Execution-Driven Performance Analysis for Distributed and Parallel Systems *

Thomas Fahringer[†] Bernhard Scholz[†] Xian-He Sun[‡]

[†]Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstrasse 22, A-1092, Vienna, Austria
tf@par.univie.ac.at

[‡] Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, U.S.A.
sun@cs.iit.edu

Abstract

Lack of effective performance analysis environments is a major barrier to the broader use of distributed and parallel computing. Many existing performance tools collect and visualize performance data for programs that have been generated and transformed by a compiler. It is very difficult for the programmer to meaningfully relate such performance data back to the input program. We present a novel framework for a portable instrumentation in order to selectively monitor the performance of distributed and parallel programs. Among others, a rich set of array information can be collected. Code transformations are recorded in order to maintain the relationship between collected performance data of a compiler-generated code and the input program. Performance overhead introduced by a transformation system can be separately measured and displayed in relation to the performance behavior of the input program. Many performance metrics and statistics are computed. Performance data can be filtered, summarized, analyzed, and visualized at various levels of detail. Experiments are presented to illustrate the usefulness of our system.

1 Introduction

The development of efficient programs for distributed and parallel systems is an error-prone and time-consuming process that may involve many cycles of code editing, compiling, executing, and performance analyzing. For performance-oriented program development, programmers must understand the intricate details of the programming model, program transformation system, and architecture.

Many different programming paradigms such as explicit message passing [14], High Performance Fortran (HPF) [19], OpenMP [7], Java RMI [36], and HPC++ [25] have been introduced for distributed and parallel architectures. There is a large variety of reasons that can cause poor performance of a distributed or parallel program. For instance, limited data and work parallelism, high communication or synchronization overhead, uneven load balance, etc. Performance bottlenecks can be caused by the intricate structure and details of a program, by the target architecture and by various code transformation systems. Compilers may restructure a high-level program into a different program (for instance, into lower level or optimized program) which makes it very difficult to relate performance behavior of the transformed code back to the original input program. A performance analysis system must have access to transformation systems in order to record code changes and associate performance problems with the input program.

Many existing performance monitoring and analysis systems focus on collecting and presenting performance data for programs that have been generated and modified by transformation systems. Frequently performance information is provided for low-level system calls (i.e., operating and runtime library calls) that cannot be traced back to the input program. Furthermore, many performance tools provide only summary information for entire programs without relation to specific program points of interest. Crucial correlation of performance bottlenecks with exact positions in the input program is disabled which severely restricts the usefulness of such tools.

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFB F011. The third author was supported in part by National Science Foundation under NSF grant ASC-9720215 and CCR-9972251

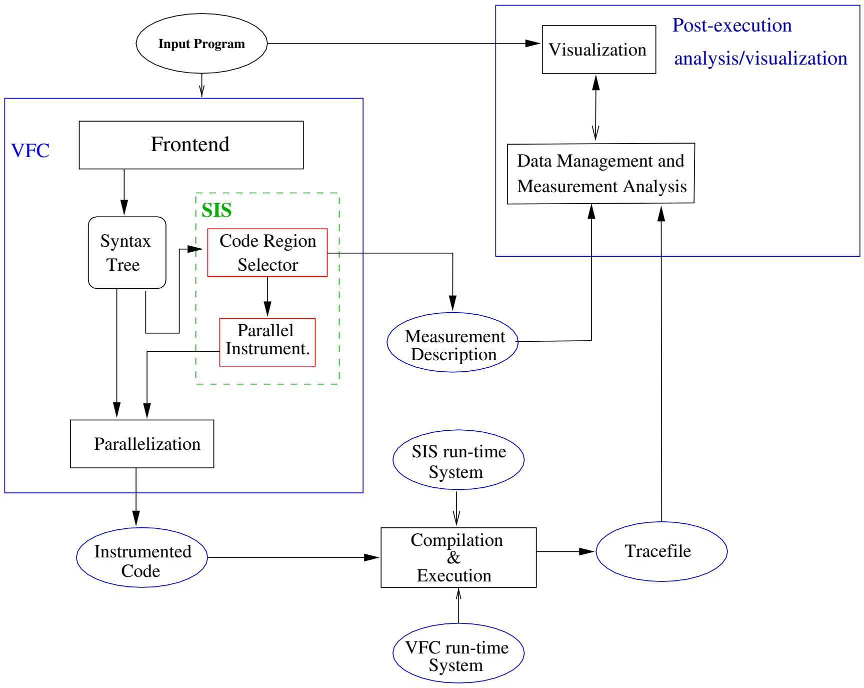

Figure 1. Execution-Driven Performance Analysis System

In this paper we describe Scala which is an instrumentation, measurement, and post-execution performance analysis system for distributed and parallel programs that combines a portable instrumentation system, performance data correlation, data management and measurement analysis, and an interface for performance visualization. Various instrumentation features are supported that enable both comprehensive and selective monitoring in order to control the monitoring overhead and the performance data generated. Performance data correlation maintains the performance relationship between the input program and code changes applied by transformation systems. Data management and measurement analysis supports a rich set of performance data reduction, filtering, summary, and analysis techniques. Many performance metrics and statistics can be computed. A particular new feature of Scala is its ability to collect information about array references (i.e., shape, rank, distribution, alignment, and mapping of arrays). Scala can be used to monitor and analyze the performance of many different programming paradigms, including data parallel, task parallel and message passing paradigms. Moreover, high-level paradigms such as HPF are supported. Scala supports several trace formats which allows the use of various visualization systems (e.g. Medea [5], Upshot [18], and Grace [17]). Finally, Scala has highly sophisticated scalability analysis integrated, that examines the scaling behavior of a program for varying input data and machine sizes. The work about scalability analysis goes beyond the scope of this paper and has been described in [35].

In the next Section, we give an overview of Scala as an integrated system of a restructuring and optimizing compiler. Section 3 gives a detailed overview of the Scala instrumentation system. The following section presents the performance data gathered during execution of a program and how this data can be correlated with the input program. In section 5 we describe the various performance metrics and statistics as computed by Scala. Performance visualization is discussed in the following section.

Section 7 shows experiments with two real-world application codes in order to demonstrate the usefulness of Scala. We conclude with a brief discussion of related work, conclusions and directions for future work.

2 Scala

Scala is a post-execution performance system that instruments, measures, and analyses the behavior of distributed and parallel programs. The architecture of Scala is based on a portable instrumentation system, runtime-libraries that collect performance data during program execution, and post-execution performance analysis that computes various performance metrics and relates them back to the input program. In addition, Scala supports an interface to external visualization systems. Although Scala has been integrated with an existing compiler it can be easily ported to front-ends and compilers for other programming languages and architectures by porting its instrumentation and runtime libraries.

