $T_{controllofpar}(R_3)$ of OMP PARALLEL as follows:

$$T_{controlofpar}(R_3) = T_{controlofpar}(R_4) + T_{controlofpar}(OMPPARALLEL) + T_{controlofpar}(OMPENDPARALLEL)$$

where $T_{controlofpar}(OMPPARALLEL)$ and $T_{controlofpar}(OMPENDPARALLEL)$ are the control of parallelism overhead generated by OMP PARALLEL and OMP END PARALLEL directive, respectively. This method is different from the method used to compute the total overhead. It based only the DRG of parallel version, raw performance data of parallel version, type of code regions, and training sets data in some cases.

Clearly, number of overheads and portion of identified overhead within total overhead determined depend on the instrumentation task. To apply two methods: the first is used to compute the total overhead T_o, the second is used to compute each kind of overhead of each code region, we can derive the identified overhead T_i and the unidentified part T_u.

[1]It can be proved by induction. We compare the unparallelised overhead when using p and p+1 processors. Let call $Tp_{overhead}$ and $Tp'_{overhead}$ are the overhead of unparallelised code in p and p' processors respectively. With $p' = p + 1$, We have:
$Tp_{overhead} = (p-1)T_p(seq)/p$ and $Tp'_{overhead} = (p'-1)T_p(seq)/p' = pT_p(seq)/(p+1)$.
$Tp_{overhead} < Tp'_{overhead} \Leftrightarrow (p-1)T_p(seq)/p < pT_p(seq)/(p+1) \Leftrightarrow (p^2-1) < p^2$

```
cybill:/home/truong
Window  Edit  Options                                                    Help
papisis/bin> ./sisprofile.pl -p omp -df /home/linh/projects/papisis/mds.sisdesc

****************************************************************
TIMING PERFORMANCE DATA

NODE :0,THREAD: 0
----------------------------------------------------------------
 Name                  Desc.     Calls      Subs Wallclock  Usertime  Sys.time  pf. pfio  swaps

    1                     MD         1      2001 286577385 281930000   4640000   19   12      0
    3    OMP PARALLEL COMPUTE      1001      3003 282360771 279750000   2640000    2    1      0
    6          OMP DO COMPUTE      1001    500500 282336303 279750000   2620000    2    1      0
   14     OMP PARALLEL UPDATE      1000      3000   4179276   2150000   2000000    0    0      0
   17           OMP DO UPDATE      1000    500000   4153902   2140000   1990000    0    0      0


****************************************************************
HARDWARE PERFORMANCE DATA

NODE :0,THREAD: 0
----------------------------------------------------------------
 Name                  Desc.     Calls      Subs    PAPI_L2_TCA    PAPI_L2_TCM

    1                     MD         1      2001       62615215           4810
    3    OMP PARALLEL COMPUTE      1001      3003       57872673           2102
    6          OMP DO COMPUTE      1001    500500       57806150           2076
   14     OMP PARALLEL UPDATE      1000      3000        4000514           2192
   17           OMP DO UPDATE      1000    500000        3924096           2189
papisis/bin>
```

Figure 12: A prototype of *sisprofile* utility.

```
cybill:/home/truong
Window  Edit  Options                                                    Help
papisis/bin> ./sisoverhead.pl -p omp -df /home/linh/projects/papisis/mds.sisdesc -seq ../../testsistool/mdsseq.sisperf

1. General performance information

Total exec.           = 146754592.000000(usec)
Seq.part(without MPI) = 50219(usec)
OpenMP.part           = 146704373(usec)
MPI part              = 0(usec)
Rel. Speedup          = 1.95276605041429

. Worksharing Region & Load imbalancing

   Regions        Avg.      Loadim        Rel.           1           2
         6   1.425e+08   7.488e+05   5.227e-03   1.417e+08   1.432e+08
        17   3.373e+06   2.983e+04   8.766e-03   3.344e+06   3.403e+06

OVERHEAD OF CODE REGIONS

   regions               desc.  loss.par fork/join  schedule      wait   datamov      add.        Ti        Tu   Tp-Ts/p

         1                  MD 2.511e+04 1.728e+04 9.962e+05 1.572e+06 0.000e+00 0.000e+00 2.610e+06 8.558e+05 3.466e+06
         3OMP PARALLEL COMPUTE 0.000e+00 9.589e+03 4.536e+05 1.507e+06 0.000e+00 0.000e+00 1.970e+06 1.244e+05 2.094e+06
         6      OMP DO COMPUTE 0.000e+00 0.000e+00 4.536e+05 0.000e+00 0.000e+00 0.000e+00 4.536e+05 1.244e+05 5.779e+05
        14 OMP PARALLEL UPDATE 0.000e+00 7.692e+03 5.426e+05 6.475e+04 0.000e+00 0.000e+00 6.150e+05 7.250e+05 1.340e+06
        17       OMP DO UPDATE 0.000e+00 0.000e+00 5.426e+05 0.000e+00 0.000e+00 0.000e+00 5.426e+05 7.240e+05 1.267e+06
papisis/bin>
```

