Development and Performance Analysis of Real-World Applications for Distributed and Parallel Architectures *

T. Fahringer† P. Blaha∗ A. Hössinger∗∗ J. Luitz∗ E. Mehofer† H. Moritsch‡ B. Scholz†

†Institute for Software Science, University of Vienna
Liechtensteinstrasse 22, A-1092, Vienna, Austria
[tf,mehofer,scholz]@par.univie.ac.at

‡Department of Business Administration, University of Vienna
Brünner Strasse 72, A-1210 Vienna, Austria
moritsch@finance2.bwl.univie.ac.at

∗Institute of Physical and Theoretical Chemistry, Vienna University of Technology
Getreidemarkt 9/156, A-1060 Vienna, Austria
[p.blaha,j.luitz]@tuwien.ac.at

∗∗Institute for Microelectronics, Vienna University of Technology
Gusshausstr. 27-29/E 360, A-1040 Vienna, Austria
hoessinger@iue.tuwien.ac.at

Accepted for publication in *Concurrency: Practice and Experience, John Wiley & Sons, to appear in 2001.*

Abstract

Several large real-world applications have been developed for distributed and parallel architectures. We examine two different program development approaches: First, the usage of a high-level programming paradigm which reduces the time to create a parallel program dramatically but sometimes at the cost of a reduced performance. A source-to-source compiler, has been employed to automatically compile programs – written in a high-level programming paradigm – into message passing codes. Second, manual program development by using a low-level programming paradigm – such as message passing – enables the programmer to fully exploit a given architecture at the cost of a time-consuming and error-prone effort.

Performance tools play a central role to support the performance-oriented development of applications for distributed and parallel architectures. SCALA – a portable instrumentation, measurement, and post-execution performance analysis system for distributed and parallel programs – has been used to analyse and to guide the application development by selectively instrumenting and measuring the code versions, by comparing performance information of several program executions, by computing a variety of important performance metrics, by detecting performance bottlenecks, and by relating performance information back to the input program. We show several experiments of SCALA when applied to real-world applications. These experiments are conducted for a NEC Cenju-4 distributed memory machine and a cluster of heterogeneous workstations and networks.

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

1 Introduction

Performance-oriented 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 analysing. Many different programming paradigms such as explicit message passing [20], High Performance Fortran (HPF) [26], OpenMP [11], Java RMI [44], and HPC++ [30] have been introduced for distributed and parallel architectures. A trade-off is implied by the programming paradigm employed. On the one hand, programming at a low level (i.e., message passing paradigm) enables the programmer to fully exploit and to control the features of a specific architecture at the cost of a very time-consuming and error-prone programming effort. On the other hand, choosing a high-level programming paradigm can reduce the program development effort dramatically, however, sometimes at the cost of a reduced performance. There is a large variety of reasons that can cause performance losses in distributed or parallel programs. For instance, ineffective data and work parallelism, uneven load balance, compiler organization overhead, ineffective memory access behavior (i.e., cache), and high communication, synchronization and input/output overhead. The source for these performance bottlenecks can frequently be related to the intricate structure and details of an application program, to the code transformation system, or to the target architecture.

Performance tools play a crucial role to support the performance-oriented development of applications for distributed and parallel architectures by locating performance problems and mapping them back to the input program. Many existing performance monitoring and analysis systems collect and present performance data for programs that have been generated and modified by transformation systems without the possibility to relate performance data back to the input program. A performance system must have access to transformation systems in order to record code changes and associate performance problems with the input program. Commonly, performance information is provided for low-level system calls (i.e., operating and runtime library calls) that cannot be mapped to specific locations in the input program. There are performance tools that provide only summary information for entire programs without relation to specific program points or regions of interest. Crucial correlation of performance bottlenecks with exact positions in the input program is disabled which severely restricts the usefulness of such tools. Finally, many performance tools are restricted to a specific programming paradigm.

SCALA [19, 43, 41, 42] 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. SCALA can be used to monitor and analyze the performance of many different programming paradigms ranging from high-level (i.e. High Performance Fortran) to low-level programs including data parallel, task parallel and message passing programs. 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. SCALA supports several trace formats which allows the use of various visualization systems (e.g. Medea [7], and Upshot [23]). Finally, SCALA has highly sophisticated scalability analysis integrated, that examines the scaling behavior of a program for varying input data and machine sizes.

This paper describes the performance-oriented development of real-world applications for distributed and parallel architectures. SCALA has been used to analyse and to guide the application development by selectively instrumenting and measuring the code versions, by comparing performance information of several program executions, by computing a variety of important performance metrics, by detecting performance bottlenecks, and by relating performance information back to the input program. We used the following three applications: (1) a Monte-Carlo ion implantation simulator for three-dimensional crystalline structures [25, 5] developed by Prof. Selberherr and his associates at the Vienna University of Technology; (2) a system for pricing of financial derivatives [12] developed by Prof. Dockners' group at the University of Vienna; and (3) WIEN97 [3], a system for quantum mechanical calculations of solids developed by Prof. Schwarz and his group at the Vienna University of Technology.

The ion implantation 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. The pricing system for financial derivatives has been developed from scratch as an HPF/Fortran90 program that uses data parallelism. WIEN97 has been parallelized by employing HPF/Fortran77 which benefits from data parallelism. Whereas the pricing system has been developed from scratch with the aim to carefully uncover and exploit parallelism, the other two applications have been parallelized based on existing sequential codes. The pricing system and WIEN97 have been parallelized by using VFC [2] – a compiler that translates HPF programs into message passing programs (Fortran90 with MPI message passing calls) – and executed on a NEC Cenju-4 [38] distributed

memory parallel machine.

In the next Section, we give an overview of SCALA as an integrated system of a restructuring and optimizing compiler. Sections 3.1 to 3.3 describe the three applications and show how they have been parallelized for distributed and parallel architectures. We demonstrate the usefulness of SCALA to instrument, monitor, and analyze the performance for the application codes. Experiments are shown for a NEC Cenju-4 distributed memory machine and a cluster of heterogeneous workstations and networks. Related work is discussed in Section 4. Summary and concluding remarks are are given in Section 5.

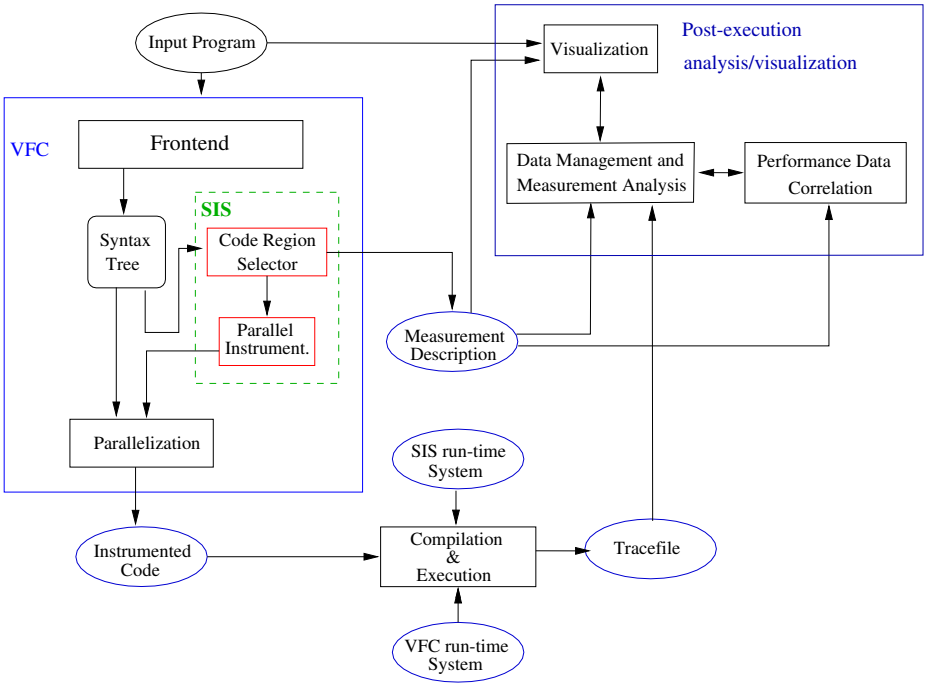

Figure 1. Execution-Driven Performance Analysis System

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 several interfaces to 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 [2] which is a compiler that translates Fortran programs into message passing programs (Fortran90 with MPI message passing calls). The input programs (Fortran77, Fortran90, 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). Then the compiler 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, communication, and work distribution. Several interfaces for visualization systems have been developed 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 what follows we give a brief overview of each of these modules. A detailed description of SCALA and its functionality can be found in [19, 43, 41, 42]

