On Using SCALEA for Performance Analysis of Distributed and Parallel Programs [*][†]

Hong-Linh Truong[‡], Thomas Fahringer[‡]
Georg Madsen[§], Allen D. Malony[**], Hans Moritsch[¶], Sameer Shende[**]

[‡]Institute for Software Science, University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
{truong,tf}@par.univie.ac.at

[¶]Department of Bussiness Administration, University of Vienna
Bruenner Strasse 72, A-1210, Vienna, Austria
moritsch@finance2.bwl.univie.ac.at

[§]Institute of Physical and Theoretical Chemistry, Technical University Vienna
Getreidemarkt 9/156,A-1060 Vienna, Austria
gmadsen@theochem.tuwien.ac.at

[**]Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{malony,sameer}@cs.uoregon.edu

Abstract

In this paper we give an overview of SCALEA, which is a new performance analysis tool 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. Source code and HW-profiling is combined in a single system which significantly extends the scope of possible overheads that can be measured and examined, ranging from HW-counters, such as the number of cache misses or floating point operations, to more complex performance metrics, such as control or loss of parallelism. Moreover, SCALEA uses a new representation of code regions, called the dynamic code region call graph, which enables detailed overhead analysis for arbitrary code regions.An instrumentation description file is used to relate performance information to code regions of the input program and to reduce instrumentation overhead. Several experiments with realistic codes that cover MPI, OpenMP, HPF, and mixed OpenMP/MPI codes demonstrate the usefulness of SCALEA.

Keywords: performance analysis, performance overhead classification, distributed and parallel systems

1 Introduction

As hybrid architectures (e.g., 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

[*]This research is supported by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104.

OpenMP [27], MPI [13], and HPF [15], mixed programming paradigms such as OpenMP/MPI are increasingly being evaluated.

In this paper we introduce a new performance analysis system, SCALEA, for distributed and parallel programs that covers all of the above mentioned programming paradigms. SCALEA is based on a novel classification of performance overheads for shared and distributed memory parallel programs which includes data movement, synchronization, control of parallelism, additional computation, loss of parallelism, and unidentified overheads. SCALEA is among the first performance analysis tools that combines source code and HW profiling in a single system, significantly extending the scope of possible overheads that can be measured and examined. These include the use of HW counters for cache analysis to more complex performance metrics such as control or loss of parallelism. 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 enables a detailed overhead analysis for every code region. The DRG is not restricted to function calls but also covers loops, I/O and communication statements, etc. Moreover, it allows arbitrary code regions to be analyzed. These code regions can vary from a single statement to an entire program unit. This is in contrast to existing approaches that frequently use a call graph which considers only function calls.

A prototype of SCALEA has been implemented. We will present several experiments with realistic programs including a molecular dynamics application (OpenMP version), a financial modeling (HPF, and OpenMP/MPI versions) and a material science code (MPI version) that demonstrate the usefulness of SCALEA.

The rest of this paper is structured as follows: Section 2 describes an overview of SCALEA [24]. In Section 3 we present a novel classification of performance overheads based on which SCALEA instruments a code and analyses its performance. The dynamic code region call graph is described in the next section. Experiments are shown in Section 5. Related work is outlined in Section 6. Conclusions and future work are discussed in Section 7.

2 SCALEA Overview

SCALEA is a post-execution performance tool that instruments, measures, and analyses the performance behavior of distributed memory, shared memory, and mixed parallel programs.

Figure 1 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 [3] which is a compiler that translates Fortran programs (Fortran90, MPI, OpenMP, HPF, and mixed programs) into Fortran90/MPI or mixed OpenMP/MPI programs. The input 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 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* (see Section 2.2) that enables all gathered performance data to be related back to the input program and to reduce instrumentation overhead.

SIS [25] targets a performance measurement system based on the TAU performance framework. TAU is an integrated toolkit for performance instrumentation, measurement, and analysis for parallel, multi-threaded programs. The TAU measurement library provides portable profiling and tracing capabilities, and supports access to hardware counters. SIS automatically instruments parallel programs under VFC by using the TAU instrumentation library and builds on the abstract syntax tree of VFC and on the TAU measurement system to create the dynamic code region call graph (see Section 4). The main functionality of SIS is given as follows:

- Automatic instrumentation of pre-defined code regions (loops, procedures, I/O statements, HPF INDEPENDENT loops, OpenMP PARALLEL loops, OpenMP SECTIONS, MPI send/receive, 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.