Figure 1 shows the architecture of Scala as an integrated system of VFC [3] which is a compiler that translates Fortran programs into message passing programs (F90 with MPI message passing calls). The input programs (F77, F90, HPF, and explicit message passing programs) of Scala are processed by the compiler front-end which generates an abstract syntax tree (AST). The Scala Instrumentation System (SIS) enables the user to select (by directives or command-line options) code regions of interest. Based on the selected code regions, SIS automatically inserts monitoring code in the AST which will collect all relevant performance information during execution of the program. SIS also generates a measurement description file that enables relating all gathered performance data back to the input program. This is a crucial aspect of Scala, as instrumentation may be done at a different level (e.g. message passing program) than the original input program (e.g. HPF). The compiler then generates an instrumented distributed or parallel program which will be executed on the target architecture. Note that the compiler can also process explicitly distributed and parallel programs for instrumentation and performance analysis. During execution all relevant performance data is collected in a trace-file.

The trace-file provides a generic input for a post-mortem data management and measurement analysis to reduce, filter, summarize, and analyse performance information. Among others, a variety of performance metrics are computed which includes speedup, efficiency, efficacy, communication, and work distribution. An interface for a visualization system is employed in order to graphically display various performance statistics and profiles that can be shown together with the original input program.

The general structure of Scala comprises several modules which combined together provide a robust environment for advanced performance analysis:

- Scala Instrumentation System (SIS)
- Performance data correlation
- Data management and measurement analysis
- Performance visualization interface

In the following sections we will describe each of this modules in more detail.

3 Scala Instrumentation System (SIS)

SIS inserts instrumentation code in the program for each information of interest which includes:

- timing events
- execution frequency events
- values for program unknowns (unknowns in array subscript expressions, loop bounds, etc.)
- array information (rank, shape, alignment, distribution, mapping, etc.)

For each information that is traced during execution of the program an *event record* (event identification, source file, location in source file, etc.) is written to the trace file.

Performance systems are commonly susceptible to generating vast amounts of performance data. Moreover, the monitoring overhead may impact the performance behavior of a program while it is executing. Therefore, it is of paramount importance for the user to control instrumentation of a program. SIS supports the programmer to control monitoring and generating performance data through:

- Selective instrumentation of specific types of code regions
- Instrumentation of arbitrary code regions
- Selective turning on/off monitoring

SIS enables the programmer to optionally measure all code regions of a specific type which includes:

- Program
- Procedures (e.g. functions and subroutines)
- Loops
- Special loops (e.g. Fortran independent loops)
- Communication Statements
- Input/Output Operations
- Arrays (e.g. distribution, alignment, and mapping information)

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

```
!sis$ SIS_BEGIN_MEASURE reference

      code region

!sis$ SIS_END_MEASURE reference
```

The directive SIS_BEGIN_MEASURE must be inserted before the region starts whereas after the code region ends, the directive SIS_END_MEASURE is inserted. The *code region* of interest is uniquely identified by an integer number *reference*. Note that there can be several entry and exit points for a code region. Appropriate directives must be inserted in each entry and exit point of a given code region.

Furthermore, SIS provides specific directives in order to control monitoring. The following directive enables the programmer to turn on and off monitoring of a program.

```
!sis$ SIS_TRACE_ENABLE

      code region

!sis$ SIS_TRACE_DISABLE
```

For instance, the following example instruments a portion of a program, where the inner loop is not measured by using the dynamic tracing facility mentioned above.

```
      n=10
!sis$ SIS_BEGIN_MEASURE 10
      j=100
      do i,1,100
!sis$ SIS_TRACE_DISABLE
         do k,1,1000
            j=1/j+1/(j*j)
         end do
!sis$ SIS_TRACE_ENABLE
      end do
!sis$ SIS_END_MEASURE 10
```

3.1 Instrumenting Code Regions

Our instrumentation model is designed to monitor the performance of code regions where a *code region* r is a portion of a code (sequence of statements). Program points B_r, which are located at the entry of r, are *entry points* of r. The *exit points* X_r of r are positioned at the exits of r. For tracing the code region r we need a unique pair of events (e_b, e_x) where e_b denotes the begin event of r and e_x the exit event. When r is entered through a program point in B_r, the event e_b is triggered. After leaving code region r through an exit point X_r the event e_x is logged. Note that our instrumentation model is correct if and only if there is no program path between two different program points in B_r. The same condition must hold for exit points X_r.

In our instrumentation model more than one processor can execute the same program (single program multiple data programming paradigm [16]). In order to compute performance parameters we need to know which processor invokes an event at what time. Therefore, we identify each processor of the underlying system with a unique label p and introduce a global time[1] τ [28], which is available for all processors of the system. Tracing is implemented by a *write operation* ω to a *global stream* Γ. The write operation can be issued simultaneously such that there is no need for an order on Γ. The write operation puts a tuple (τ, e, p) onto Γ where τ is the current global time, e the triggered event, and p the label of the processor. Note that already written tuples on global stream Γ can never be overwritten. After executing the program the global stream contains sufficient information to reconstruct the order and the elapsed time of *states*. More formal, a state s is associated with a code region r and a processor p. A state s is activated if entry event e_b of r occurs and s is deactivated by its exit event e_x.

To implement our instrumentation model for a given program we insert a write operation ω for all code regions r at the entry points B_r such that

$$\forall l \in B_r : \text{insert}\, \omega(\tau, e_b, p) \text{ at } l$$

and exit points X_r such that

$$\forall l \in X_r : \text{insert } \omega(\tau, e_x, p) \text{ at } l.$$

Note that the model described above can be extended to trace additional program behavior characteristics. E.g., communication overhead and memory requirement. This additional information can be saved on the stream as well.

[1] If at the same point in time all processors request a time-stamp τ, they will retrieve the same result with a specific accuracy.

4 Performance Data Correlation

A crucial aspect of performance analysis is to relate performance information back to the original input program. A compiler may imply many codes changes (e.g. copying, hoisting and sinking of code sections) so that the relationship between its execution dynamics and its input program is obscure. An example for such a code change is optimizing communication through latency hiding [13]. This optimization causes send statements to be hoisted upwards and receive statements downwards in the control flow of a program. Other transformations may fuse or distribute loops that respectively results in collapsing several loops or generating several loops out of a specific loop. Irregular programs are frequently compiled based on the inspector/executor paradigm [3] which causes a loop to be transformed into a preparation (inspector) and an execution (executor) phase. In order to examine which performance aspect corresponds to what code region, a *measurement description file* is generated and updated while the compiler is applying code transformations. Through the measurement description file the relationship between the associated program regions in the generated and original program is maintained. For instance, for send and receive statements the source of communication – which can be an array assignment – in the original program is recorded.

Whereas the measurement description file collects information at compile-time, SIS has very sophisticated support for runtime data collection which includes timing and event data, values for program unknowns, and array information.