2.1 SCALA Instrumentation System

Based on user-provided command-line options or directives, 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.), and array information (rank, shape, alignment, distribution, mapping, etc.). SIS supports the programmer to control monitoring and generating performance data through selective instrumentation of specific types of code regions (i.e., program, procedures, loops, communication, and I/O operations). SIS also enables instrumentation of arbitrary code regions through explicit instrumentation of all entry and exit points of 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$ SIS_BEGIN_MEASURE reference

      code region

!sis$ SIS_END_MEASURE reference
```

The directive SIS_BEGIN_MEASURE must be inserted by the programmer 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 by the programmer in each entry and exit point of a given code region.

Furthermore, SIS provides specific directives in order to control monitoring. The directives SIS_TRACE_ENABLE and SIS_TRACE_DISABLE enable 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 the pricing code discussed in Section 3.1, where for the sake of demonstration, the call to function *RANDOM_PATH* is not measured by using the facilities to control monitoring as mentioned above.

```
    !sis$ SIS_BEGIN_MEASURE 10
          !HPF$ INDEPENDENT, NEW(PATH), ON HOME(VALUE(I))
          DO I = 1, N
    !sis$ SIS_TRACE_DISABLE
```

```
          PATH = RANDOM_PATH(0,0,N)
!sis$ SIS_TRACE_ENABLE
          VALUE(I) = DISCOUNT(0,CASH_FLOW(B,1,N),FACTORS_AT(PATH))
      END DO
      PRICE = SUM(VALUE)/N
!sis$ SIS_END_MEASURE 10
```

Note that directives must be inserted by the programmer based on which SCALA automatically instruments the code.

2.2 Data management and measurement analysis

Tracing the performance data can lead to a 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 the application 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 management and measurement analysis* module (see Figure 1) implements several data-reduction techniques. 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. The raw performance data (for instance, timing and frequency events) is summarized from aggregate (i.e., entire program) through procedure, procedure calls, loops, arbitrary code sections, and individual source code lines. The programmer thus can concentrate performance analysis on the most important code sections. For instance, the upper *Profiling Visualization Window* in Figure 5 displays timing information only for a specific procedure call highlighted in the *Source Code Profiling Window*. 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 application performance, while reducing the amount of data.

The data management and measurement analysis module computes a variety of performance metrics which includes speedup, efficiency, execution signatures, and communication and work distribution. For instance, the *Metric Visualization Windows* in Figure 5 show the execution (time) signature and speedup for varying number of processors with respect to an entire parallel program. 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 "goodness" of work and communication distributions across processors. As an example see Figure 14, where the Grace graphical user interface of SCALA shows uneven distribution of idle times for different number of slave processors which indicates a severe load balancing problem. In many cases, the data measured need to be scaled in a common interval so that further statistical techniques can be successfully applied. Timing indices also can be scaled in a more significant metric for the analysis such as from seconds to microseconds or nanoseconds.

A crucial aspect is to compare different code versions with respect to specific performance metrics. Programmers frequently determine a performance problem, for instance idle times or communication overhead. After applying appropriate code changes the performance outcome of the code before and after the code change should be comparable which is supported by the data management and measurement analysis module of SCALA. Performance data is stored together with the associated code version. Performance metrics can be computed for different code versions and displayed together so that the programmer can easily interpret the impact of code changes. For instance, Figure 9 shows the execution time behavior for two different code versions for different problem sizes in the same SCALA visualization window.

2.3 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 [18]. 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 [2] which causes a loop to be transformed into a preparation (inspector) and an execution (executor) phase. Moreover, for specific procedure calls, a compiler may imply additional overhead such as distribution of data before and after the call. In order to examine which performance aspect corresponds to what code region, SCALA generates a *measurement description file* which is updated while the compiler is applying code transformations. The data correlation module is invoked by the data management and measurement analysis module to relate performance metrics to a given code region which is the visualized by the performance visualization module. Figure 5 shows part of the application source code in the *Source Code Profiling Window*. The user can click on a specific statement, for instance, *call traverse_discount* for which the upper most *Profiling Visualization Window* of SCALA (based on Medea graphical user interface – see Section 2.4) displays the compiler overhead before (head overhead) and after (tail overhead) the call. Moreover, the executor overhead for the entire call is presented.

2.4 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 several different trace formats – including ALOG and Grace formats – for collected performance data which enables the usage of various performance visualization systems.

Based on the ALOG trace format we can use well-established visualization systems such as Medea [7], TAU [36], and Upshot [23]. 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. This work has been described in detail in [8]. Figure 5 shows Medea visualizing various performance metrics for one of our experimental codes (see Section 3.1).

SCALA also generates Grace [22] data files for various 2D performance data visualizations and for comparing different program versions and their performance outcome as seen in Figures 4, 9, 13, and 14. Note that all performance analysis is conducted by SCALA whereas Grace is solely used for visualization of performance data.

SCALA's graphical user interface is connected with the execution-driven performance analysis system as shown in Figure 1 through a communication layer implemented as a Java RMI distributed middle-ware (supports Java JDK 2.0). The execution-driven performance analysis system and SCALA's 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 measurement description files, performance data provided by the data management and measurement analysis module, and the input program without having full access to VFC and SIS.

3 Experiments

The functionality of SCALA as described in Section 2 has been implemented and runs under most Unix systems. In this section we present three 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 [12] 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 [38] distributed memory parallel machine.

The second experiment employs SCALA to examine the performance for two different HPF parallelization strategies of a material science code called WIEN97 [4]. This code has been developed by Prof. Schwarz's group at the Vienna University of Technolgy. The sequential code originally existed and has been extended by HPF directives which exploits data parallelism and was executed on a NEC Cenju-4 distributed memory parallel machine.

In the third experiment we used SCALA to analyze the performance of a Monte-Carlo ion implantation simulator for three-dimensional crystalline structures [25, 5] as developed by Prof. Selberherr and his associates at the Vienna University of Technology. The ion implantation 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 data and task parallelism and runs on a cluster of heterogeneous workstations and networks.

3.1 Pricing of Financial Derivatives

The pricing of derivate and interest rate dependent products is an important field in finance. A *derivative* (or *derivative security*) is a financial instrument whose value depends on other, so called underlying securities (e.g. stock options) [27]. We concentrate on the pricing of interest rate dependent products, whose payments depend on actual or past interest rates

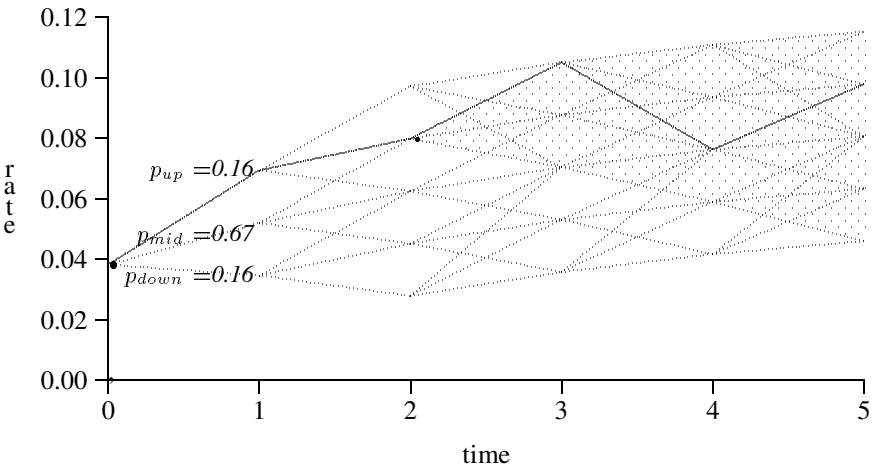

Figure 2. Hull and White tree for the Δt spotrate with selected path

(e.g. variable coupon bonds). 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 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 [6],[10]. By utilizing massively parallel architectures very efficient implementations can be achieved [29],[46]. For a detailed description of the technique implemented see [37].