Figure 13: A prototype of *sisoverhead* utility.

5 Post-execution performance analysis

Besides the existed tools such as pprof, racy [32][29] developed in University of Oregon and VAMPIR [25][27] developed at Pallas, in the prototype of SCALEA we supply two tools: *sisprofile* and *sisoverhead* [31] based on methods discussed in this paper for post-execution performance analysis.

sisprofile reads raw performance data and provides it to the user. As you can see in figure 12, execution time(wall clock time, user time, system time), system information(number of page faults, number of swaps) and hardware parameter counter of each code region are provided. In addition, We report a summary execution time of MPI code regions that consists of point-to-point , collective communication, MPI IO,etc.

The dynamic code region call graph is used to automatically compute overhead associated with a given code region. This function is implemented in *sisoverhead* utility. Figure 13 shows three main output of sisoverhead utility. The first part presents the overall of the entire program such as total execution time, total execution time spent in sequential, OpenMP, and MPI code regions. Furthermore, it shows the speedup/improvement of code regions. The second part shows the load imbalance of a code region among processes or threads. This part provides load balance information of a code region such as average execution time, load imbalance, relative load imbalance, time spent each threads, processes of the code region. The last part presents the overhead related to individual code region such as loss of parallelism, fork/join, data movement, etc.

We also compare the improvement or speedup of a code region in two versions or two different invocations. In our implementation, *sisoverhead* will take the performance data of previous version or invocation and compare with the performance data of current version or invocation.

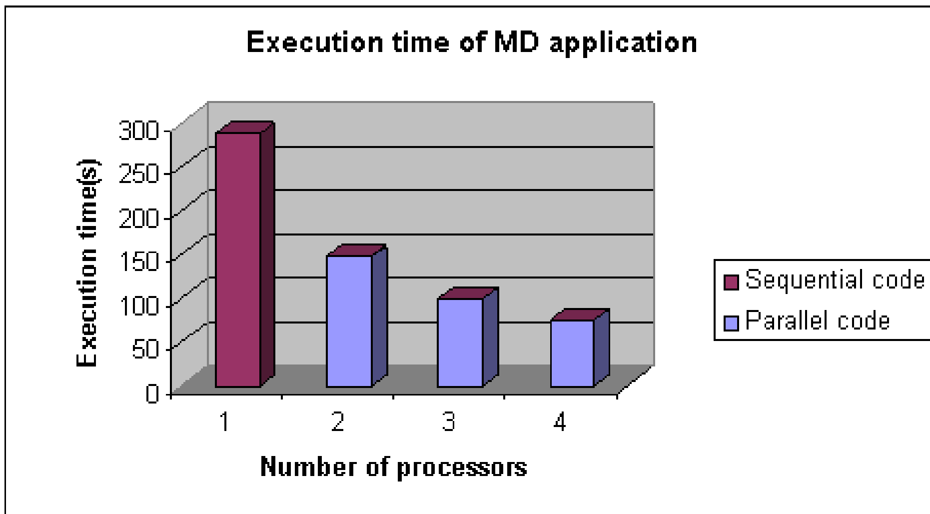


Figure 14: Execution time of MD application.

The instrumentation can be applied to various code regions. According to the instrumented program, *sisoverhead* can automatically analyze the overhead. Notice that the number of overhead that can be analyzed depends on the control of instrumentation and the input program.

6 Experiments

We present some experiments to demonstrate the usefulness of *SCALEA*. Our experiments are performed on computational nodes of the 4 processors Pentium III subcluster of *gescher*[30] system at Vienna Center of Parallel Computing. Each computational node that has 4 Intel Pentium III Xeon 700MHz CPUs with 1MB full-speed L2 cache, 2Gbyte ECC RAM, Intel Pro/100+Fast Ethernet, Ultra160 36GB harddisk is run with Linux 2.2.18-SMP patched with perfctr for hardware counters performance. We use MPICH[18], and PGI compiler version 3.3.

