Estimating and Optimizing Performance for Parallel Programs

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
e-mail: tf@par.univie.ac.at
Published in IEEE Computer Vol. 28, No. 11, Nov, 1995

Abstract

P^3T is an interactive performance estimator that assists users in performance tuning of scientific Fortran programs. It detects performance bottlenecks in the program, identifies the causes of performance problems, and guides the user in selecting effective program transformations in order to gain performance. Four of the most critical performance aspects of parallel programs are estimated: load balance, cache locality, communication and computation overhead. P^3T is an integrated tool of the Vienna Fortran Compilation System, which enables the estimator to aggressively exploit considerable knowledge about the compiler's analysis information and code restructuring strategies. We evaluate existing features and describe substantial enhancements in three key areas: graphical user interface, performance parameters and input programs.

$P^3T's$ graphical user interface directs the user to bottlenecks in a computation that prevent the program from performing well. In addition, it allows filtering and visualizing of performance data at various levels of detail. A new performance parameter, which models computation time is discussed. Furthermore, P^3T has been extended to handle a subset of HPF programs.

Experiments show the ability of P^3T to successfully guide the user in performance tuning of parallel programs. This includes the selection of performance efficient data distribution strategies and profitable program transformations. The results are very encouraging, and demonstrate the feasibility of automatic performance prediction for parallel programs.

Key words: Performance prediction, Parallelizing compiler, Performance visualization, Performance tuning

1 Introduction

Widespread use of distributed memory multicomputers has been hindered by the difficulty of programming such machines, since users must write message-passing programs that deal with separate address spaces, communication, and synchronization. Parallelizing compilers provide the programmer with a global address space, numerous optimizing program transformations and data distribution strategies to simplify this task. Unfortunately, as of today all strategic parallelization decisions are really left to the user. Many programmers instrument (frequently done by manually inserting WRITE statements), compile and execute their codes to evaluate the potential performance gains or losses. Using runtime monitors automates instrumentation and runtime collection of performance data. However, runtime monitors still require the user to execute the parallel program which takes excessive time to arrive at the performance results. Other problems such as perturbation on the program's behavior and generation of vast seas of (mostly useless) data that require a performance expert to interpret makes performance analysis a tedious, error-prone and time-consuming task. Performance prediction tools can significantly expedite this task by providing fast and accurate information to guide the programmer to efficient data distribution strategies and/or profitable program transformations which results in increased performance.

The area of performance prediction for parallel programs has seen considerable research activity during the last few years: V. Sarkar ([11]) estimates the runtime of parallel programs at compile time using profiling and pre-measured kernel codes. This approach targets single assignment languages, which simplifies analysis significantly.

Balasundaram et al. ([1]) introduced a performance estimator to evaluate different data partitioning strategies by using pre-measured kernel codes. They achieve good results for the loosely synchronous programming model. K.Y. Wang ([10]) characterizes parallel programs by a parameterized performance model, which uses symbolic analysis to obtain runtime information. Although petri nets ([6]), queueing networks ([9]) and markov chains ([12]) can be valuable in understanding the dynamic behavior of parallel programs, the associated analysis costs both in terms of runtime and memory requirements prohibits their use in compilers. F. Hartleb and V. Mertsiotakis ([7]) derive upper and lower execution time bounds for parallel programs with the aim of tuning implementation and mapping alternatives. Parallel programs are modeled as a stochastic graph and random variables are used to describe the runtime behavior of a specific processor.

Most of the above approaches have serious drawbacks associated with them. Some of them have the problem of restricted applicability; they apply only to programs that may be modeled as multiple nested loops with constant loop bounds and no procedure calls and conditional statements. Some others are too expensive in terms of runtime and memory requirements. Purely theoretic approaches are difficult to validate against actual program behavior on real architectures.