```
...
!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)
      VALUE(I) = DISCOUNT(0,CASH_FLOW(B,1,N),FACTORS_AT(PATH))  ! discount the bond's cash flow to time 0
    END DO
    PRICE = SUM(VALUE)/N                                     ! mean value
...
```

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

The Monte Carlo simulation is based on a discrete representation of a stochastic process that describes the dynamics of interest rates over time [28]. The *Hull and White tree* describes the future development of the short term interest rate, which is a state variable used to calculate the interest rates for different maturities for a specific state of the system [27]. Each state is represented by a node in a directed graph and has three successor nodes, representing increasing, constant, and decreasing interest rates. Nodes are described by (time, interest rate) pairs. Arcs are labeled with the transition probabilities $p_{up}, p_{mid}, p_{down}$. A state can be reached by more than one predecessor; this *recombining* property establishes a lattice structure. Figure 2 shows a Hull and White tree with time (i.e. in years) on the horizontal axis and interest rates on the vertical axis.

To price interest rate dependent products the interest rate tree is used either to solve it backwards in time or by simulating paths through the tree and averaging the corresponding prices. The Monte Carlo Simulation algorithm selects a number N of paths in the Hull and White tree from the root node to some final node (see Figure 2). Along each path, it iteratively discounts, backwards from the final node to the root node, the cash flow generated by the instrument along this path. For

variable coupon bonds, the cash flows are path dependent, i.e. depend on the interest rates at predecessor nodes. Discounting is performed using the interest rates along this path. The resulting price of the instrument is the mean value over all selected paths. The HPF/Fortran90 code segment in Figure 3 shows the main loop of the simulation procedure TRAVERSE_DISCOUNT .

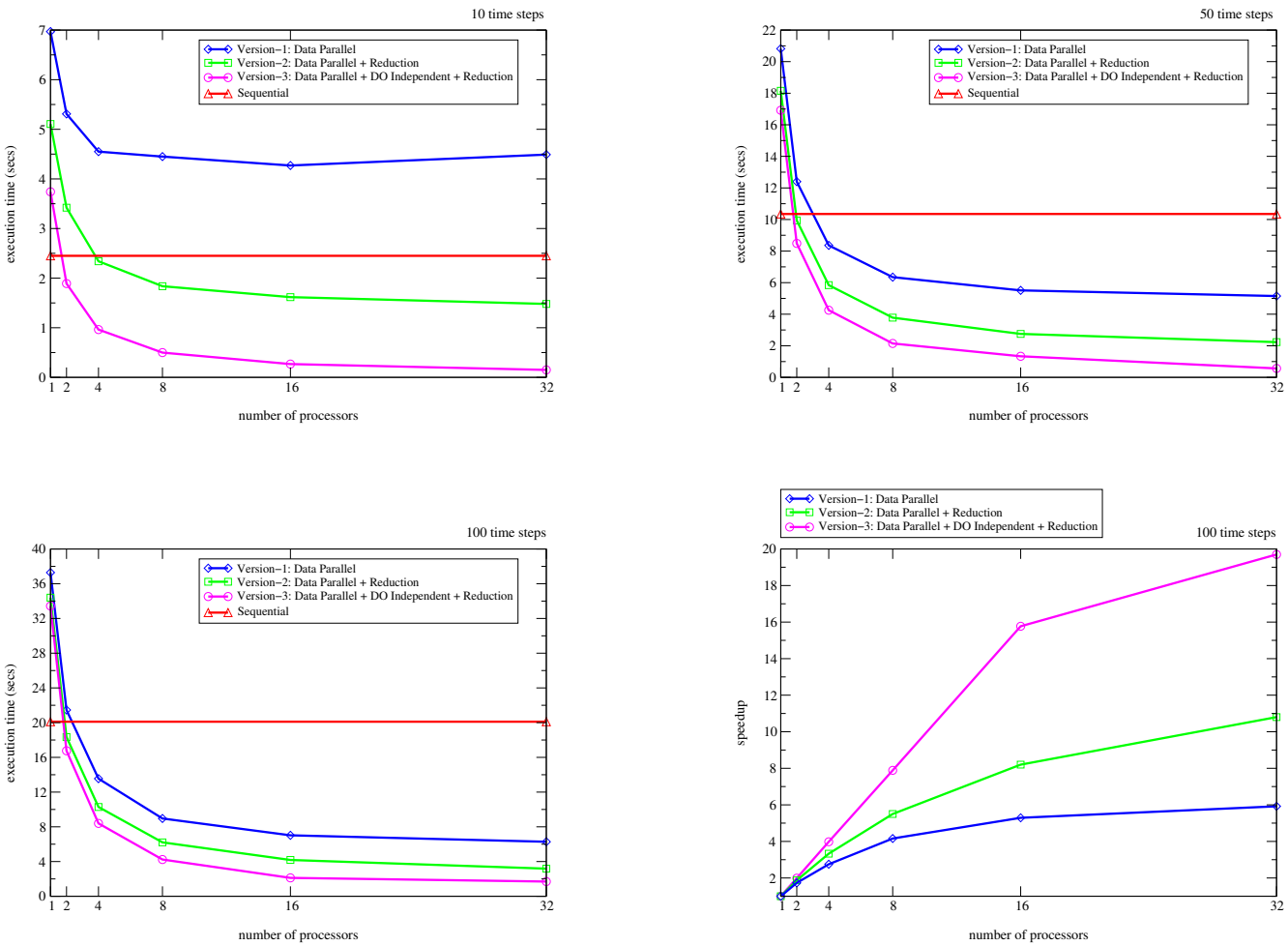

Figure 4. Accumulated execution times are measured and visualized by SCALA for all three versions of function TRAVERSE_DISCOUNT of the system for pricing of financial derivatives (three HPF and one sequential version). Speedups are for the problem size of 100 time steps only. Experiments have been conducted by using SCALA for three different problem sizes (number of time steps) and varying number of processors on a NEC Cenju-4

We extend the procedure in order to price also bonds with embedded options (callable or putable bonds, see [13]). The holder of a callable bond has given the issuer the right to redeem the bond before its maturity date. The issuer will redeem (and pay the principal), when the present value of future cash flows is greater than a specified "exercise" value. In the case of a putable bond, the holder will sell the bond to the issuer, when the present value of future cash flows is less than the exercise value.

We model the effect of redemption through a modification of the cash flows: the cash flow at redemption time is set to the principal payment value, and the cash flows after that time are set to zero. For the computation of the decision, whether redemption takes place at a node k or not, we perform a nested Monte Carlo simulation, which samples paths in the "subtree" defined by node k. We end up with two levels of simulation [37]. At the first level, paths starting at the root node are processed. At the second level, for each node in such a path, "subpaths" emanating from this node are selected. Discounting along the subpaths is performed to compute the redemption decision at node k.

At both levels the same recursive simulation procedure TRAVERSE_DISCOUNT is used. During the simulation at the first level, it calls an extended DISCOUNT function, which again invokes TRAVERSE_DISCOUNT to perform the nested simulation as shown in Figure 3. At the second level, redemption is handled without further recursion. More details about this application on pricing of financial derivatives can be found in [12].

3.1.1 Parallelization

During the simulation, the information at the tree nodes is potentially used the by the computation of every path. This motivates a replication of the whole tree over all processors. The storage requirements for these structures are comparatively small and not critical in terms of local memory size.

Sampling as well discounting along the paths can be done in parallel. Because all the path computations are independent from each other, they can be performed without communication. Every path computation has access to the whole tree data. After processing the individual paths, the final price is computed via a summation of the path results over all processors. A reduction operation is used, which first computes partial sums on each processor simultanously, and then sends the partial results to a selected processor which computes the final sum. This is the only operation which requires communication.