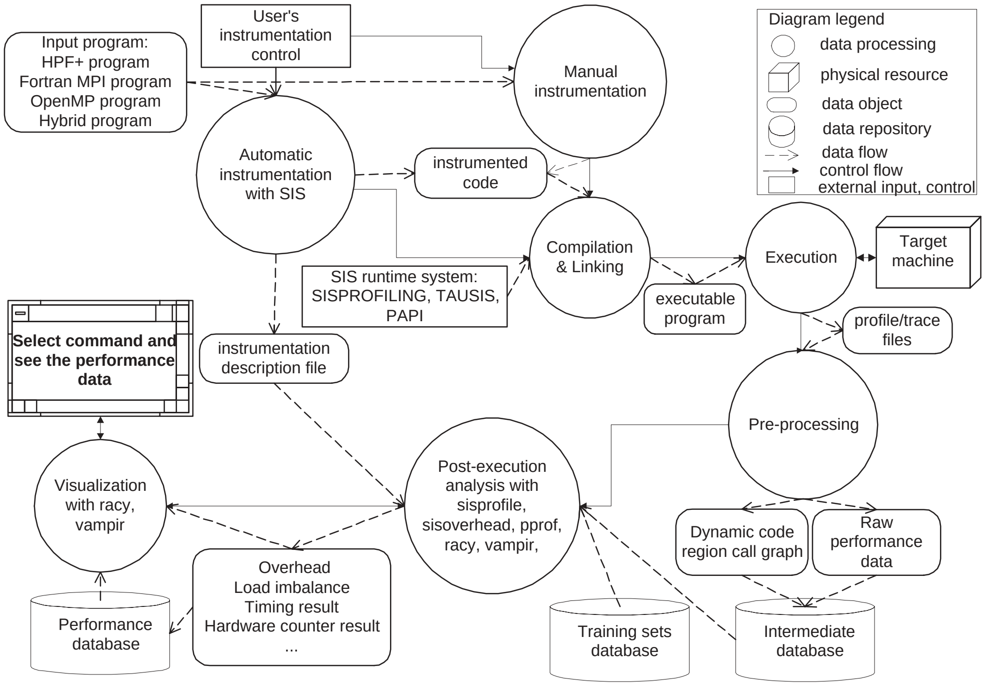


Figure 1: Architecture of SCALEA

- Manual instrumentation to turn on/off profiling for a given code region.

A pre-processing phase of SCALEA filters and extracts all relevant performance information from profiles/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 overhead analysis 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 other code regions. 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. In the following we describe the SCALEA instrumentation system and the instrumentation description file. More details about SIS and SCALEA's post-execution performance analysis can be found in [24, 25].

2.1 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.

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. Note that there can be several entry and exit nodes for a code region. Appropriate directives must be inserted by the

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)

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 5.2), 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.

2.2 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.

3 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. In theory T_o can never be negative, which implies that the speedup T_s/T_p can never exceed p [16]. However, in practice it occurs that temporal overhead can become negative due to super linear speedup of applications. This effect is commonly caused by an increased available cache size. In Figure 2 we give a classification of temporal overheads based on which the performance analysis of SCALEA is conducted:

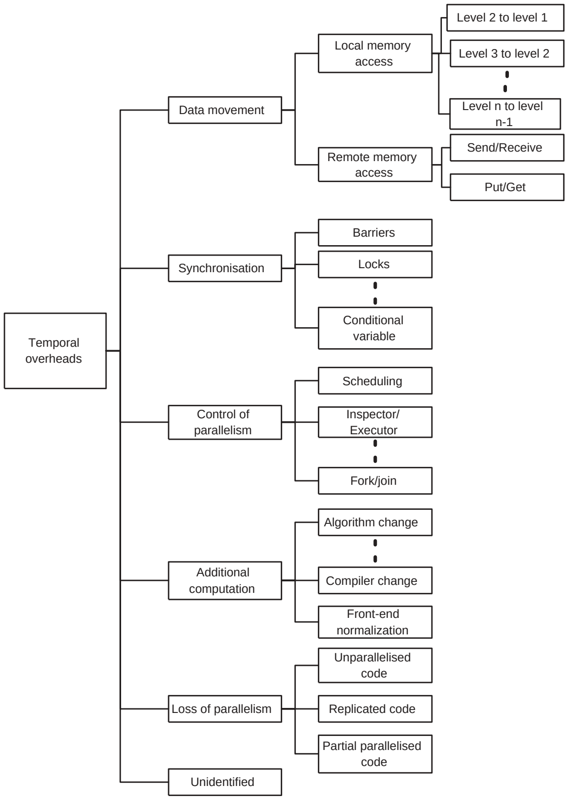


Figure 2: Temporal overheads classification

- *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.