In the domain of parallel computing, the problem of deriving performance estimates has a number of interesting facets:

- How should program unknowns be handled ?
- What performance information should be given to the programmer ?
- How accurate are the performance estimates ?
- Where are the program portions requiring performance improvement ?
- How long does it take to compute the performance data ?
- How should the performance data be filtered and visualized ?

In this paper we describe P^3T – a *Parameter based Performance Prediction Tool* – which is a performance estimator for parallel programs running on distributed memory parallel architectures. P^3T has been carefully designed to address all of the above performance estimation issues. In order to achieve high estimation accuracy, we aggressively exploit compiler analysis and optimization information. Our method is based on modeling loop iteration spaces, array access patterns, and data distributions by intersection and volume operations on n-dimensional polytopes. The most critical architecture specific factors such as cache line sizes, number of cache lines available, routing policy, startup times, message transfer time per byte, etc. are modeled to reflect the performance impact of the target machine. P^3T has been developed in the context of the *Vienna Fortran Compilation System (VFCS)*, which is a state-of-the-art parallelization tool for distributed memory systems. *VFCS* ([2]) translates Fortran programs into explicitly parallel message passing programs. P^3T is successfully used to guide the interactive and automatic restructuring of programs under this system.

The rest of this paper is organized as follows: Section 2 outlines the underlying compilation and programming model. The next section discusses the most critical design decisions made for P^3T, and briefly outlines the implementation of the parallel program parameters. The context in which P^3T is applied under *VFCS* and its graphical user interface is described in the next section. Section 5 reports on experimental results. Finally, some concluding remarks are made and future work is outlined.

2 Compilation and Programming Model

The programs which are estimated by P^3T are based on the underlying compilation and programming model of *VFCS* ([2]), which is a High Performance Fortran (HPF - [8]) style compiler. The parallelization strategy of *VFCS* is based on data decomposition in conjunction with the Single-Program-Multiple-Data (SPMD) programming model. With this method, the data arrays in the original program are each partitioned and mapped to the processors of the target architecture. The specification of the mapping of the array elements to the set of processors is called the *data distribution* of that program. A processor is then thought of as *owning* the data assigned to it; these data elements are stored in its local memory. The work contained in the program is distributed according to the data distribution: computations which define the data elements owned by a processor are performed by it – this is known as the *owner computes* paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally. If a computation requires data which is owned by a remote processor, then such non-local data is accessed through inter-processor communication, which is automatically implemented by *VFCS* through message passing.

The input to *VFCS* are Vienna Fortran programs ([2]). Vienna Fortran is a machine-independent language extension to Fortran77, which provides annotations for the specification of data distributions. Furthermore, *VFCS* supports a limited frontend for HPF programs ([8]), which includes PROCESSORS, DISTRIBUTE, ALIGN, and INHERIT HPF directives. Both Vienna Fortran and HPF programs are translated to the same internal program representation. As most compiler analysis – which also accounts for P^3T – is based on this internal representation, it can be applied to both HPF and VF programs without extra cost. The output of *VFCS* is a parallel Fortran program with explicit message passing.

3 Design Philosophy

In this section, we will describe the important design decisions made for the development of P^3T. Basically, there are four major aspects to consider:

- dealing with program unknowns
- making performance information transparent
- guiding the performance tuning of parallel programs
- displaying and filtering of performance data

3.1 Program Unknowns

Many existing performance estimators have a fundamental problem to obtain concrete values for variables in data and control flow such as statement execution and average loop iteration counts. We use a single sequential profile run of the original input program – ignoring all explicit parallel language constructs – on a single processor. For this purpose, we developed the Weight Finder ([3]), which is an advanced and highly optimized automatic profiler for Fortran programs. Profile runtime, memory requirements, and perturbation on the program's behavior is reduced by minimizing instrumentation overhead. In [3] we have shown that large portions of the profile data can be successfully adapted for most important program changes without redoing the profile run. We believe that profiling is a reasonable choice to handle program unknowns, in particular if we consider the fact that most other performance estimators apply guessing or simply ask the user for program unknowns. For a more detailed discussion on the issue of program unknowns and the Weight Finder, the reader may refer to [3].