We encoded three different HPF versions of the pricing system and executed them on a NEC Cenju-4 [38] distributed memory parallel machine. First, a data parallel version was developed 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 and as a consequence deteriorates the scalability behavior of this version. SCALA has been used to instrument and measure the communication of this code. In order to reduce the communication, we created a second code version that uses the HPF reduction directive, which causes the summation of the path results to be executed by an efficient machine function. SCALA determined that the second code still does not sufficiently scale with increasing number of processors. This led to the development of a third code version by using the HPF DO-INDEPENDENT directive which 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 [2].

The development of the sequential pricing code took several months. Only a few HPF directives needed to be inserted in order to provide a parallel code that could be effectively parallelized by VFC. Actual parallelization with VFC, performance analysis, and performance tuning by using SCALA took only a few days.

3.1.2 Experimental results and further work

VFC has been used to generate Fortran90/MPI programs based on input HPF programs. SCALA has been employed to find the best code version with respect to performance out of the three code versions as described above. The code versions have been instrumented and measured on the NEC Cenju-4 machine for different machine sizes by using SCALA. The data management and measurement analysis module of SCALA examines the tracefiles, and computes the execution times for every code version including the sequential program which are then plotted in graphs shown in Figure 4. Note that these graphs are based on SCALA's interface to the Grace visualization system as described in Section 2.4.

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 performance plots as derived by SCALA 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. SCALA has been used to examine the execution behavior of all described 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 speedup (see Figure 4) for up to 16 processors based on a problem size of 100 time steps. Based on the experiments conducted with SCALA, we believe that larger problem and machine sizes should cause a better performance scaling behavior.

Figure 5 shows several snapshots of the MEDEA system which is used as one of several graphical user interfaces of SCALA (see Section 2.4). Various performance metrics are displayed together with the input program. Note that all performance measurements have been obtained by SCALA based on the generated Fortran90/MPI program, whereas the perfor-

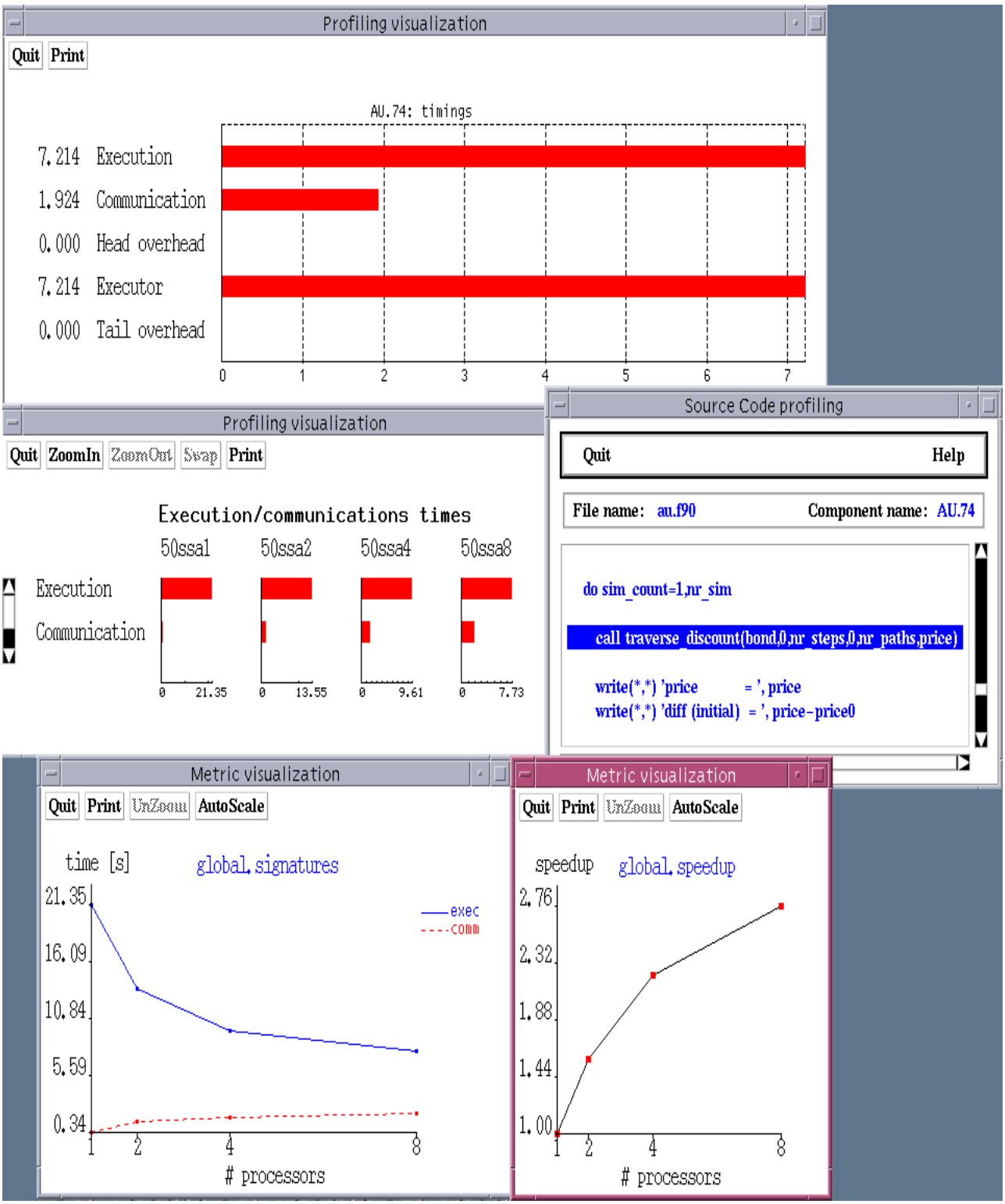

Figure 5. Snapshots of the MEDEA system which displays various performance metrics together with the code section of interest (call to function *TRAVERSE_DISCOUNT* in middle-right window).

mance 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 processors that need this data. This procedure implies some extra communication, however, may save substantial computation time.

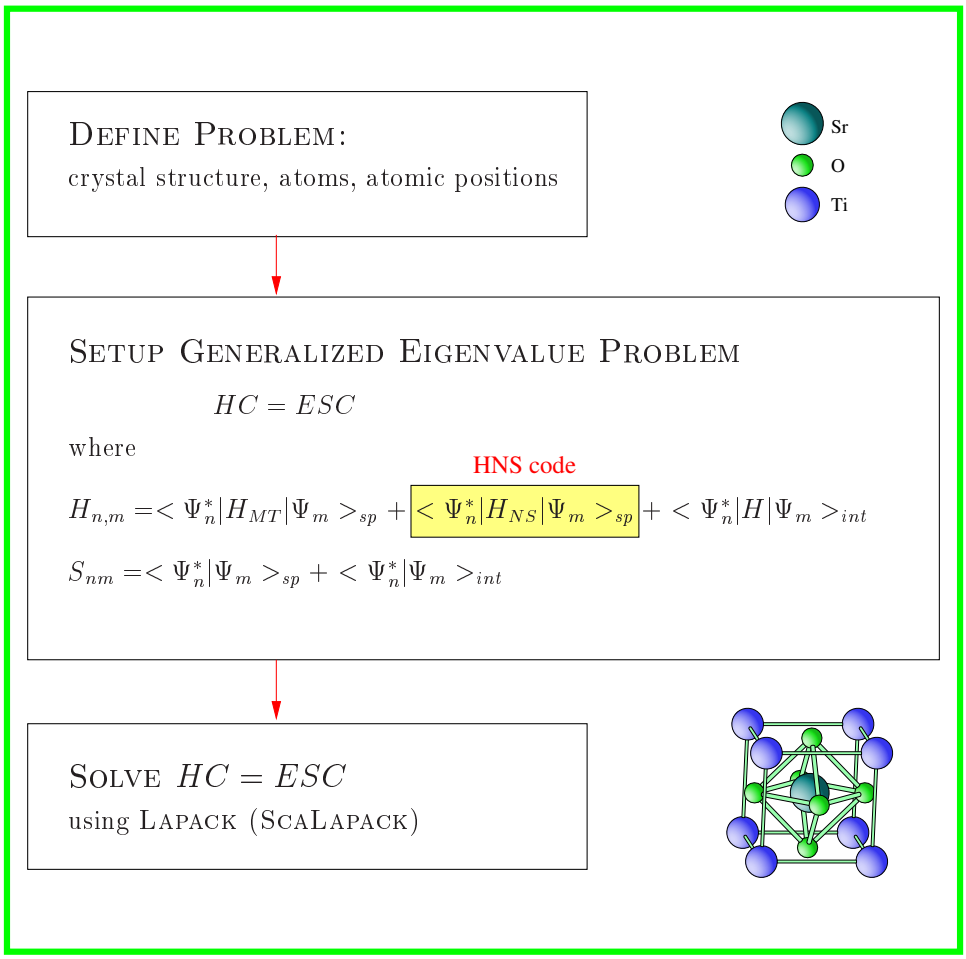

Figure 6. Computation of a crystal structure using WIEN97

3.2 Quantum Mechanical Calculations of Solids

During the last 16 years a program package called WIEN97 [4] has been developed and is used worldwide by more than 280 research groups. It is based on density functional theory, for which Walter Kohn received the Nobel prize for chemistry in 1998, and the LAPW method [40] which is one of the most accurate methods to investigate theoretically the properties of high technology materials. Applications to the new high temperature superconductors, magnetic structures (for magnetic

recording), surfaces (catalysis) or intercalation compounds (new Li batteries) require a reliable computer code that can run even for weeks on a single CPU to produce final results. For this reason parallel computing is highly desirable.

WIEN97 calculates the electronic structure of solids. Figure 6 describes the principle tasks of such a calculation: After the definition of the problem, a *generalized eigenvalue problem* must first be setup and then solved iteratively (i.e. many times) leading to energies (eigenvalues, E) and the corresponding coefficients (eigenvectors, C). The size (N) of the corresponding Hamilton (H) and Overlap (S) matrices is related to the accuracy of the calculation and thus to the number of plane wave (PW) basis functions. About 50 - 100 PWs are needed per atom in the unit cell. For systems containing 50 up to 100 atoms per unit cell matrices of the size 2500 to 10000 must be handled.

The most CPU intensive part of WIEN is the solution of the generalized eigenvalue problem which at present is solved using modified LAPACK (or ScalaPack in parallel) routines. The second most important step is setting up the matrix elements of H and S, which are complicated sums of various terms (integrals between basis functions). A large fraction of this time is spent in the subroutine HNS, where the contributions to H due to the nonspherical potential are calculated.

In HNS radial and angular dependent contributions to these elements are precomputed and condensed in a number of vectors which are then applied in a series of rank-2 updates to the symmetric (hermitian) Hamilton matrix. HNS has 17 one-, 14 two-, 5 three-, and 6 four-dimensional arrays. The computational complexity of HNS is of the order $O(N^2)$. All floating point operations are done in double (eight bytes) precision.