- *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 runtime library, user, and compiler operations.

- *Additional computation* reflects any change of 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 as follows: unparallelized code (executed by only one processor), replicated code (executed by all processors), and partially parallelized code (executed by more than one but not all processors).

- *Unidentified overhead* corresponds to the overhead that is not covered by the above categories.

Note that the above mentioned classification has been stimulated by [6] but differs in several respects. In [6], 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 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 unit. A code region can be, respectively, entered and exited by multiple entry and exit control flow points (see Figure 3). In most cases, however, code regions are single-entry-single-exit code regions.

In order to measure the execution behavior of a code region, the instrumentation system has to detect all *entry* and *exit* nodes of a code region and insert probes at these nodes. Basically, this task can be done with the support of a compiler or guided through manual insertion of directives. Figure 3 shows an example of a code region with its entry and exit nodes. To select an arbitrary code region, the user, respectively, marks two statements as the entry and exit statements – which are at the same time entry and exit nodes – of the code region (e.g., by using SIS directives [25]). Through a compiler analysis,

SIS then automatically tries to determine all other entry and exit nodes of the code region. Each node represents a statement in the program. Figure 3 shows an example code region with multiple entry and exit nodes. The instrumentation tries to detect all these nodes and automatically inserts probes before and after all entry and exit nodes, respectively.

Code regions can be overlapping. SCALEA currently does not support instrumentation of overlapped code regions. The current implementation of SCALEA supports mainly instrumentation of single-entry multiple-exit code regions. We are about to enhance SIS to support also multiple-entry multiple-exit code regions.

4.1 Dynamic Code Region Call Graph

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 (see Section 2.1) 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 runtime 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 DRG is used as a key data structure to conduct a detailed performance overhead analysis under SCALEA. Notice that the timing overhead of a code region r with n explicitly instrumented sub-regions $r_1, ..., r_n$ is given by

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

where $T(r_i)$ is the timing overhead for an explicitly instrumented code region r_i $(1 \leq i \leq n)$. $T(Start_r)$ and $T(End_r)$ correspond 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. $T(Remain)$ corresponds to the code regions that have not been explicitly instrumented. However, we can easily compute $T(Remain)$ as region r is instrumented as well.