3.2 Parallel Program Parameters

A key issue for a useful performance estimator is to provide critical information to the programmer and compiler, which allows steering of the performance tuning process. Most existing tools estimate only total runtime. The problem with this parameter is that all important information is hidden in a single runtime figure. As a consequence, the cause of potential performance losses remains unknown. It is not clear whether a parallel program's performance is poor due to cache, load balance, communication or computation behavior. Other performance parameters may as well play an important role. Without making such information transparent, performance tuning is extremely difficult. P^3T computes a set of parallel program parameters each of which reflects a different performance aspect. In the following all P^3T parallel program parameters are briefly outlined.

- *Work Distribution (WD)*: estimates how even the work contained in a parallel program is distributed across all processors executing the program. The work distribution is inherently defined by the owner computes paradigm and reflects how many times array assignment statements are executed by the processors. A statistical formula (variation coefficient) is used to describe the corresponding work distribution goodness.
- *Number of Transfers (NT)*: is an estimate of the number of transfer operations induced by a parallel program. A single *transfer* refers to the exchange of a data message between two processors. This and all other communication parameters model communication statements at all loop nest levels.
- *Amount of Data Transferred (TD)*: approximates the data volume (in bytes) transferred over the communication network as implied by a parallel program.

- *Network Contention (NC)*: represents an upper bound of the number of channel contentions on the underlying physical communication network as induced by a parallel program. It is assumed that a channel contention occurs if a network channel is traversed by two messages at the same time in the same direction.
- *Transfer Time (TT)*: predicts the time needed to transfer the required non-local data over the communication network.
- *Computation Time (CT)*: is an estimate for the sequential computation time as induced by a parallel program. This parameter does not include communication time, which is separately modeled by the previous parameter.
- *Number of Cache Misses (CM)*: models the data cache locality of parallel programs. It is defined by an upper bound of the number of cache lines accessed, which correlates with the number of cache misses.

In contrast to many other performance tools which predict only total execution time, the parallel program parameters as computed by P^3T can answer three fundamental performance questions:

1. *What performance aspects need to be improved ?*
2. *Where are the program portions requiring performance improvement ?*
3. *What must be done in order to gain performance ?*

The first question is answered, because every different parameter describes a specific performance aspect. This is in contrast to those approaches which hide all performance information in a single estimated runtime figure. P^3T is able to answer the second question, because the parallel program parameters can be selectively determined for statements, loops, procedures, and the entire program; furthermore their effect with respect to individual processors can be examined. Conventional performance estimators provide performance information, however, the user is not supported in finding effective strategies for performance tuning. With our approach the user can apply well-directed program transformations to eliminate or alleviate every specific performance drawback as indicated by the parallel program parameters. This answers the third performance question.

We achieve very good estimation accuracy for work distribution, number of transfers, amount of data transferred and transfer time parameters (estimation error usually less than 5 % for most experiments observed so far – see also [4]). For these parameters the accuracy consistently improves for increasing problem sizes. The parameters for number of cache misses, computation time and network contention specify upper bounds. The last three parameters are primarily used to rank different program versions with respect to their actual performance outcome. An in-depth evaluation of the estimation accuracy is also shown in [3].

The implementation of the parameters is quite general and handles programs containing procedure calls (Fortran subroutine and function calls), loop bounds and array subscript functions as linear functions of all enclosing loop variables, communication inside and outside of loops.