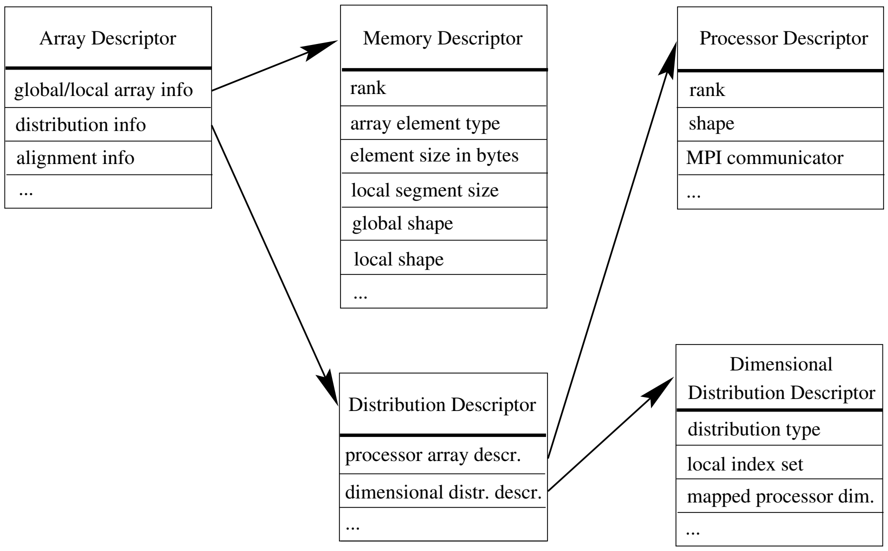

Figure 2. Array runtime information

Specific instrumentation statements are provided to monitor values of program unknowns such as scalar variables that appear in loop bound or array subscript expressions. Providing characteristic values for program unknowns can be very useful to support performance prediction [9, 1] and various program analysis and optimization techniques [11, 12]. A particular novel feature of Scala is its ability to collect comprehensive information about important data structures such as arrays. Figure 2 shows the information that can be collected for arrays during runtime which includes rank and shape of arrays, size of local and global arrays, dimensional distribution, alignment and mapping information, etc. Note, besides general array information much of this data is related to data parallel programs such as HPF. Array information can be traced at any program point where the array is live [2]. This information has been shown to be very useful, for instance, to determine all data distributions that can reach a specific procedure, the shape and rank of allocateable arrays, and memory requirements of all arrays live at a given program point.

Note that the array information is maintained by the runtime libraries as provided by the underlying compiler [3] and can be collected during execution by a specific instrumentation of SIS.

5 Data management and measurement analysis

The dynamic properties of a program can be investigated by measuring the associated execution time and the interactions with the underlying distributed or parallel system. Tracing the performance data can lead to large amount of performance information which needs to be filtered and reduced to provide the user with a compact and easy-to-interpret set of parameters describing the characteristics of the program. The level of detail of this set depends mainly on the scope of the performance analysis. As an example, if the communication behavior of a program is under investigation, a performance analysis tool should be able to extrapolate from measured data only the most significant information regarding the communication activities such as total time spent transmitting a message, amount of data transmitted, and communication protocols used in each transmission.

The data collected by Scala need to be analyzed and reduced so that they can be sufficiently detailed for analyzing the behavior of parallel and distributed programs. We provide extensive performance data filtering through a range of performance metrics from aggregate (i.e., entire program) through procedure, procedure calls, loops, arbitrary code sections, and individual source code lines. Performance metrics for program regions can be split into individual performance components (for instance, computation, communication, and compiler organization overhead).

The *data management and measurement analysis* module (see Figure 1) implements several data-reduction techniques that can be applied in isolation or in combination for data management. Filtering is the simplest form of data reduction to eliminate data that do not meet specified performance analysis criteria retaining only the pertinent characteristics of performance data. Among the range of statistical techniques that can be applied to a data set, mean, standard deviation, percentiles, minimum, and maximum are the most common and provide a preliminary insight on the program performance, while reducing the amount of data. Other statistical methods such as the analysis of distribution and variability provide more detailed information on the dynamic behavior of distributed and parallel programs. For example, the analysis of distribution gives information on how the communication and computation are distributed across processors and the user or compiler can apply transformations to improve the performance. Moreover, the coefficients of variation of communication time and computation time are good metrics to express the quality of work and communication distributions across processors. In most cases, the data measured need to be scaled in a common interval (i.e. 0 mean and 1 standard deviation or 0-1) so that further statistical techniques can be successfully applied. Timing parameters can also be scaled in a more significant metric for the analysis such as from seconds to microseconds or nanoseconds.

All these statistics can be applied to the entire data set as well as to a subset of data such as a reduced number of processors or sliding windows of the most recent data. However, to manage large amount of data and large number of performance parameters, advanced statistical techniques need to be applied. Clustering is one of the most common techniques used to reduce the amount of performance data and consequently reduce the number of performance parameters needed to identify sources of performance losses. Statistical clustering is a multidimensional technique useful to automatically identify equivalence classes. For example, clustering classifies processors such that all processors in one cluster have similar behavior.

6 Performance Visualization

Visualization of performance metrics and statistics, and also dynamic information about arrays is of crucial importance to support the programmer in performance tuning of distributed and parallel programs. Scala supports two trace formats – ALOG and Scala specific format – for collected performance data which enables the usage of several performance visualization systems. Based on the ALOG trace format we can use visualization systems of well-established systems such as Medea [5], TAU [30], and Upshot [18]. Medea is a post-mortem performance analysis and visualization system. Among others we use Medea to derive and visualize performance metrics together with the input program based on ALOG trace files. This work has been described in detail in [6].

Scala also generates Grace [17] data files for various 2D performance data visualizations. A Scala specific trace format is used as input to our own graphical user interface. This interface has been developed based on an HTML display engine which enables the user to view the input program together with performance parameters and dynamic array information (see Section 4). Our user interface is connected with Scala through a communication layer implemented as a Java RMI distributed middle-ware (supports Java JDK 2.0). Scala and the graphical user interface can thus run on different workstations possibly on different networks. Such an architecture has been shown to be very useful, as the graphical user interface may require only trace-files, measurement description files, and the input program without having full access to VFC and SIS.

7 Experiments

In this section we present two experiments to demonstrate the usefulness of Scala for real-world applications. In the first experiment Scala is employed to examine the performance of a system for pricing of financial derivatives [8] developed by Prof. Dockners' group at the University of Vienna. The pricing system has been implemented from scratch as an HPF/Fortran90 program that exploits data parallelism and was executed on a NEC Cenju-4 [32] distributed memory parallel machine.