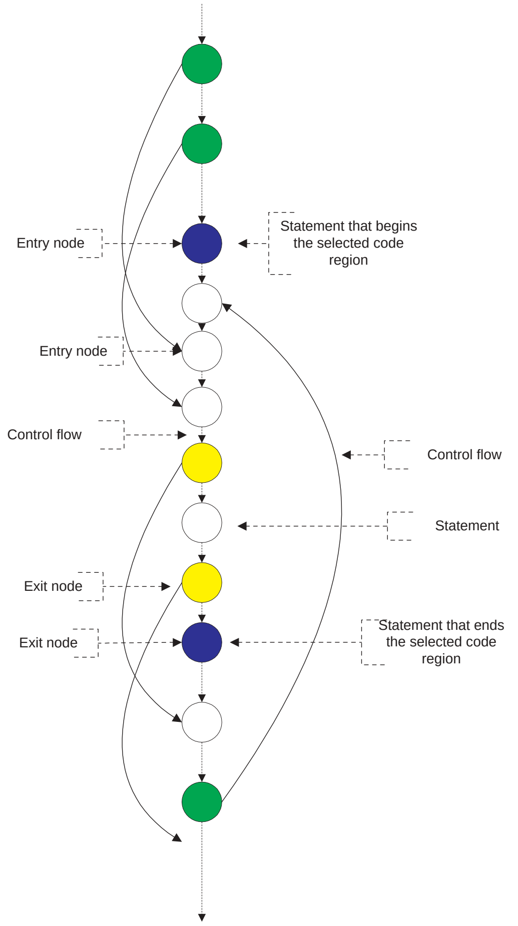
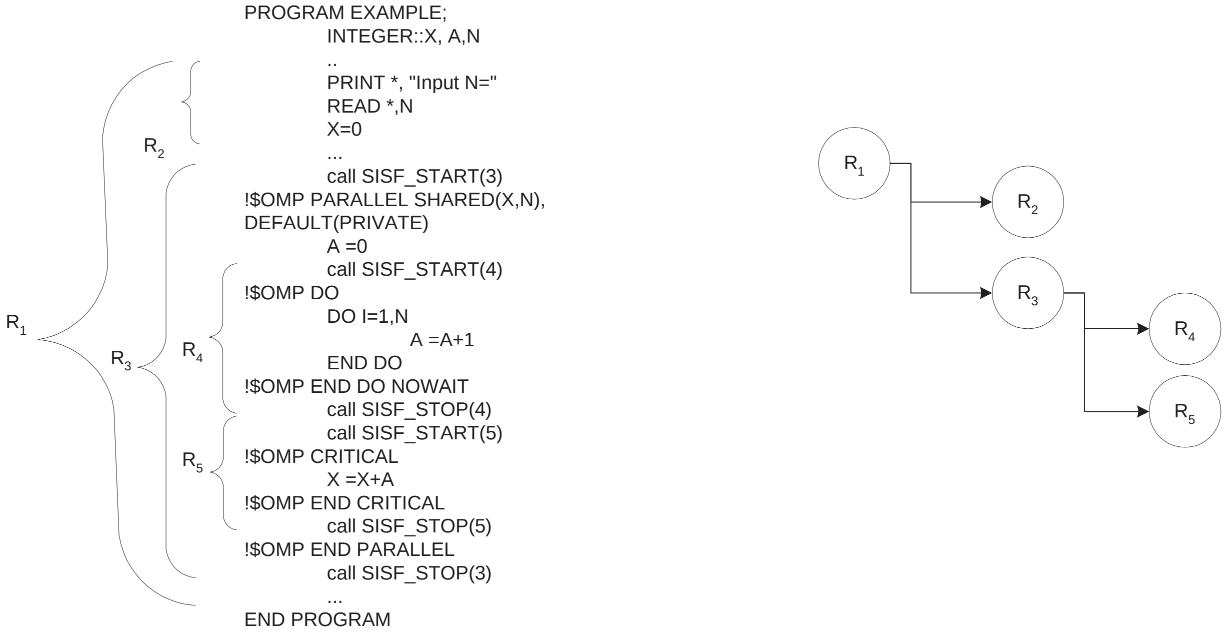

Entry node
Statement that begins the selected code region
Entry node
Control flow
Control flow
Statement
Exit node
Exit node
Statement that ends the selected code region

Figure 3: A code region with several entry and exit points

Figure 4 shows an excerpt of an OpenMP code together with its associated DRG.

Call graph techniques have been widely used in performance analysis. Tools such as Vampir [20], gprof [11, 10], CXperf [14] support a call graph which shows how much time was spent in each function and its children. In [7] a call graph is used to improve the search strategy for automated performance diagnosis. However, nodes of the call graph in these tools represent function calls [10, 14]. In contrast our DRG defines a node as an arbitrary code region (e.g. function, function call, loop, statement, etc.).



```
PROGRAM EXAMPLE;
          INTEGER::X, A,N
          ..
          PRINT *, "Input N="
          READ *,N
          X=0
          ...
          call SISF_START(3)
!$OMP PARALLEL SHARED(X,N),
DEFAULT(PRIVATE)
          A =0
          call SISF_START(4)
!$OMP DO
          DO I=1,N
                    A =A+1
          END DO
!$OMP END DO NOWAIT
          call SISF_STOP(4)
          call SISF_START(5)
!$OMP CRITICAL
          X =X+A
!$OMP END CRITICAL
          call SISF_STOP(5)
!$OMP END PARALLEL
          call SISF_STOP(3)
          ...
END PROGRAM
```

Figure 4: OpenMP code excerpt with DRG

4.2 Generating and Building the Dynamic Code Region Call Graph

Calling code region r_2 inside a code region r_1 during execution of a program establishes a parent-children relationship between r_1 and r_2. The instrumentation library will capture these relationships and maintain them during the execution of the program. If code region r_2 is called inside r_1 then a data entry representing the relationship between r_1 and r_2 is generated and stored in appropriate profiles/trace files. If a code region r is encountered that isn't child of any other code region (e.g., the code region that is executed first), an abstract code region is assigned as its parent. Every code region has a unique identifier which is included in the probe inserted by SIS and stored in the instrumentation description file.