```
...
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(*,CYCLIC) ONTO PR :: H
...
DO 60 I = 1, N
    !HPF$ INDEPENDENT, ON HOME (H(:,J))
    DO 70 J = 1, I
        H(I,J) = H(I,J) + A1R(1,J)*A2R(1,I)
        H(I,J) = H(I,J) - A1I(1,J)*A2I(1,I)
        H(I,J) = H(I,J) + B1R(1,J)*B2R(1,I)
        H(I,J) = H(I,J) - B1I(1,J)*B2I(1,I)
70    CONTINUE
60 CONTINUE
...
DO 260 I = N+1, N+NLO
    !HPF$ INDEPENDENT, ON HOME (H(:,J))
    DO 270 J = 1, I
        H(I,J) = H(I,J) + A1R(1,J)*A2R(1,I)
        H(I,J) = H(I,J) - A1I(1,J)*A2I(1,I)
        H(I,J) = H(I,J) + B1R(1,J)*B2R(1,I)
        H(I,J) = H(I,J) - B1I(1,J)*B2I(1,I)
        H(I,J) = H(I,J) + C1R(1,J)*C2R(1,I)
        H(I,J) = H(I,J) - C1I(1,J)*C2I(1,I)
270   CONTINUE
260 CONTINUE
...
```

Figure 7. HNS based on HPF DO-INDEPENDENT