In the second experiment we used Scala to analyze the performance of a Monte-Carlo ion implementation simulator for three-dimensional crystalline structures [21, 4] as developed by Prof. Selberherr and his associates at the Vienna University of Technology. The ion implementation simulator is a heterogeneous code which comprises both Fortran and C code segments. A master-slave model has been manually implemented as an MPI message passing program that exploits both data and task parallelism and runs on a cluster of heterogeneous workstations and networks.

7.1 Pricing of Financial Derivatives

The pricing of derivate products is an important field in finance theory. A *derivative* (or *derivative security*) is a financial instrument whose value depends on other, so called underlying securities [22]. Examples are stock options and variable coupon bonds, the latter paying interest rate dependent coupons. The pricing problem can be stated as follows: what is the price today of an instrument which will pay some cash flows in the future, depending on the development of an underlying security, e.g. stock prices or interest rates ? For simple cases analytical formulas are available, but for a range of products, whose cash flows depend on a value of a financial variable in the past - so called *path dependent* products - Monte Carlo simulation techniques have to be applied [33, 27]. By utilizing massively parallel architectures very efficient implementations can be achieved [24, 38]. For a detailed description of the technique implemented see [8] and [31].

```
...
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (BLOCK) ONTO PR :: VALUE
...
   TYPE(BOND) :: B                                              ! the bond to be priced
   INTEGER :: PATH(0: N_STEPS)                                  ! path in the Hull and White tree
   REAL(DBLE) :: VALUE(1: N)                                    ! all path results

   !HPF$ INDEPENDENT, NEW(PATH), ON HOME(VALUE(I))
   DO I = 1, N
     PATH = RANDOM_PATH(0,0,N)                                  ! select a path starting at node (0,0)
                                                                ! discount the bond's cash flow to time 0
     VALUE(I) = DISCOUNT(0,CASH_FLOW(B,1,N),FACTORS_AT(PATH))
   END DO
   PRICE = SUM(VALUE)/N                                         ! mean value
...
```

Figure 3. HPF DO-Independent Code of the Pricing System

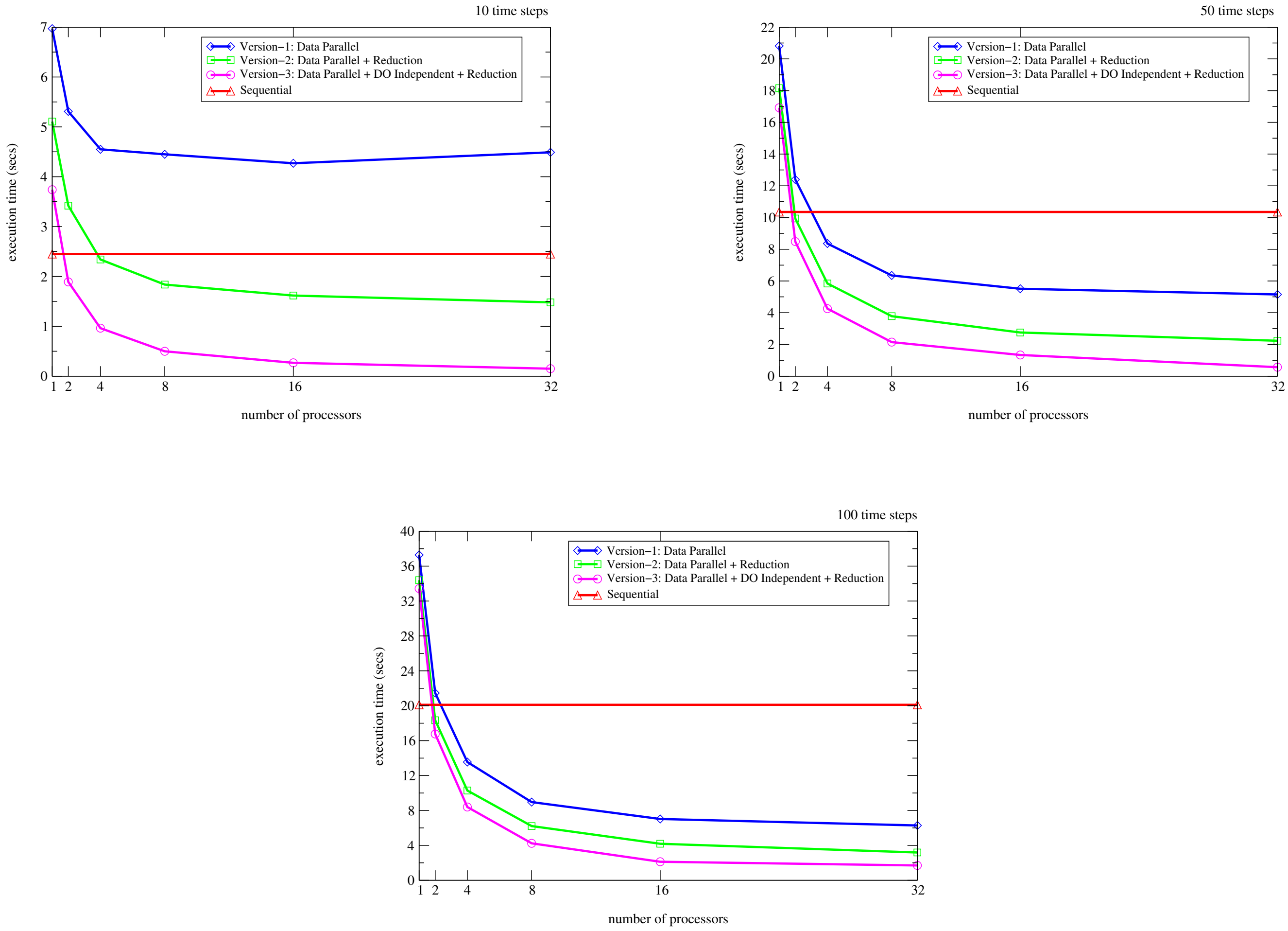

Figure 4. Measured accumulated execution times for all three versions of function TRAVERS_DISCOUNT of the system for pricing of financial derivatives (three HPF and one sequential version). Experiments have been conducted for three different problem sizes (number of time steps) and varying number of processors on a NEC Cenju-4