The DRG data structure maintains the information of code regions that are instrumented and executed. Every thread of each process will build and maintain its own sub-DRG when executing.

In the pre-processing phase (cf. Figure 1) the DRG of the application will be built based on the individual sub-DRGs of all threads. A sub-DRG of each thread is computed by processing the profiles/trace files that contain the performance data of this thread. The algorithm for generating DRGs is described in detail in [24].

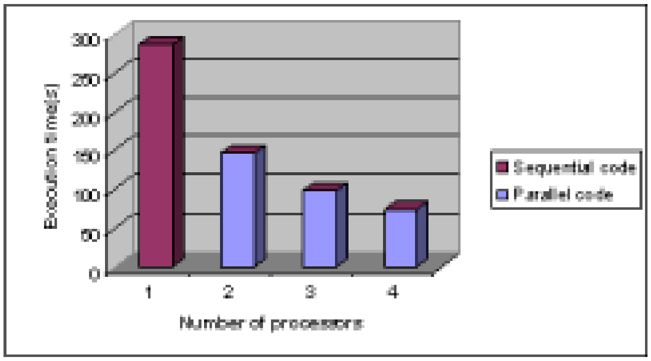


Figure 5: Execution time(s) of the MD application

5 Experiments

We have implemented a prototype of SCALEA which is controlled by command-line options and user directives. Code regions including arbitrary code regions can be selected through specific SIS directives that are inserted in the input program. Temporal performance overheads according to the classification shown in Figure 2 can be selected through command-line options. Our visualization capabilities are currently restricted to textual output. We plan to build a graphical user interface by the end of 2001. The graphical output except for tables of the following experiments have all been generated manually. More information of how to use SIS and post-execution analysis can be found in [25, 24].

Overhead	**2CPUs**	**3CPUs**	**4CPUs**
Loss of parallelism	0.025	0.059	0.066
Control of parallelism	1.013	0.676	0.517
Synchronization	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: Overheads (sec) of the MD application. T_i, T_u, T_o are identified, unidentified and total overhead, respectively.

In this section, we present several experiments to demonstrate the usefulness of SCALEA. Our experiments have been conducted on Gescher [23] which is an SMP cluster with 6 SMP nodes (connected by FastEthernet) each of which comprises 4 Intel Pentium III Xeon 700 MHz CPUs with 1MB full-speed L2 cache, 2Gbyte ECC RAM, Intel Pro/100+Fast Ethernet, Ultra160 36GB hard disk is run with Linux 2.2.18-SMP patched with perfctr for hardware counters performance. We use MPICH [12] and pgf90 compiler version 3.3. from the Portland Group Inc.

5.1 Molecular Dynamics (MD) Application

The MD program implements a simple molecular dynamics simulation in continuous real space. This program obtained from [27] has been implemented as an OpenMP program which was written by Bill Magro of Kuck and Associates, Inc. (KAI).

The performance of the MD application has been measured on a single SMP node of *Gescher*. Figure 5 and Table 2 show the execution time behavior and measured overheads, respectively. The results demonstrate a good speedup behavior (nearly linear). As we can see from Table 2, the total overhead is very small and large portions of the temporal overhead can be identified.

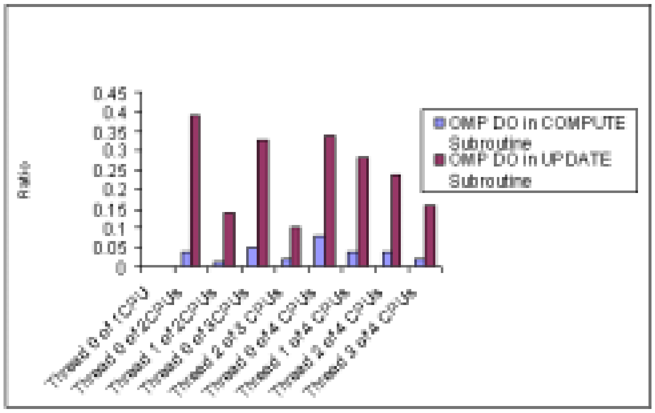


Figure 6: The L2 cache misses/cache accesses ratio of OMP DO regions in the MD application

The time of the sequential code regions (unparallelized) doesn't change as it is always executed by only one processor. Loss of parallelism for an unparallelized code region r in a program q is defined as $t_r - \frac{t_r}{p}$ if p processors are used to execute q and t_r is the sequential execution time of r. By increasing p it can be easily shown that the loss of parallelism increases as well which is also confirmed by the measurements shown in Table 2.