```
...
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(*,CYCLIC) ONTO PR :: H
...
DO 60 I = 1, N
    H(I,1:I) = H(I,1:I) + A1R(1,1:I)*A2R(1,I)
    H(I,1:I) = H(I,1:I) - A1I(1,1:I)*A2I(1,I)
    H(I,1:I) = H(I,1:I) + B1R(1,1:I)*B2R(1,I)
    H(I,1:I) = H(I,1:I) - B1I(1,1:I)*B2I(1,I)
60 CONTINUE
...
DO 260 I = N+1, N+NLO
    H(I,1:I) = H(I,1:I) + A1R(1,1:I)*A2R(1,I)
    H(I,1:I) = H(I,1:I) - A1I(1,1:I)*A2I(1,I)
    H(I,1:I) = H(I,1:I) + B1R(1,1:I)*B2R(1,I)
    H(I,1:I) = H(I,1:I) - B1I(1,1:I)*B2I(1,I)
    H(I,1:I) = H(I,1:I) + C1R(1,1:I)*C2R(1,I)
    H(I,1:I) = H(I,1:I) - C1I(1,1:I)*C2I(1,I)
260 CONTINUE
...
```

Figure 8. HNS based on HPF/Fortran90 array operations

3.2.1 Parallelization

We have created two different HPF versions of the HNS code by using VFC. In both versions H, the main HNS array, has been distributed CYCLIC [26] in the second dimension onto the maximum number of processors (HPF intrinsic function NUMBER_OF_PROCESSORS) – that are available on a given architecture. We choose this distribution due to the triangular access pattern of array H (see Figure 7) which favors CYCLIC over BLOCK distribution. In the first HNS version (see Figure 7), we use the HPF DO-INDEPENDENT directive to indicate that the iterations of DO-loops 70 and 270 can be executed simultaneously. This version is solely based on Fortran77. In accordance with the owner-computes paradigm [2] an iteration is executed by the processor that owns array section H(:,J). The second code version is based on executing Fortran90 array operations [34] inside of DO-loops 60 and 260. The array operations are executed in parallel based on the owner-computes-paradigm and the HPF distribution directives. Note that both code versions have identical semantics. They differ only in their parallelization strategy. It should also be stated that detecting parallelism and inserting HPF directives took less than 1/2 day for both parallel versions of the HNS code. Performance analysis added another day which included the time for instrumentation, measurement, and performance analysis by using SCALA.

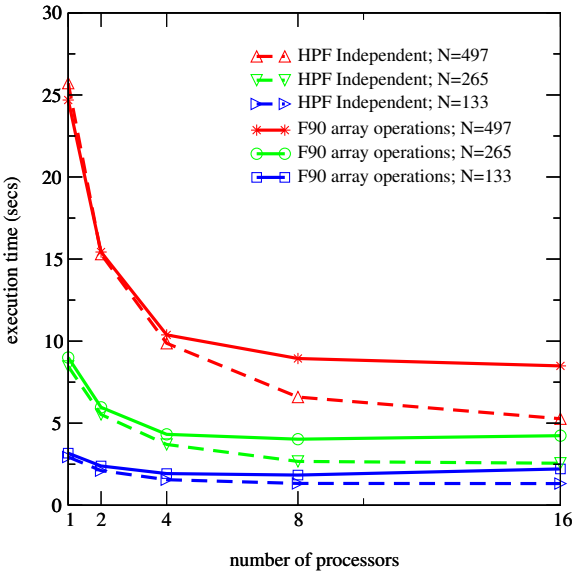

Figure 9. Execution times measured and visualized by SCALA for two different parallel versions of the WIEN97 HNS code (HPF DO-INDEPENDENT and HPF/Fortran90 array operations) for varying processors and problem sizes (N) on a NEC Cenju 4.

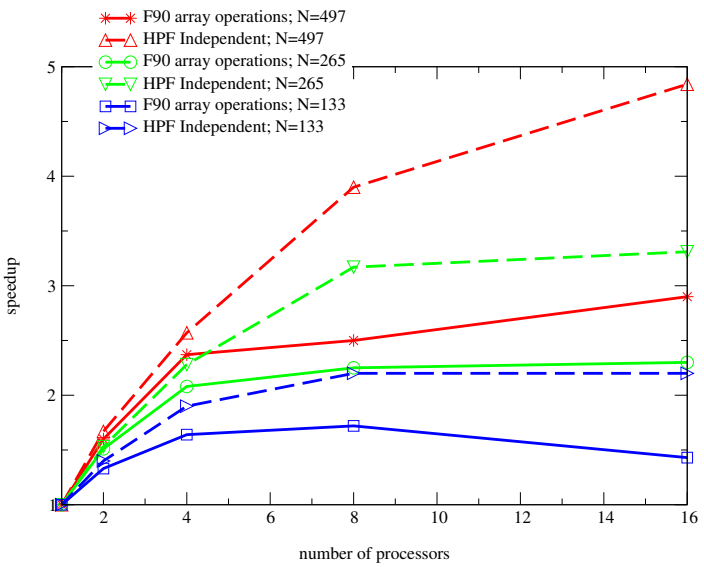

Figure 10. Speedups measured and visualized by SCALA for two different parallel versions of the WIEN97 HNS code (HPF DO-INDEPENDENT and HPF/Fortran90 array operations) for varying processors and problem sizes (N) on a NEC Cenju 4.

3.2.2 Experimental results and further work

SCALA has been used to instrument and measure both code versions and to plot the corresponding performance metrics in order to determine the best version for various problem (N - controls size of array H) and machine sizes (number of processors). Figure 9 shows the execution times of both versions on a NEC Cenju-4 machine based on the Grace graphical user interface of SCALA. SCALA clearly shows that the HPF DO-INDEPENDENT version outperforms the HPF/Fortran90 version for all problem and machine sizes. VFC generates more efficient code for the HPF DO-INDEPENDENT version by determining the work distribution outside of the innermost loops. Moreover, the array subscript expressions and loop bounds are only changed very little. Whereas for the second version, VFC uses ADLIB [9] to parallelize the Fortran90 array operations which requires changing array subscript expressions and loop bounds more extensively and the overhead for computing the work distribution is larger than for the HPF DO-INDEPENDENT version. The speedup metric (see Figure 10) provided by SCALA also demonstrate that the performance scales better for larger than for smaller problem sizes. For instance, the speedup achieved for a 8 processor HPF DO-INDEPENDENT version is 3.8 for N=497 and 2.2 for N=133. Similarly, the speedup achieved for a 8 processor HPF/Fortran90 is 2.5 for N=497 and 1.7 for N=133.

In the current work only the main loop of HNS was parallelized. The initialization part, which consumes approximately 15 % of the overall execution time (as determined by experiments with SCALA), will also be implemented. We will examine various data distributions for the diagonalization routines. SCALA will be used to evaluate the corresponding execution and communication time behavior. Thereafter, we plan to parallelize the setup phase of the spherical part H_{MT}. Overall we are very confident that HPF has the potential to parallelize large and substantial portions of the WIEN97 application.

3.3 Ion Implantation Simulator for Three-Dimensional Crystalline Structures

In modern semiconductor process technology, ion implantation is the most important technique to introduce dopants (atom species like boron, phosphorus or arsenic) into semiconductor materials like silicon or gallium-arsenide. Ion implantation means that accelerated ionized atoms are shot at a semiconductor material and penetrate into the solid as a consequence of their high kinetic energy. The introduced dopants are used to selectively set the resistivity of the semi-conducting material and to form diodes or transistors or other devices in the semi-conducting material. There are various methods for the simulation of ion implantation, but the high accuracy which is a prerequisite when simulating modern semiconductor production processes

often requires the application of the Monte Carlo method, which is a physically based simulation method. Thereby non-planarity effects and phenomena resulting from ion channelling and large tilt angles can be accurately described. By the Monte Carlo method the trajectories of the implanted particles in the target material are evaluated by calculating the interaction of the fast moving ion with the electrons and the nuclei of the target material. The method is schematically illustrated in Figure 11. The implanted particle moves through the target and changes its direction of motion by interaction processes with the target material and it successively looses energy until come to rest inside the target material. The major drawback of the Monte Carlo method is that it requires large simulation times, especially if three-dimensional simulations have to be performed with very sophisticated models [25] to reach the expected accuracy. Simulations can take up several days and even weeks for realistic problem sizes which makes it a first-order target for parallelization. Because the serial code for the ion implantation simulator existed we followed a parallelization strategy that modified the original code and computation models [24, 25, 5] as little as possible by isolating communication and synchronization in a few routines.

As already mentioned the Monte Carlo ion implantation simulation method is based on the concept that the trajectory of an ion is calculated (Figure 11). During simulation the trajectories of a large number of ions entering the device structure – equally distributed over the device surface – are computed. As a result of the simulation the distribution of the dopants and the crystal damage are derived from the final final position of the ions and the displaced target atoms. As an extension of the serial code a transient simulation had to be introduced for the parallelization of the simulator to correctly consider the damage accumulation and the influence of the damage on the ion trajectories. Transient simulation means that time steps are defined by assuming that ions belonging to the same time step do not interact with each other and can be treated independently. Furthermore it is assumed that the ions belonging to one time step are equally distributed over the device surface.