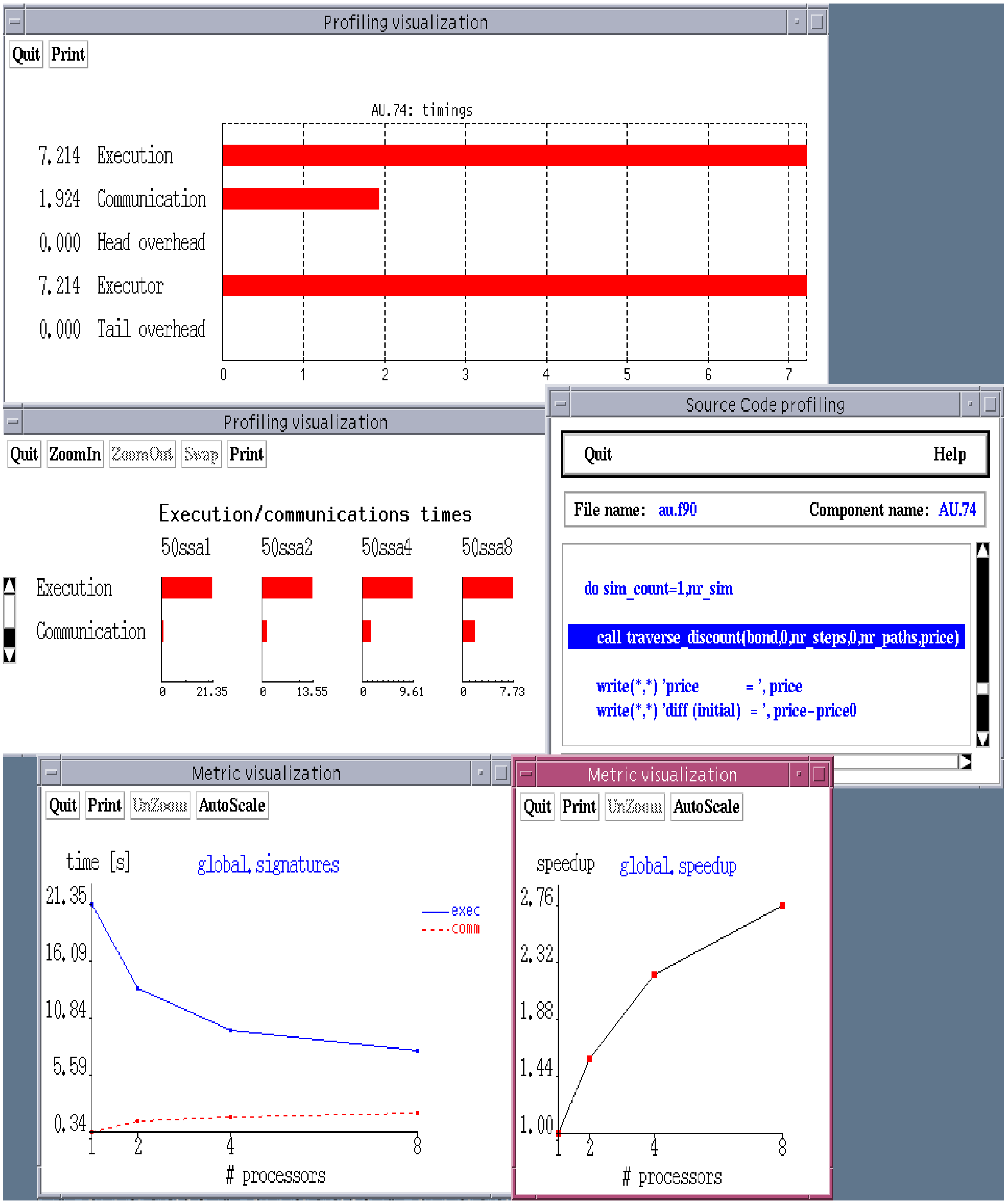

Figure 5. Snapshots of the MEDEA system which displays various performance metrics together with the code section of interest (call to function *TRAVERSE_DISCOUNT* in middle-right window).

The Monte Carlo simulation is based on a discrete representation of a stochastic process that describes the dynamics of the underlying security over time [23]. In the case of interest rate dependent products, the *Hull and White tree* describes the future development of the short rate, which is used to calculate the entire interest rate curve for a specific state of the system [22]. Each state is represented by a node in a directed graph and has three successor nodes, representing increasing, constant, and decreasing interest rates.

The HPF/Fortran90 code segment in Figure 3 shows the main loop of the Monte Carlo simulation procedure TRAVERSE_DISCOUNT.

Parallelization

We encoded three different versions of the pricing system and executed them on a NEC Cenju-4 [32] distributed memory parallel machine. First, a data parallel version which is based on distributing array VALUE block-wise onto the maximum number of processors – by using the HPF intrinsic function NUMBER_OF_PROCESSORS() – that are available on a given architecture. The summation of the path results is replicated which causes communication. Second, we improved the first version by using the HPF reduction directive, which causes the summation of the path results to be executed by an efficient machine function. Third, the second version is improved by using the HPF DO-Independent directive, that specifies that each iteration of the main simulation loop can be executed simultaneously. Every iteration of the simulation loop is executed by the processor that owns array element VALUE(I) based on the owner-computes paradigm [3].

Experimental results and further work

VFC has been used to generate Fortran90/MPI programs based on input HPF programs. Each graph in Figure 4 shows the corresponding accumulated execution times for function TRAVERSE_DISCOUNT for all three versions including the sequential implementation of a specific problem size (number of time steps). Note that the sequential execution corresponds to the execution on a single processor. The experiments clearly show that version-3 is superior to all other versions, and version-2 is better than version-1 for all problem and machine sizes. Version-3 exploits more parallelism than version-2 due to the fact that a processor only executes a loop iteration if it owns the corresponding array element (VALUE(I)). Whereas for version-2 every processor executes all loop iterations. The array assignment, however, is only executed if a processor owns VALUE(I). Version-1 sequentializes the reduction operation which causes the largest communication overhead across all code versions. For the smallest problem size (10 time steps), version-1 performs worse than even the sequential version. For increasing problem sizes, however, the difference among the code versions becomes less dramatic, as the impact of communication on the overall performance diminishes. Version-3 shows almost linear speed-up for up to 8 processors. Larger problem sizes should cause a better performance scaling behavior for increasing number of processors.

Figure 5 shows several snapshots of the MEDEA system that visualize various performance metrics together with the input program. Note that all performance measurements have been conducted based on the generated Fortran90/MPI program, whereas the performance metrics are displayed together with the input HPF program (see middle-right window). The upper window displays how much of the execution time (of a 8 processor version) of the call to *TRAVERSE_DISCOUNT* accounts for communication, for executing the main simulation loop in *TRAVERSE_DISCOUNT*, and for compiler overhead (head/tail) before and after the call statement. Note that communication time is part of the execution time. Furthermore, the entire communication – caused by the reduction operation – is spent in the main simulation loop of code version-3. The time for the executor corresponds to the simulation loop and the reduction operation as well. The middle-left window shows how long it took to execute the call to *TRAVERSE_DISCOUNT* and how much has been spend in communication for 1, 2, 4 and 8 processors. The lower-left and lower-right windows, respectively, show the execution signature and speedup of the entire application code for various number of processors.