6.1 Molecular dynamic(MD) application

The MD program implements a simple molecular dynamics simulation in continuous real space. This program is implemented parallelism via OpenMP directives and written by Bill Magro in Kuck and Associates, Inc. (KAI).

The original program exploits parallelism through two PARALLEL code regions. We obtain this program from[26] and instrumented all OpenMP code regions and entire program.

The performance of MD application has been measured on one computational node of *gescher*. Figure 14 and table 2 show timing and measured overhead of MD application, respectively. The results show this program has good speedup(nearly linear). As we can see in table 2, the total overhead is small and almost temporal overhead has been determined.

In table 2, because the timing of unparallelised code is unchangeable so that loss of parallelism overhead increases when number of processors is increased. The synchronisation overhead reduces because we have better load balance between threads when increasing number of threads.

Overhead	2CPUs	3CPUs	4CPUs
Loss of parallelism	0.025	0.059	0.066
Control of parallelism	1.013	0.676	0.517
Synchronisation	1.572	1.27	0.942
T_i	2.61	2.009	1.527
T_u	0.855	0.903	0.908
T_o	3.466	2.913	2.435
Total execution time	146.754	98.438	74.079

Table 2: Overhead(sec) of MD application. T_i, T_u, T_o are identified, unidentified and total overhead, respectively.

We then examine the ratio L2 cache misses/cache accesses of OMP DO code regions. There are two interesting OMP DO code regions; namely *OMP DO COMPUTE* and *OMP DO UPDATE*. In figure 15, the ratio cache misses/cache accesses in sequential equals nearly 0, it means that the data fits in L2 cache. However, in parallel version, this ratio is increased. We observe that the master thread always has larger cache misses/cache accesses than other threads. The reason for this figure is that all threads compute data on global arrays and each processor has a private L2 cache so that number of cache misses is increased due to maintaining caches between processors while number of cache accesses in each processors is reduced.

6.2 Backward pricing kernel

The backward pricing code [12] implements the backward induction algorithm to compute the price of an interest rate dependent financial product, such as a variable coupon bond. Two parallel code versions have been created. First, an HPF+ version that exploits only data parallelism, and second, a mixed version that combines HPF+ with OpenMP. For this version, VFC generates an OpenMP/MPI program. HPF+ directives are used to distribute data onto a set of SMP nodes. Communication among between nodes is realized by MPI calls. Within each node an OpenMP program is executed.

The execution times for both versions are shown in Figure 16. The term "all" in the legend denotes the entire program, whereas "loop" refers to the main computation loops (HPF INDEPENDENT loop and an OpenMP parallel loop for version 1 and 2, respectively). The HPF+ version performs worse than the OpenMP/MPI version which shows almost linear speedup for up to 2 nodes. Tables 3 and 5 displays the overheads for the HPF+ and mixed OpenMP/MPI version, respectively. In both cases the largest overhead is caused by the control of parallelism overhead which rises significantly for the HPF+ version with increasing number of nodes. This effect is less severe for the OpenMP/MPI version. In order to find the cause for the high control of parallelism overhead we use SCALEA to determine the individual components of this overhead (see Tables 4 and 6). We can see that two routines (Update_HALO and MPI_Init) are mainly responsible for the high control of parallelism overhead of the HPF+ versions. Update_HALO updates the overlap areas of distributed arrays which causes communication if one process requires data that is owned by another process in a different node. MPI_Init initializes the MPI runtime system which also involves communication. The reasons why these two routines imply a much higher overhead for the HPF+ version compared to the mixed OpenMP/MPI version is as follows. The first version generates a separate process on every CPU of each SMP node, whereas the mixed OpenMP/MPI version only uses one process per node.

6.3 LAPW0

The program LAPW0 [6] calculates the effective potential of the Kohn-Sham eigen-value problem. LAPW0 has been implemented as a Fortran MPI code. We used SCALEA to localize the most important code regions of LAPW0 which can be further subdivided into