Control of parallelism – mostly caused by loop scheduling – actually decreases for increasing number of processors. A possible explanation for this effect can be that for larger number of processors the master thread processes less loop scheduling phases than for a smaller number of processors. The load balancing improves by increasing the number of processors/threads in one SMP node which at the same time decreases synchronization time.

We then examine the cache miss ratio – defined by the number of L2 cache misses divided by the number of L2 cache accesses – of the two most important OMP DO code regions namely *OMP DO COMPUTE* and *OMP DO UPDATE* as shown in Figure 6. This ratio is nearly 0 when using only a single processor which implies very good cache behavior for the sequential execution of this code. All data seem to fit in the L2 cache for this case. However, in a parallel version, the cache miss ratio increases substantially as all threads process data of

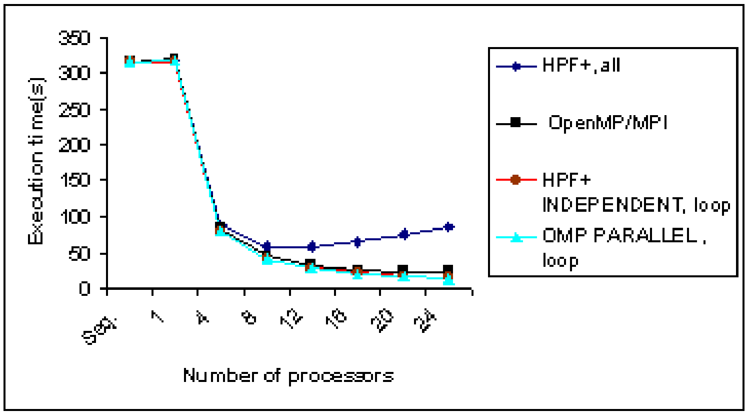


Figure 7: Execution times of the HPF+ and OpenMP/MPI version for the backward pricing application

global arrays that are kept in private L2 caches. The cache coherency protocol causes many cache lines to be exchanged between these private caches which induces cache misses. It is unclear, however, why the master thread has a considerably higher cache miss ratio then all other threads. Overall, the cache behavior has very little impact on the speedup of this code.

5.2 Backward Pricing Application

The backward pricing code [8] 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 is compiled to an MPI program, and second, a mixed version that combines HPF+ with OpenMP. For the latter version, VFC generates an OpenMP/MPI program. HPF+ directives are used to distribute data onto a set of SMP nodes. Within each node an OpenMP program is executed. Communication among SMP nodes is realized by MPI calls.

The execution times for both versions are shown in Figure 7. The term “all” in the legend denotes the entire program, whereas “loop” refers to the main computational 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 (overall 8 processors). Tables 3 and 5 display 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). Two routines (Update_HALO and MPI_Init) are mainly responsible for the high control of parallelism overhead of the HPF+ version. 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 HPF+ version implies a much higher overhead for these two routines compared to the OpenMP/MPI reason because it employs a separate process on every CPU of each SMP node. Whereas the OpenMP/MPI version uses one process per node.

5.3 LAPW0

LAPW0 [4] is a material science program that calculates the effective potential of the Kohn-Sham eigenvalue problem. LAPW0 has been implemented as a Fortran MPI code which can be run across several SMP nodes. The pgf90 compiler takes care of exchanging data between processors both within and across SMP nodes. 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*
- 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 8 and 9, respectively. LAPW0 has been examined for a problem size of 36 atoms which are distributed onto the processors of a set of SMP nodes. Clearly when using 8, 16, and 24 processors we can't reach optimal load balance, whereas 1, 2, 4, 6, 12 and 18 processors display a much better load imbalance. This effect is confirmed by SCALEA (see Figure 9) for the the most computationally intensive routines of LAPW0 (*Interstitial Potential* and *Loop_50*).