The parallel simulation algorithm can cause redundant price computations due to the properties of the Hull and White tree. We plan to implement an optimized version that avoids redundant price computations by having one processor compute prices and broadcast the result to all processor that need this data. This procedure implies some extra communication, however, may save substantial computation time.

7.2 Ion Implantation Simulator for Three-Dimensional Crystalline Structures

When simulating semiconductor production processes, ion implantation is a very important, but also one of the most critical steps concerning the simulation time. Due to the complicated structures and the small dimensions of modern semiconductor devices, Monte-Carlo simulation methods often have to be used to describe non-planarity effects and phenomena resulting from ion channelling and large tilt angles. To reach the expected accuracy, three-dimensional simulations have to be performed with very sophisticated models [21], especially for very shallow implantation conditions. Simulations can take up several days and even weeks for realistic problem sizes which made it a first-order target

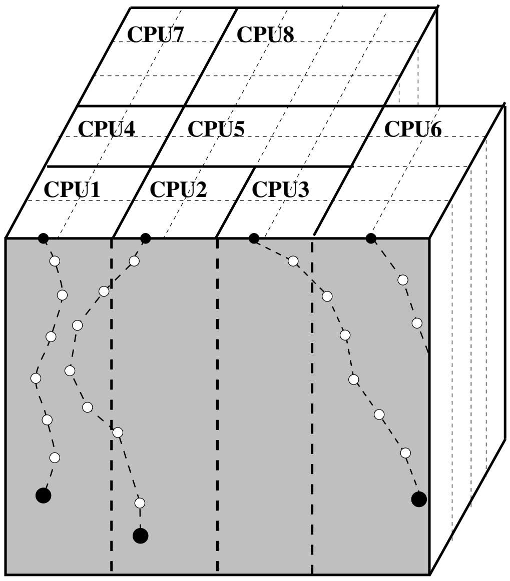

Figure 6. Schematic presentation of the distribution of the geometry among several processors

for parallelization. As a serial code for the ion implantation simulator existed we followed a parallelization strategy that modified the original code and computation models [20, 21, 4] as little as possible by isolating communication and synchronization in a few routines.

The Monte-Carlo ion implantation simulation method is based on the concept that trajectories of a large number of ions – entering the device structure equally distributed over the device surface – are calculated. The distribution of the dopants and crystal damage are derived from the final positions of the ions and the generated point-defects. In order to take the influence of crystal damage on the trajectory of an ion into account a transient simulation is introduced. The ions belonging to the same time step are equally distributed over the device surface and do not interact with each other.

Parallelization

We parallelized the ion implantation simulation by distributing the geometry of the simulation domain based on a master-slave computational model. The bounding box of the simulated structure is split into small rectangular prisms. At the beginning of each time step the master processor distributes one or several prisms to a set of slave processors as shown in Figure 6. Each processor is responsible to calculate the trajectories of all ions residing inside of its assigned prisms. The ion trajectories are computed sequentially by each processor. Parallelism is exploited as all slaves can execute their ion trajectories simultaneously. The order of computing ion trajectories is arbitrary under the restriction that no damage accumulation occurs which is a very reasonable assumption. Both ions that enter through the surface as well as newly created particles can move inside of the simulation domain. The proposed master-slave parallelization method inherently models both cases without restriction. The master communicates with the slaves to inform them about a new time step of the simulation, and to maintain a record with the number of ions that reside in the geometry domain of every specific slave. The slaves communicate among each other by exchanging ions that cross the slave's geometry domain (see Figure 6). Currently, we use a static load balancing strategy based on information about the computational capabilities of each processor. For instance, if a processor P_i is twice as fast as another processor P_j then P_i gets a workload (number of ion trajectories) assigned that is twice as large than that of P_j.

The sequential version of the ion implantation simulator is composed of Fortran and C code components. Our parallel version of the ion implantation simulation exploits primarily coarse-grain parallelism and has been implemented by using MPI which took several months and was very error-prone.

Experimental results and further work

We ported our code onto a network of workstations (NOWs) as this architecture is well-suited to support coarse-grain parallelism and is also becoming increasingly popular due to the availability of unused computation cycles. We used a heterogeneous NOW consisting of several DEC alpha workstations including DEC 3000 (175 MHz), DEC 7000 (200 MHz), and DEC 600 (333 MHz) workstations. The DEC 600 workstations are connected by a 100 Mbits/sec Ethernet, and all others by a 10 Mbits/sec Ethernet. This NOW is a non-dedicated system where all workstations are office computers. We examined the performance of the parallel ion implantation simulator for two different problem sizes. 2000 (192 ions per time step) and 5000 (768 ions per time step) ions have been, respectively, distributed to the slaves considering the different computing capabilities of the various workstations. The execution times (see Figure 7) have been measured during regular office time which means that there has been computational load that is not related to the given application. The measurements for the problem size of 5000 ions has been done twice in order to demonstrate the impact of different workloads (unrelated to the given application) on the NOW. For the problem sizes 2000 and 5000 ions (1st run) we observe a maximum speedup of 1.6 and 2.3, respectively, for 5 slaves. Whereas, for the second run of the 5000 ions problem size, we achieve a very reasonable speedup of 5.3 for 6 slaves. Figure 8 shows the idle times (waiting for data from other slaves and the master) for each slave of every specific execution (number of