- sequentialized code regions: *FFT_REAN0, FFT_REAN3, FFT_REAN4*

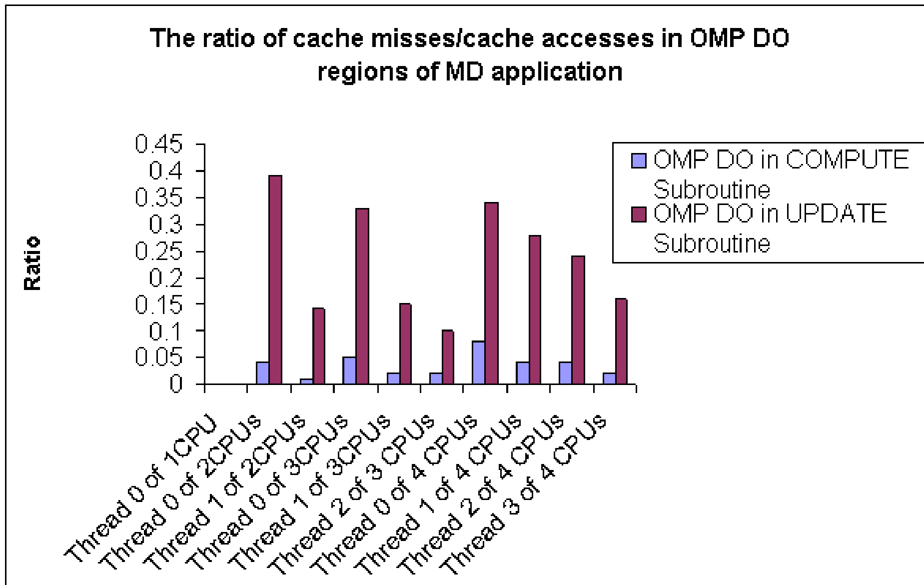


Figure 15: The ratio L2 cache misses/cache accesses of OMP DO regions of MD application.

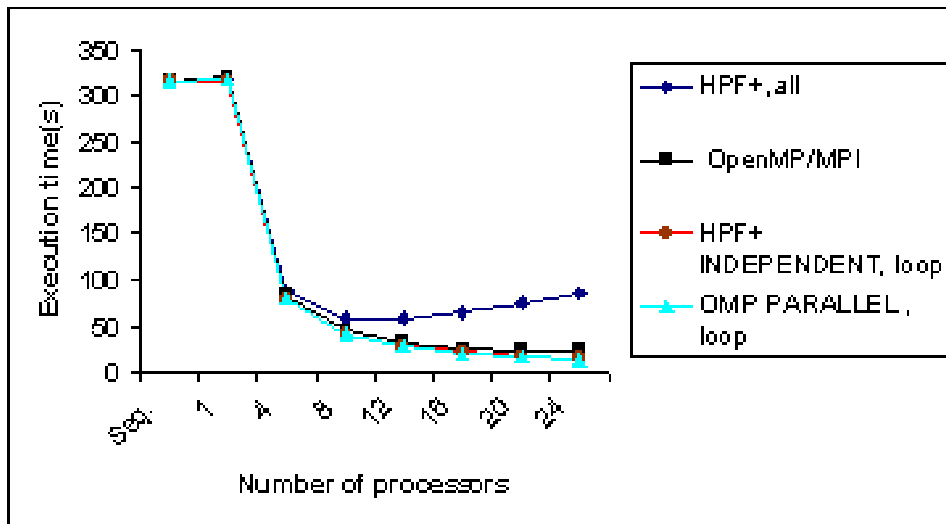


Figure 16: Execution times of HPF+ and OpenMP/MPI version for the backward pricing kernel.

Processors	Seq.	1N, 1P	1N, 4P	2N, 4P	3N, 4P	4N,4P	5N,4P	6N,4P
Data movement	0	0	0.012	0.03	0.0207	0.0233	0.03028	0.0353
Control of parallelism	0	0.244258	6.59928	17.2419	28.9781	41.4966	56.4554	70.7302
T_i		0.244258	6.611285	17.2719	28.9988	41.5199	56.48576	70.76559
T_u		3.139742	1.726465	1.835957	2.047059	2.3549	2.99739	2.5173
T_o		3.384	8.33775	19.10787	31.0459	43.8749	59.48315	73.2829
Total execution time	316.417	319.801	87.442	58.66	57.414	63.651	75.304	86.467

Table 3: Overheads of the HPF+ version for the backward pricing kernel. T_i, T_u, T_o are identified, unidentified and total overhead, respectively. *1N, 4P* means 1 SMP node with 4 processors.