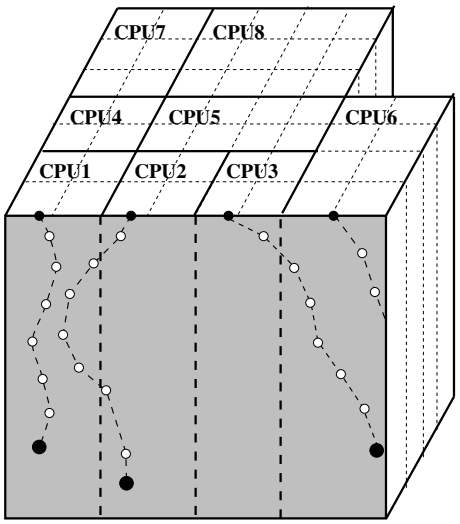

Figure 11. Schematic presentation of the calculation of the ion tracejctories and the distribution of the geometry among several processors.

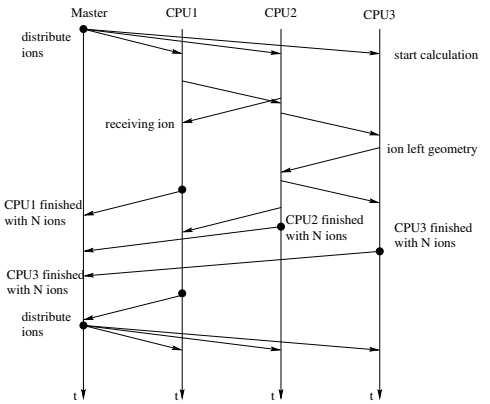

Figure 12. Communication among master and slaves (CPU1 - CPU3); t is the time-axis.

3.3.1 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 blocks. At the beginning of each time step the master processor distributes one or several rectangular blocks to a set of slave processors as shown in Figure 11. Each processor is responsible to calculate the trajectories of all ions residing inside of its assigned rectangular blocks. 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 does not matter in practice. Both ions that enter through the surface as well as newly created particles (due to collisions of mobile particles with atoms of the target) can move inside of the simulation domain. The proposed master-slave parallelization method inherently models both cases without restriction.

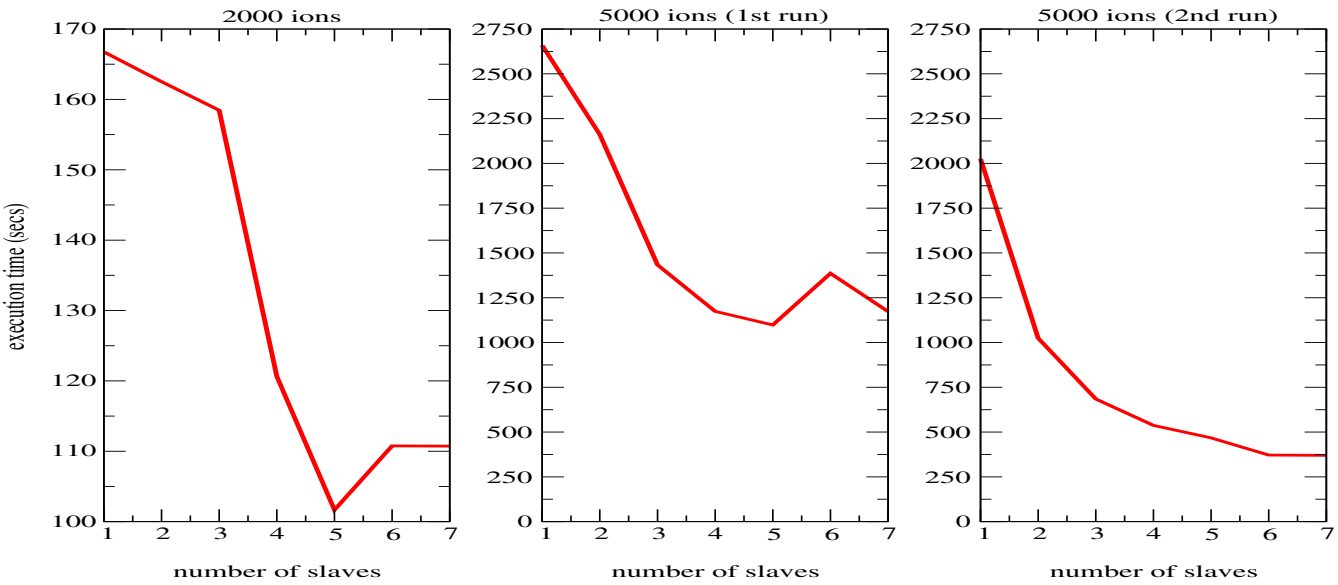

Figure 13. Execution times of the ion implantation simulator measured and visualized by SCALA for different problem sizes and varying number of slave processors on a heterogeneous NOW

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 Figures 11 and 12). 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. In order to optimize the performance of our master-slave method based on static load balancing which is executed on a dedicated distributed or parallel architecture (exclusively used by our application), the following two constraints should be considered:

$$\sum_i \frac{O_i}{V_i} \rightarrow min \tag{1}$$

$$\frac{V_i}{CPU_i} \simeq const., \forall i \tag{2}$$

V_i, O_i, and CPU_i (e.g., floating point operations per second) are, respectively, the volume, the surface of the rectangular area, and the relative computing capability of a slave i.

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 (message passing interface [20]) which took several months and was very error-prone.

The experiments (see Figures 13 and 14) conducted by using SCALA's instrumentation, performance analysis and visualization functionalities, 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).

3.3.2 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. SCALA has

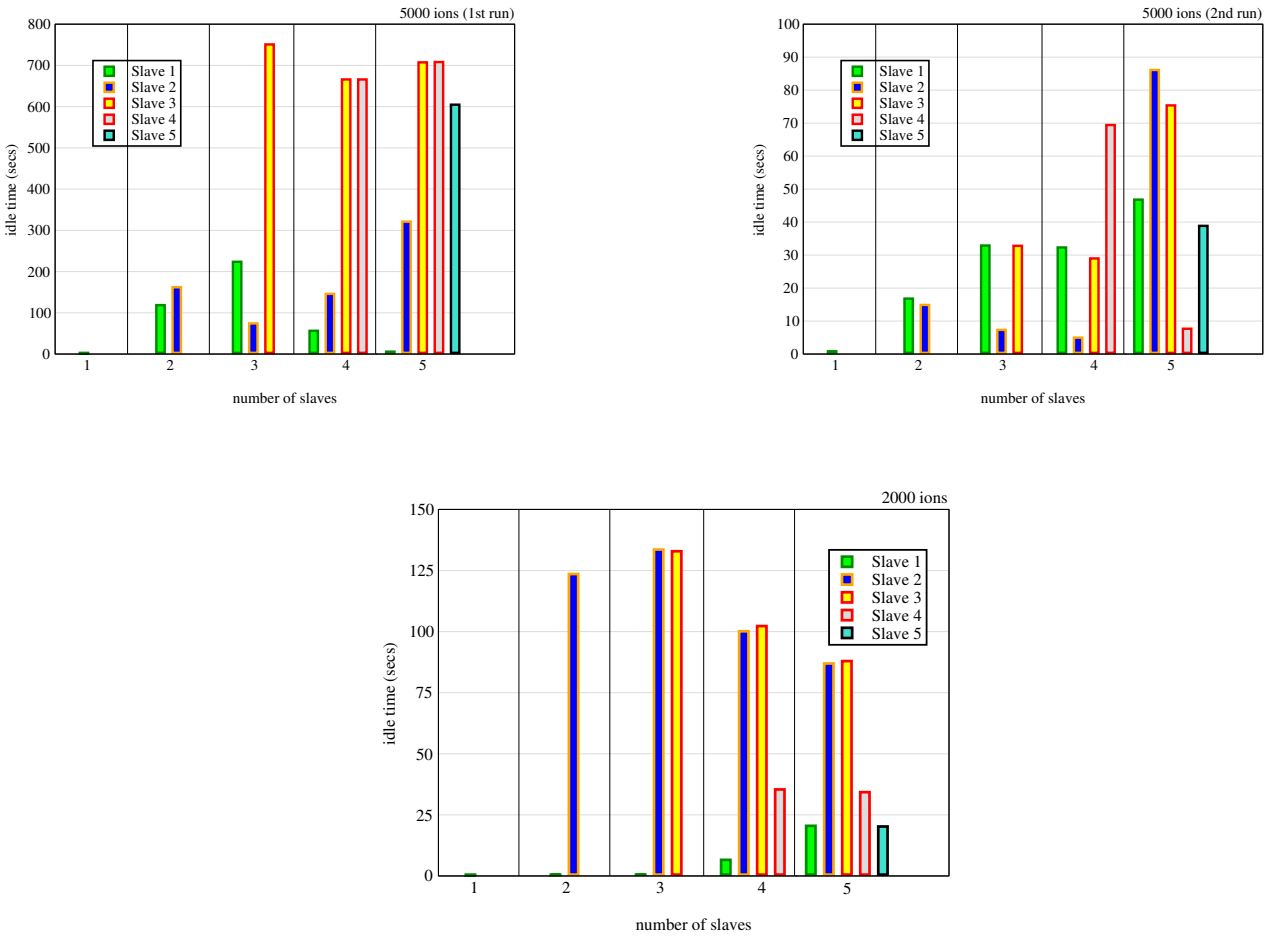

Figure 14. Idle times for the ion implantation simulator as measured and visualized by SCALA for two different problem sizes (2000 and 5000 ions) for varying number of slave nodes on a heterogeneous NOW.

been used to instrument and to examine 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 13 - based on the Grace graphical user interface of SCALA) have been measured by SCALA 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 (first 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. As load balancing is a common problem on non-dedicated systems we used SCALA to examine the idle times for various problem sizes on the NOW. Figure 14 shows the idle times (waiting for data from other slaves and the master plotted by the Grace graphical user interface of SCALA) for each slave of every specific execution (number of slaves is fixed) of the ion implantation simulator. Clearly, we can observe that the idle times across the slaves can be quite different. This is due to an 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 computational load. SCALA is one of very few tools that currently allows to measure idle times for parallel and distributed programs.

4 Related Work

The Paradyn system [35] is a dynamic performance instrumentation and measurement system. Instrumentation is currently restricted to functions and is controlled by a consultant module. Performance bottlenecks are tried to be found automatically through a rule-based refinement system. The user can control instrumentation overhead by limiting instrumentation to a threshold.

TAU [36] is a sophisticated instrumentation, tracing and profiling system that has been shown to be very useful for various programming paradigms including PC++ and HPC++ [30].

Forge90 [32] reports on communication costs at the level of a generated message passing code, but not at the level of the input program.

The SUIF Explorer [33], an interactive and interprocedural parallelizer, provides two sub-modules for performance analysis. First, the Execution Analyser which determines the loops that dominate the execution time of the program. Moreover, this tool can instrument a program for determining data dependences during execution of the code with the goal to locate parallelizable loops. Second, the Parallelization Guru provides two quantitative metrics to guide the parallelization process. Parallelism coverage reflects the limit on the speedup factor. Parallelism granularity defines the average length of computation between synchronizations in parallel regions. The SUIF Explorer guides the programmer through the parallelization process whereas SCALA provides more detailed performance information that can be exploited during program parallelization. Moreover, the SUIF Explorer provides performance analysis for mostly regular applications whereas SCALA also covers irregular applications.

An approach for visualizing the performance for HPF programs is described in [31]. 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 [21] and the Pablo performance system [39]. MPP Apprentice [45] supports post-execution performance analysis for C, C++, and Fortran90 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 (e.g. nested loops or procedures). SCALA can also deal with new code inserted by a compiler whose performance can be linked to a specific source of the input program.

5 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 which can be very error-prone and time consuming. In recent years compilers provide extensive support to develop distributed and parallel programs at a very high-level which reduces the time effort of code development substantially but sometimes at the cost of a reduced performance. Furthermore, compilers aggressively apply code transformations in order to convert a high-level program to a program with communication and synchronization and in order to improve the resulting performance. This poses a substantial problem for performance measurement and analysis tools. 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 order for performance measurement and analysis tools to be effective and useful, they must be applicable to both high- and low-level programming paradigms.