Overall, LAPW0 scales poorly due to load imbalances and large overheads due to loss of parallelism, data movement, and synchronization; see Table 7. 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 Table 7). The main sources of the control of parallelism overhead is caused by MPI_Init (see Figure 10). SCALEA also discovered the main subroutines that cause the loss of parallelism overhead: *FFT_REAN0*, *FFT_REAN3*, and *FFTP_REAN4* all of which are sequentialized.

6 Related Work

Paraver [26] is a performance analysis tool for OpenMP/MPI tools which dynamically instruments binary codes and determines various performance parameters. This tool does not cover the same range of performance overheads supported by SCALEA. Moreover, tools that use dynamic interception mechanisms commonly have problems to relate performance data back to the input program.

Ovaltine [2] measures and analyses a variety of performance overheads for Fortran77 OpenMP programs. Paradyn [18] is an automatic performance analysis tool that uses dynamic instrumentation and searches for performance bottlenecks based on a specification language. A function call graph is employed to improve performance tuning [7].

Recent work on an OpenMP performance interface [19] based on directive rewriting has similarities to the SIS instrumentation approach in SCALEA and to SCALA [9] – the predecessor system of SCALEA. Implementation of the interface (e.g., in a performance measurement library such as TAU) allows profiling and tracing to be performed. Conceivably, such an interface could be used to generate performance data that the rest of the SCALEA system could analyze.

The TAU [22, 17] performance framework is an integrated toolkit for performance instrumentation, measurement, and analysis for parallel, multithreaded programs. SCALEA uses TAU instrumentation library as one of its tracing libraries.

PAPI [5] specifies a standard API for accessing hardware performance counters available on most modern microprocessors. SCALEA uses the PAPI library for measuring hardware counters.

gprof [11, 10] is a compiler-based profiling framework that mostly analyses the execution behavior and counts of functions and function calls.

VAMPIR [20] is a performance analysis tool that processes trace files generated by VAMPIRtrace [21]. It supports various performance displays including time-lines and statics that are visualized together with call graphs and the source code.

7 Conclusions and Future Work

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 is based on a novel classification of performance overheads for shared and distributed memory parallel programs. SCALEA is among the first performance analysis tools that combines source code and HW-profiling in a single system which significantly extends the scope of possible overheads that can be measured and examined. 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 enables a detailed overhead analysis for every code region. The

Processors	**Sequential**	**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 application. 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 for the backward pricing application.

DRG is not restricted to function calls but also covers loops, I/O and communication statements, etc. Moreover, it allows to analyze arbitrary code regions that can vary from a single statement to an entire program unit. This is in contrast to existing approaches that frequently use a call graph which considers only function calls. 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 a performance specification language in order to support automatic performance bottleneck analysis.

The SISPROFILING measurement library for DRG and overhead profiling is an extension of TAU's profiling capabilities. Our hope is to integrate these features into the future releases of the TAU performance system so that these features can be offered more portably and other instrumentation tools can have access to the API.

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. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.

[4] 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.

[5] 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.

[6] 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.

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 application.

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 the backward pricing application.

[7] 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.

[8] 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.

[9] T. Fahringer, B. Scholz, and X. Sun. Execution-Driven Performance Analysis for Distributed and Parallel Systems . In *Proc. of the 2nd International ACM Sigmetrics Workshop on Software and Performance (WOSP'2000)*, Ottawa, Canada, September 2000. ACM Press.

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

[11] 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.*

[12] 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.

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

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

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

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

[17] Allen Malony and Sameer Shende. Performance technology for complex parallel and distributed systems. In *In G. Kotsis and P. Kacsuk (Eds.), Third International Austrian/Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, pages 37–46. Kluwer Academic Publishers, Sept. 2000.