Processors	1N, 1P	1N, 4P	2N, 4P	3N, 4P	4N,4P	5N,4P	6N,4P
Inspector	0.089	0.022	0.0116	0.00798	0.00643	0.00489	0.00415
Work distribution	0.000258	0.000285	0.000318	0.000161	0.000204	0.01059	0.000142
Update HALO	0.149	3.114	9.110	16.170	24.060	33.868	43.830
MPI_Init	0.005	3.462	8.113	12.784	17.420	22.568	26.860
Other	0	0.001	0.007	0.016	0.010	0.004	0.036

Table 4: Control of parallelism overheads for the HPF+ version of the backward pricing kernel.

Processors	1N, 1P	1N,4P	2N, 4P	3N, 4P	4N, 4P	5N,4P	6N,4P
Data movement	0	0	0.01352	0.01432	0.01618	0.01753	0.01938
Control of parallelism	0.1914	0.2457	1.8481	3.5198	4.8865	6.5339	7.9698
Loss of parallelism	0	2.413	1.535	1.215	1.055	0.9432	0.9175
Synchronization	0.00401	0.7355	0.2878	0.224	0.1379	0.217	0.1025
T_i	0.19541	3.39422	3.68448	4.97318	6.09565	7.71163	9.0091
T_u	3.87599	1.30753	0.87239	0.65773	0.57228	0.48251	0.51576
To	4.071	3.98175	4.556875	5.63091	6.66793	8.19415	9.525958
Total execution time	320.488	83.086	44.109	31.999	26.444	24.015	22.71

Table 5: Overheads of the OpenMP/MPI version for the backward pricing kernel.

Processors	1N, 1P	1N,4P	2N, 4P	3N, 4P	4N, 4P	5N,4P	6N,4P
Inspector	0.00895	0.00891	0.051	0.0303	0.0228	0.018	0.0157
Work distribution	0.00614	0.000245	0.00599	0.00598	0.00594	0.000122	0.00593
Update HALO	0.142	0.140	0.594	1.122	1.349	1.783	2.136
MPI_Init	0.02874	0.005207	1.185	2.35	3.497	4.721	5.799
Fork/join	0.00559	0.0107	0.0116	0.0113	0.0104	0.0115	0.0129
Other	0.000005	0.000008	0.000226	0.000217	0.000234	0.000253	0.000249

Table 6: Control of parallelism overheads of the OpenMP/MPI version for backward pricing kernel.

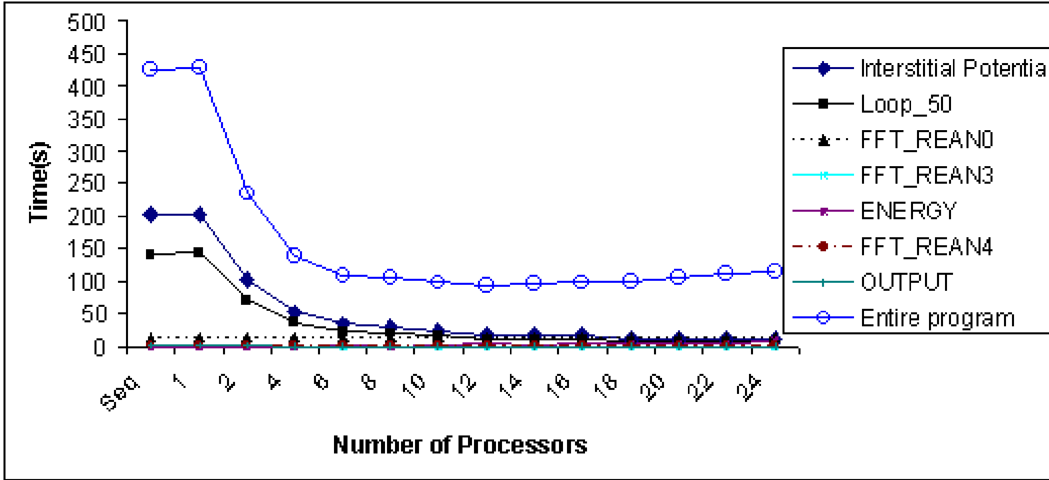


Figure 17: Execution times for code regions of LAPW0.

Processors	**Seq**	**1**	**4**	**8**	**12**	**16**	**20**	**24**
Loss of parallelism		0	12.047	14.222	14.83258	15.356	15.8289	15.9179
Data movement		0.000015	1.267	1.361	1.594	2.677	3.298	7.993
Control of parallelism		0.00248	6.58	12.99	18.15	21.39	26.62	31.56
T_i		0.002495	19.894	28.573	34.576	39.423	45.7469	55.4709
T_u		2.601505	13.004	23.043	23.388	31.328	41.811	41.633
T_o		2.604	32.898	51.617	57.965	70.751	87.558	97.104
Total execution time	425.207	427.811	139.2	104.768	93.399	97.327	108.819	114.821

Table 7: Overheads of LAWP0. T_i, T_u, T_o are identified, unidentified and total overhead, respectively.