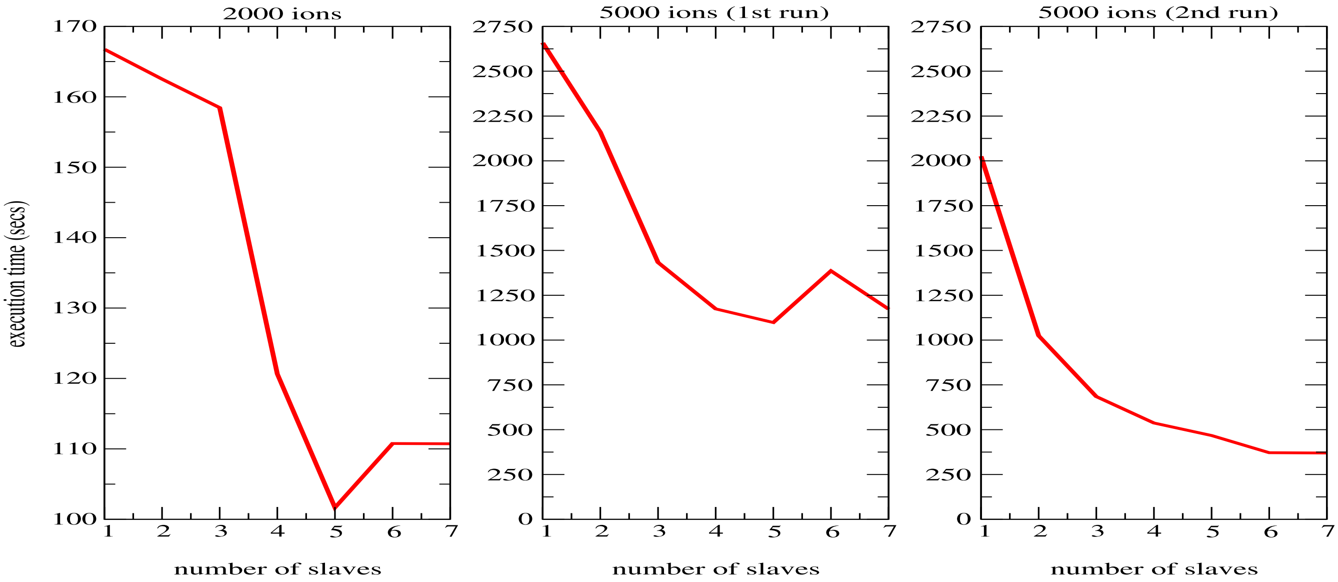

Figure 7. Execution times of the ion implantation simulator for different problem sizes and varying number of slave processors on a heterogeneous NOW

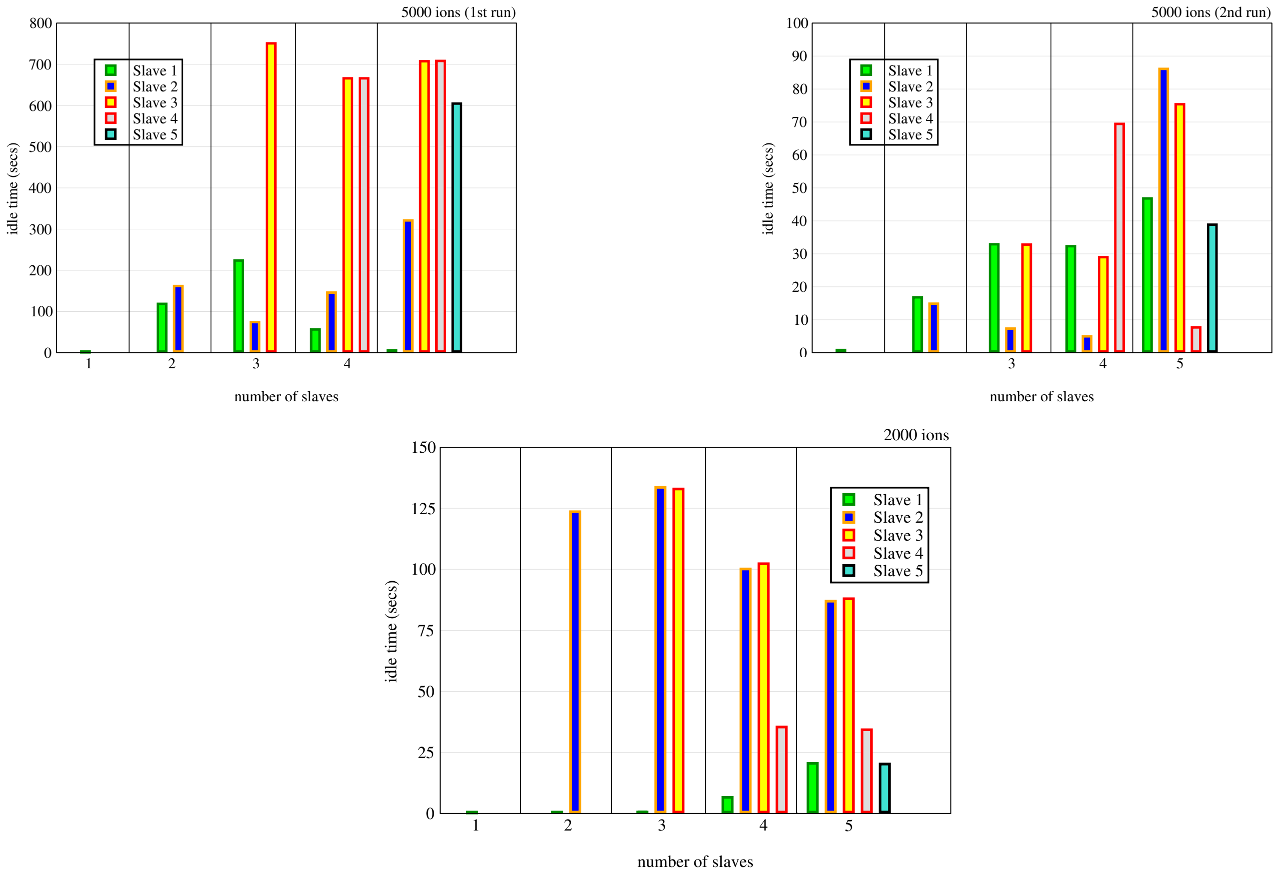

Figure 8. Idle times for the ion implantation simulator for two different problem sizes (2000 and 5000 ions) for varying number of slave nodes on a heterogeneous NOW.

slaves is fixed) of the ion implantation simulator. Clearly, we can observe that the idle times across different slaves can be quite different. This is due to uneven workload (unrelated to the application) on the NOW and also due to the degree of ions that move from the geometry domain of one slave to another. Note the strong difference in idle times for two different executions of identical number of slaves and problem sizes (5000 ions). The slaves of these problem sizes have been executed on the same workstations but with different unrelated workload.

The experiments clearly imply that the static load balancing should be replaced by a dynamic load balancing which is sensitive towards dynamically changing application and machine characteristics (for instance, moving ions and application unrelated workload).

8 Related Work

TAU [30] is a sophisticated instrumentation, tracing and profiling system that has been shown to be very useful for various programming paradigms including PC++ and HPC++ [25].

Forge90 [29] reports on communication costs at the level of a generated message passing code, but not at the level of the input program.

An approach for visualizing the performance for HPF programs is described in [26]. Various insights about the interplay between data mapping and communication for HPF programs are offered by this system.

In [1] the performance of Fortran D programs is analyzed at the source-level which is based on an integration with the Fortran D compiler [15] and the Pablo performance system [34]. MPP Apprentice [37] supports post-execution performance analysis for C, C++, and Fortran 90 programs on the Cray T3D machine. The previous two approaches are most similar to our approach. The Fortran D/Pablo integrated performance system has sophisticated capabilities to link performance data with distribution, alignment and mapping information for data parallel programs. It is unclear how accurate this system can record code transformations and optimizations which is a strength of Scala. Moreover, Scala collects more comprehensive information about arrays and can also describe the memory requirements for a given program. Apprentice maintains information about code restructuring for basic blocks. It reports time statistics for loops and for an entire application. Scala goes beyond basic blocks and can also record code changes that imply larger code sections than basic blocks. Scala can also deal with new code inserted in a code whose performance can be linked to a specific source of the input program.