In this paper we describe the performance-oriented development of three real-world applications for distributed and parallel architectures. Two applications have been developed based on a high-level programming paradigm (HPF) and executed on a dedicated parallel machine (NEC Cenju-4). They benefit by fast program development and also achieve reasonable performance speedup. A third application is based on a master-slave programming model that has been manually developed and ported onto a cluster of heterogeneous workstation and networks. The development time was much longer (several months) as compared to the HPF code version for the previous application codes and was very error-prone. Good speedup figures have been observed for low system loads that are not related to the measured application. This application suffered by a static load balancing which is very insensitive towards dynamically changing application and machine characteristics.

SCALA, a portable instrumentation, measurement and post-execution performance analysis tool for distributed and parallel systems, has been used to support the performance-oriented program development of all three applications. The following features of SCALA have been particularly useful for these applications.

- Portable instrumentation system supports selective and comprehensive instrumentation of pre-defined types of code regions and arbitrary code regions.
- Code restructuring information of transformation systems records and collects in a measurement description file which enables to relate performance data back to the input program.
- Performance data of several executions can be compared against each other.
- Many important performance metrics can be computed (i.e., speedup, efficiency, communication and work distribution, compiler organization overhead, idle time, etc.).
- Several interfaces have been developed in order to support performance visualization of parallel and distribute applications.

SCALA is currently being used as a performance analysis system for explicit message passing programs (C and Fortran) and for programs generated by the VFC compiler [2] (translates HPF programs into message passing Fortran90 programs based on MPI). We are currently also investigating the usefulness of SCALA for performance analysis of distributed JAVA programs [16]. Moreover, we are in the process to integrate SCALA with performance prediction [14] and symbolic analysis techniques [15] to examine the scaling behavior [42] of distributed and parallel programs. Finally, we also work on fully automatizing the process of performance analysis [17]. This means that performance information is interpreted by SCALA which then automatically detects the most important performance problems of a program for a given architecture.

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 *Proc. of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems, IEEE Computer Society Press*, January 1996.

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

[3] P. Blaha, K. Schwarz, P. Dufek, and R. Augustyn. Wien95, a full-potential, linearized augmented plane wave program for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, 1995.

[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] 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.

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

[7] 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.

[8] 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.

[9] B. Carpenter. Adlib: A Distributed Array Library to Support HPF Translation. In *Proc. of the 5th Workshop on Compilers for Parallel Computers, Malaga, Spain*, June 1995.

[10] L. Clelow and C. Strickland. *Implementing derivative Models*. John Wiley & Sons, 1998.

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

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

[13] J. Fabozzi and T. Fabozzi. *The Handbook of Fixed Income Securities. Fourth Edition*. Irwin Professional Publishing, 1995.

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

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

[16] T. Fahringer. Javasymphony: A system for development of locality-oriented distributed and parallel java applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000)*, Chemnitz, Germany, Nov. 2000. IEEE Computer Society.

[17] T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff. Specification of performance problems in MPI programs with ASL. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP'00)*, pages 51–58, Montreal, CA, August 2000.
[18] 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.
[19] T. Fahringer, B. Scholz, and M. Pantano. Execution-Driven Performance Analysis for Distributed and Parallel Systems. Technical Report, Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Wien, June 1999.
[20] M. P. I. Forum. *Document for a Standard Message Passing Interface*, draft edition, Nov. 1993.
[21] 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.
[22] Grace User's Guide V0.2. http://plasma-gate.weizmann.ac.il/Grace/doc/UsersGuide.html, March 1999.
[23] 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.
[24] 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.
[25] 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.
[26] High Performance Fortran Forum, High Performance Fortran Language Specification. Version 1.1.δ, Technical Report, Rice University, Houston, TX, November 1994.
[27] J. C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, April 1997.
[28] J. C. Hull and A. White. One factor interest rate models and the valuation of interest rate derivative securities. *Journal of Financial and Quantitative Analysis*, (28):235–254, 1993.
[29] J. Hutchinson and S. Zenios. Financial simulations on a massively parallel connection machine. *The International Journal of Supercomputer Applications*, 5(2):27–45, 1991.
[30] 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.
[31] 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. IEEE Computer Society Press, 1995.
[32] 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.
[33] S.-W. Liao, A. Diwan, R. P. Bosch, A. Ghuloum, and M. S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. *ACM SIGPLAN Notices*, 34(8):37–48, Aug. 1999.
[34] M. Metcalf and J. Reid. *Fortran 90/95 explained*. Oxfor Science Publications, 1996.
[35] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37 – 46, November 1995.
[36] B. Mohr, D. Brown, and A. Malony. TAU: A portable parallel program analysis environment for pC++. In *CONPAR*, Linz, Austria, 94.
[37] H. Moritsch and E. Dockner. Numerical procedures for pricing interest rate dependent securities and their parallel implementations. Technical Report TR2000-??, Special Research Program SFB F011 AURORA, 2000.
[38] 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.
[39] 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.
[40] K. Schwarz and P. Blaha. Description of an LAPW DF Program (Wien95). *Lec.Notes in Chemistry*, pages 67:139–153, 1996.
[41] X.-H. Sun, M. Pantano, and T. Fahringer. Integrated Range Comparison for Data-Parallel Compilation Systems. Technical Report 97-004, Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803-4020, April 1997.
[42] X.-H. Sun, M. Pantano, and T. Fahringer. Performance Range Comparison for Restructuring Compilation. In *1998 International Conference on Parallel Processing*, Minneapolis, Minnesota, August 1998. IEEE Computer Society Press.
[43] 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.
[44] Sun Microsystems. Java RMI.
[45] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D, 1994.
[46] S. A. Zenios. *Parallel Monte Carlo simulation of mortgage-backed securities*. Cambridge University Press, 1993.