- parallelized code regions: *Interstitial Potential*, *Loop_50*, *ENERGY*, *OUTPUT*

The execution time behavior and speedups (based on the sequential execution time of each code region) for each of these code regions are shown in Figures 17 and 18, respectively. LAPW0 has been examined for a problem size of 36 atoms which are distributed onto the set of processing units. Clearly when using 8, 16, and 24 processing units we can't reach optimal load balance, whereas 1, 2, 4, 6, 12 and 18 processing units display a much better load imbalance. This effect is confirmed by SCALEA (see Figure 18) for the the most computationally intensive routines of LAPW0 (*Interstitial Potential* and *Loop_50*).

Overall, LAPW0 does not scale at all which is not only due to the load imbalance effects but also implied by large overheads, in particular, loss of parallelism, data movement, and control of parallelism, as tabulated in Table 7. The main sources of control of parallelism overhead is detected by SCALEA to be caused by MPI_Init (see Figure 19). SCALEA also discovered that the main reason for the loss of parallelism overhead is found in the LAPW0 code regions *FFT_REAN0*, *FFT_REAN3*, and *FFTP_REAN4* all of which are sequentialized.

LAPW0 uses many BLAS and SCALAPACK library calls that are currently not instrumented by SCALEA which is the reason for the large fraction of unidentified overhead (see Figure 7).

7 Related works

OVALTINE[2] is a tool that automates the determination of the overheads of a given OpenMP implementation with respect to a give sequential implementation. Currently OVALTINE supports only Fortran 77 source files. OVALTINE uses the classification described in[9] for the overhead analysis.

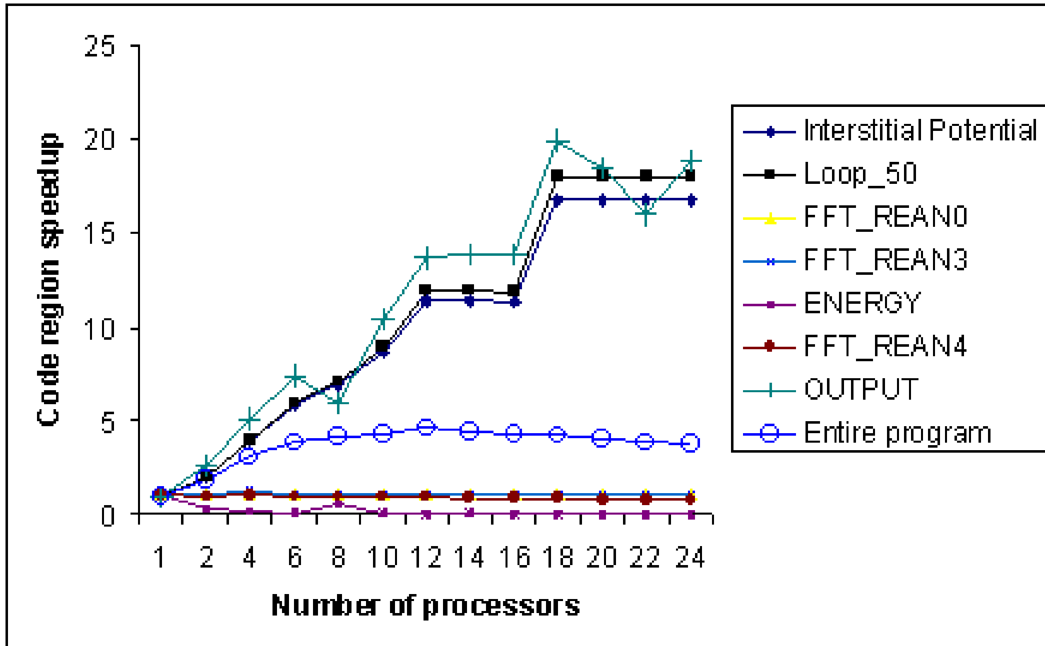


Figure 18: Speedup values for code regions of LAPW0.