[18] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel

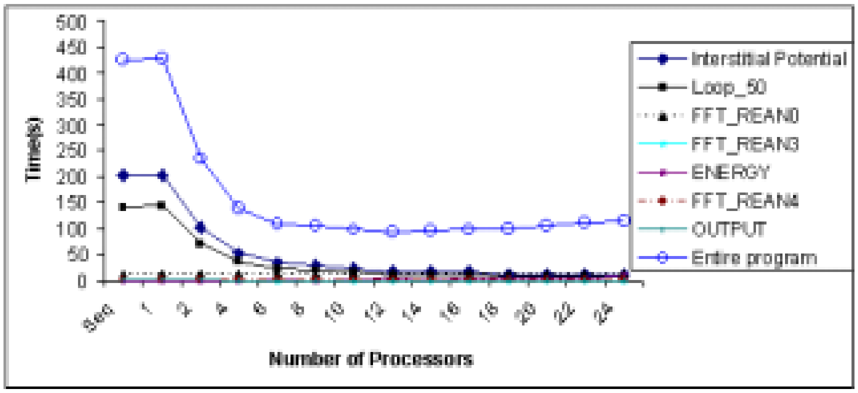


Figure 8: Execution times for LAPW0 code regions.

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.

performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[19] Bernd Mohr, Allen Malony, Sameer Shende, and Felix Wolf. Towards a performance tool interface for openmp: An approach based on directive rewriting. In *EWOMP'01 Third European Workshop on OpenMPI*, Sept. 2001.

[20] 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.

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

[22] Sameer Shende, Allen Malony, Janice Cuny, Kathleen Lindlan, Peter Beckman, and Steve Karmesin. Portable profiling and tracing for parallel, scientific applications using C++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, pages 134–147, New York, August 3–4 1998. ACM Press.

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

[24] Hong-Linh Truong and Thomas Fahringer. Scalea - a performance analysis system for distributed and parallel programs. Technical report, Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria, April 2001.

[25] 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.

[26] 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.

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

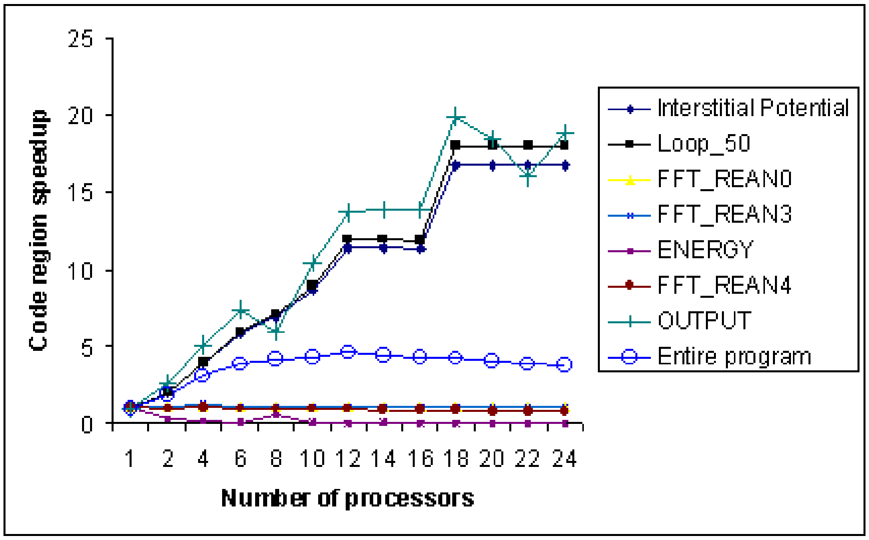


Figure 9: Speedup values for LAPW0 code regions.

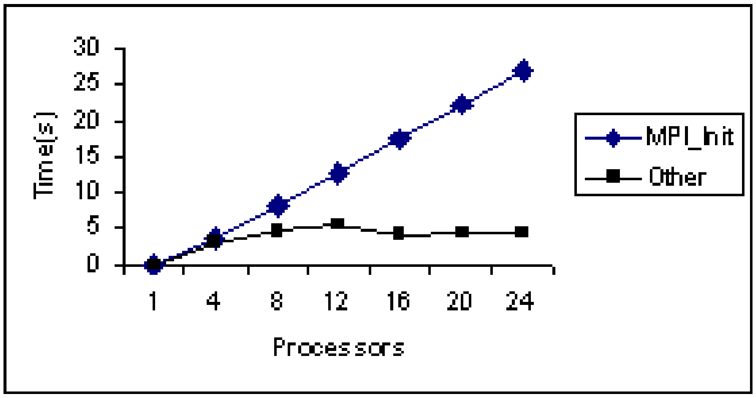


Figure 10: Sources of control of parallelism overhead for LAPW0.