9 Conclusions

There are many different ways to develop programs for distributed and parallel systems. Frequently users write programs at a very low-level (i.e., message passing programs) in order to fully exploit the computational capabilities of a target architecture. In recent years compilers provide extensive support to develop distributed and parallel programs by aggressively applying code optimizations to improve the resulting performance. This poses a serious problem for performance measurement and analysis systems. Performance data is frequently monitored at the level of a generated program or target machine without the possibility to map performance data back to the user provided program.

In this paper we describe Scala which is a novel framework that supports portable instrumentation, measurement and post-execution performance analysis for distributed and parallel systems. The most important features and contributions of our approach are as follows:

- portable instrumentation system that supports selective and comprehensive instrumentation of pre-defined types of code regions and arbitrary code regions.
- code restructuring information of transformation systems is recorded and collected in a measurement description file.
- large variety of information can be monitored: timing and execution frequency events, values for program unknowns, and array information
- comprehensive monitoring of arrays (shape, rank, distribution, mapping, alignment, local/global segments, etc.) is supported.
- many important performance metrics and statistics can be computed (i.e., speedup, efficiency, efficacy, communication and work distribution, compiler organization overhead, etc.)
- performance data can be related back to the user provided original input program based on the measurement description file.
- performance visualization is supported by several interfaces to our own and external performance visualization systems.

Scala is currently being used as a performance analysis system for explicit message passing programs and for programs generated by the VFC compiler [3] (translates HPF programs into message passing F90 programs based on MPI). We are porting the instrumentation system for distributed and parallel C programs. We also investigate the usefulness of Scala for performance analysis of JAVA/RMI programs. Moreover, we are in the process to integrate Scala with performance prediction [9] and symbolic analysis techniques [10] to examine the scaling behavior [35] of distributed and parallel programs.

References

[1] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. A. Reed. Integrating Compilation and Performance Analysis for Data Parallel Programs. . In M. Simmons, A. Hayes, D. Reed, and E. J. Brown, editors, *Proc. of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems, IEEE Computer Society Press*, January 1996.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Series in Computer Science. Addison Wesley, 1988.

[3] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1998.

[4] W. Bohmayr, A. Burenkov, J. Lorenz, H. Ryssel, and S. Selberherr. Trajectory split method for Monte Carlo simulation of ion implantation. *IEEE Transactions on Semiconductor Manufacturing*, 8(4):402–407, 1995.

[5] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A tool for workload characterization of parallel systems. *IEEE parallel and distributed technology: systems and applications*, 3(4):72–80, Winter 1995.

[6] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Integration of a compilation system and a performance tool: the hpf+ approach. In *Proc. of the International Conference on High-Performance Computing and Networking (HPCN'98), Amsterdam, The Netherlands*, pages 809–815. Lecture Notes in Computer Science, Springer Verlag, 1998.

[7] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Jan./Mar. 1998.

[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. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

[10] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing, Kluwer Academic Publishers*, 12(3):227–252, May 1998.

[11] T. Fahringer and E. Mehofer. Buffer-safe communication optimization based on data flow analysis and performance prediction. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 189–200, San Francisco, California, Nov. 10–14, 1997. IEEE Computer Society Press.

[12] T. Fahringer and E. Mehofer. Problem and Machine Sensitive Communication Optimization. In *Proc. of the 12th ACM International Conference on Supercomputing*, pages 121–128, Melbourne, Australia, July 1998. ACM Press.

[13] T. Fahringer and E. Mehofer. Buffer-Safe and Cost-Driven Communication Optimization. *Journal of Parallel and Distributed Computing, Academic Press*, 57(1):33–63, April 1999.

[14] M. P. I. Forum. *Document for a Standard Message Passing Interface*, draft edition, Nov. 1993.

[15] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[16] H. Gerndt. *Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

[17] Grace User's Guide V0.2. http://plasma-gate.weizmann.ac.il/Grace/doc/UsersGuide.html, March 1999.

[18] V. Herrarte and E. Lusk. Study parallel program behavior with Upshot. Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, Aug. 1991.

[19] High Performance FORTRAN Language Specification. Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.

[20] A. Hössinger, M. Radi, B. Scholz, T. Fahringer, E.Langer, and S. Selberherr. Parallelization of a Monte-Carlo Ion Implantation Simulator for Three-Dimensional Crystalline Structures. In *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD99), Springer*, Kyoto, Japan, Sept. 1999.

[21] A. Hössinger and S. Selberherr. Accurate Three-Dimensional Simulation of Damage Caused by Ion Implantation . In *Proc. 2nd Int. Conf. on Modeling and Simulation of Microsystems*, pages 363–366, April 1999.

[22] J. C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, April 1997.

[23] A. W. J. C. Hull. One factor interest rate models and the valuation of interest rate derivative securities. *Journal of Financial and Quantitative Analysis*, (28):235–254, 1993.

[24] S. Z. J.M. Hutchinson. Financial simulations on a massively parallallel connection machine. *The International Journal of Supercomputer Applications*, 5(2):27–45, 1991.

[25] E. Johnson, D. Gannon, and P. Beckman. HPC++: Experiments with the parallel standard template library. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 124–131, New York, July 7–11 1997. ACM Press.

[26] D. Kimelman, P. Mittal, E. Schonberg, P. F. Sweeney, K.-Y. Wang, and D. Zernik. Visualizing the execution of High Performance Fortran (HPF) programs. In IEEE, editor, *IPPS '95: 9th International parallel processing symposium — April 25–28, 1995, Santa Barbara, CA*, International Parallel Processing Symposium, pages 750–759, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[27] C. S. L. Clelow. *Implementing derivative Models*. John Wiley & Sons, 1998.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, july 1978.

[29] J. M. Levesque. FORGE90 and High Performance Fortran (HPF). In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 111–119. Springer-Verlag, 1993.

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

[31] H. Moritsch and E. Dockner. Numerical procedures for pricing interest rate dependent securities and their parallel implementations. Technical Report TR1999-?, Special Research Program SFB F011 AURORA, 1999.

[32] T. Nakata, Y. Kanoh, K. Tatsukawa, S. Yanagida, N. Nishi, and H. Takayama. Architecture and the Software Environment of Parallel Computer Cenju-4. *NEC Research and Development Journal*, 39:385–390, October 1998.

[33] P. G. P. Boyle, M. Broadie. Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, pages 1267–1321, 1997.

[34] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[35] X.-H. Sun, M. Pantano, and T. Fahringer. Integrated Range Comparison for Data-Parallel Compilation Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(5), May 1999.

[36] Sun Microsystems. Java RMI.

[37] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D, 1994.

[38] S. Zenios. *Parallel Monte Carlo simulation of mortgage-backed securities*. Cambridge University Press, 1993.