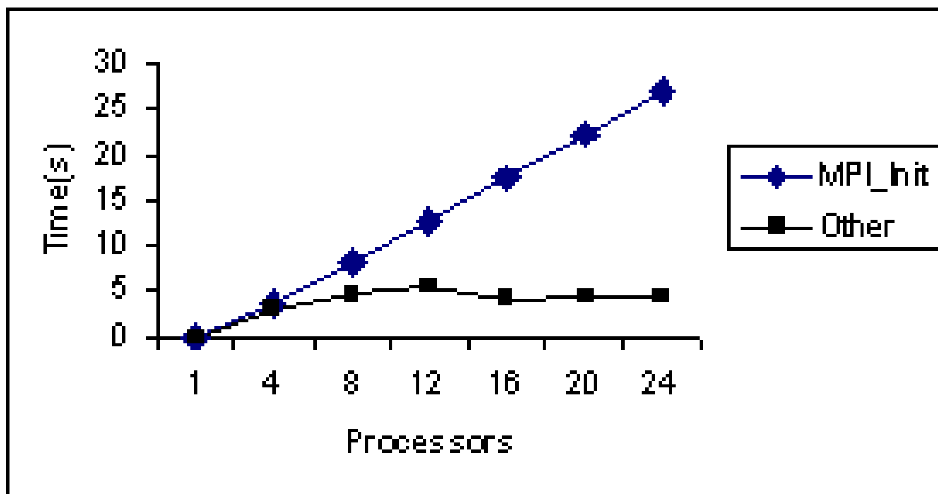


Figure 19: Sources of control of parallelism overhead for LAPW0.

The OMPtrace[15], MPItrace[13], OMPItrace[14] tool instrument parallel codes that use the OpenMP, MPI, combined OpenMP and MPI programming model. These tools generate a Paraver[33] trace file where the basic activity in an program is recorded. They support for both timing and hardware counters. OMPtrace, MPItrace, OMPI trace rely on a dynamic interception mechanism that enable them to intercept calls to the runtime library. The instrumentation code is inserted into binary file. The trace files generated by these tools can be visualized with Paraver[33].

PARADYN[23] is a well-know tool for automatic performance analysis a large-scale parallel program developed in Paradyn project[28]. PARADYN applies a dynamic instrumentation at the runtime of the application and searches for the bottle neck based on predefine constrains. PARADYN uses a call graph to improve search strategy for automated performance diagnosis[10].

TAU[29] performance framework is an integrated toolkit for performance instrumentation, measurement, and analysis for parallel, multithreaded programs. TAU attempts to target the general complex system computation model based on node, context, thread model with different instrumentation strategies: source-code instrumentation, dynamic instrumentation, instrumented library. TAU[29] supplies a sophisticated instrumentation library that supports for profiling and tracing MPI, OpenMP, and MPI mixing OpenMP programs written in C and Fortran and Java. Currently TAU restricts performance experiments to measure a single performance metric: execution time or hardware counter. *Scalea* uses TAU instrumentation library as one of its instrumentation library.

VAMPIRtrace[27] is used to trace MPI applications. VAMPIRtrace is a commercial product produced and distributed by PALLAS. In addition to recording all calls to the MPI library and all transmitted messages, VAMPIRtrace allows to define and record arbitrary user-defined events. Tracing can be switched on or off during runtime, and a powerful filtering mechanism helps to limit the amount of the generated trace data.

PAPI [8] specifies a standard API for accessing hardware performance counters available on most modern microprocessors. *Scalea* has used PAPI library for measuring hardware counters performance.

8 Conclusion

In this paper, we described *SCALEA* which is a performance analysis system for distributed and parallel programs. SCALEA currently supports performance analysis for OpenMP, MPI, HPF and mixed parallel programs (e.g. OpenMP/MPI) SCALEA instruments, executes and measures parallel/distributed programs and computes a variety of performance overheads based on a novel overhead classification. Moreover, SCALEA uses a new representation of code regions, called dynamic code region call graph, which enables a highly accurate overhead analysis for generic code regions (ranging from the entire program to single statements). SCALEA supports profiling and tracing based on source code measurement libraries and HW-profiling interfaces. Based on a prototype implementation of SCALEA we presented several experiments for realistic codes implemented in MPI, HPF, and mixed OpenMP/MPI. These experiments demonstrated the usefulness of SCALEA to find performance problems and their causes. We are currently integrating SCALEA with a database to store all derived performance data. Moreover, we plan to enhance SCALEA with an automatic experiment system and a performance specification language in order to support automatic performance analysis.

Acknowledgments

The authors would like to thank Clovis Segagiotto Jr., Radu Prodan for their valuable discussions. Sabri Pllana has supplied an UML model of LAPW0 application. Hans Moritsch in Department of Statistics and Decision Support Systems of University of Vienna, Dr. Dieter Kavasnisca and Dr. Georg Hellerup Madsen in Theoretical Chemistry group at Institute for Physical and Theoretical Chemistry of Vienna University of Technology have supplied and introduced their codes for testing.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1967.

[2] M.K. Bane and G.D. Riley. Automatic overheads profilers for openmp codes. In *Second European Workshop on OpenMP proceedings(EWOMP 2000)*, Edinburgh, Scotland, September 2000.

[3] S. Benkner. Vfc: The vienna fortran compiler. *Journal of Scientific Programming*, pages 67–81, December 1998.

[4] S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.

[5] Siegfried Benkner. HPF+: High Performance Fortran for advanced scientific and engineering applications. *Future Generation Computer Systems*, 15(3):381–391, ???? 1999.

[6] P. Blaha, K. Schwarz, and J. Luitz. WIEN97, Full-potential, linearized augmented plane wave package for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, ISBN 3-9501031-0-4, 1999.

[7] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface 2.0*. OpenMP Organisation, December 2000. Also avaible at http://www.openmp.org.

[8] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceeding SC'2000*, November 2000.

[9] J.M. Bull. A hierarchical classification of overheads in parallel programs. In P. Croll I. Jelly, I. Gorton, editor, *Proceedings of Firs IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219. Chapman Hall, March 1996.

[10] Harold W. Cain, Barton P. Miller, and Brian J.N. Wylie. A callgraph-based search strategy for automated performance diagnosis. In *Euro-Par 2000 Parallel Processing*, pages 108–122, 2000.

[11] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Memon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[12] E. Dockner and H. Moritsch. Pricing Constant Maturity Floaters with Embeeded Options Using Monte Carlo Simulation. Technical Report AuR_99-04, AURORA Technical Reports, University of Vienna, January 1999.

[13] European Center for Parallelism of Barcelona. *MPItrace tool version 1.1, Instrumentation of generic MPI Applications, User's Guide*. Technical University of Catalonia, November 2000. Also available at http://www.cepba.upc.es/paraver/.

[14] European Center for Parallelism of Barcelona. *OMPItrace tool version 1.1, Instrumentation of combined OpenMP and MPI Applications, User's Guide*. Technical University of Catalonia, November 2000. Also available at http://www.cepba.upc.es/paraver/.

[15] European Center for Parallelism of Barcelona. *OMPtrace tool version 1.1, Instrumentation of generic OpenMP Applications, User's Guide*. Technical University of Catalonia, November 2000. Also available at http://www.cepba.upc.es/paraver/.

[16] Jay Fenlason and Richard Stallman. *GNU gprof*. Free Software Foundation, Inc., September 1997.

[17] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. `gprof`: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.

[18] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[19] R. Hempel. The MPI standard for message passing. *Lecture Notes in Computer Science*, 797:247–252, 1994.

[20] Hewlett Packard. *CXperf User's Guide*, June 1998.

[21] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical report, Rice University, Houston,TX, November 1994.

[22] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing:design and analysis of parallel algorithms*. Benjamin/Cummings, 1994.

[23] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[24] B. Mohr, D. Brown, and A. Malony. TAU: A portable parallel program analysis environment for pC++. In *CONPAR*, Linz, Austria, 94.

[25] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[26] OpenMP Website: Samples. http://www.openmp.org/index.cgi?samples.

[27] Pallas GmbH. *Vampirtrace 2.0 Installation and User's Guide*, November 1999.

[28] PARADYN Project. http://www.cs.wisc.edu/paradyn/.

[29] T. Sheehan, A. Malony, and S. Shende. A runtime monitoring framework for tau profiling system. In *Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments(ISCOPE's 99)*, San Franciso, December 1999.

[30] Gescher system. http://gescher.vcpc.univie.ac.at.

[31] Hong-Linh Truong and Thomas Fahringer. SCALEA Version 1.0: User's guide. Technical report, Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria, April 2001.

[32] University of Oregon. *TAU User's guide*.

[33] T. Cortes V. Pillet, J. Labarta and S. Girona. Paraver: A tool to visualize and analyze parallel code. In *WoTUG-18*, pages 17–31, Manchester, April 1995.

[34] OpenMP Website. http://www.openmp.org.