The parameters are computed based on an analytical model without incorporating simulation techniques. Loops uniquely define a loop iteration space, which is represented by a set of inequalities. The portion of the loop iteration space which corresponds to local array accesses of a processor p is determined by mapping the associated array subscript expressions based on the local array segments (owned by p) into the loop iteration space. This may induce an intersection of the array subscript functions with the loop iteration space. The volume of the resulting intersection body approximates the amount of work to be processed by p. If there is no intersection, then there is no work to be done by p. Geometric operations, such as intersection and volume algorithms for n-dimensional polytopes ([3, 5]) are incorporated for this analysis. Similar operations are used to model non-local array accesses, which refer to array portions to be exchanged between processors. Analyzing communication patterns, process to processor mapping strategy, interconnection network, and routing mechanism of the target architecture (e.g. e-cube routing on the Intel hypercube) allows estimation of the corresponding network contention behavior. Transfer times can be obtained by modeling communication patterns, data volumes transferred, interconnection network, process to processor mapping strategy, architecture dependent startup time, and message transfer time per byte. Estimating the number of cache misses requires the analysis of array access patterns (subscript expressions), grouping array accesses into array access classes[1] and accumulating the number of cache lines accessed by each array access class. Estimating computation times is based on pre-measuring a large set of kernels which range from primitive operations (e.g. basic arithmetic and control flow operations) to entire code patterns (e.g. matrix multiply). The kernels are measured on different architectures for varying problem sizes. In order to estimate computation times, a parallel program is parsed to detect existing pre-measured kernels incorporating pattern matching. For each

[1] Two array accesses belong to the same array access class, if they access a common cache memory location.

kernel discovered, the pre-measured runtime is multiplied by its associated statement execution count (as obtained by the Weight Finder) and accumulated, which finally yields the overall computation time.

The parallel program parameters are designed as machine independent as possible. However, in order to build a highly accurate performance estimator we added some of the most important machine specific factors including cache line size, overall number of cache lines available, data type sizes, routing policy, startup times and message transfer time per byte of the target architecture.

The priorities of the parallel program parameters may vary for different architectures. This information is important to the user, in order to know which parameters are most important to concentrate on first. For this reason, we apply a so-called training phase of P^3T with a large set of kernels and programs for each different architecture. The measured runtime is compared with the the associated parallel program parameters of each kernel. Based on the ranking of the different kernels both with respect to their measured runtime and their parameter outcome, P^3T determines the machine specific priorities for every parallel program parameter. For this task statistical methods are incorporated to compute the correlation between runtime and parallel program parameters. For instance, on the iPSC/860 hypercube it turned out that transfer time (TT) and the number of transfers (NT) are most critical. Work distribution, number of cache misses and computation time are in most cases less critical than TT and NT. Network contention has usually the lowest priority. This is validated by many experiments in [3].

The complexity to compute the parallel program parameters is independent of problem size, loop iteration and statement execution counts. As a consequence, the described method is considerably faster than simulating or actually compiling and executing the parallel program. For a detailed mathematical analysis of the parallel program parameters which also shows how to extend the parameters to loops, procedures and an entire program, including implementation, accuracy and portability issues, the reader may refer to [3, 5, 4].

3.3 Guide Performance Tuning

Our research in the area of performance prediction emphasizes four points:

- P^3T provides performance information about the most important aspects of the program's behavior.
- P^3T automatically guides the programmer to the cause of performance bottlenecks, and describes the problem in terms of the source code.
- The programmer can create various parallel program versions under *VFCS* through specifying different data distributions and applying code transformations. P^3T tries to find the best program version with respect to the actual runtime based on the parallel program parameters and associated priorities.
- For each parallel program parameter there exists a set of program transformations under *VFCS* which may improve the the corresponding performance outcome. For instance, changing the data distribution may improve the program's work distribution; loop distribution, privatizing variables, and scalar expansion are useful to decrease the communication overhead; data locality can be gained by strip mining, interchanging, and unrolling of loops, and by modifying the data distribution. A cut down of the computation time can be achieved by expression simplification, loop iteration elimination, etc. Data locality can be gained by strip mining, interchanging, and unrolling of loops, and by modifying the data distribution. Therefore, our approach enables the programmer to apply well-directed optimizations – provided by the underlying compiler – to alleviate or even eliminate each performance drawback as reported by P^3T.

3.4 Graphical User Interface

A graphical user interface is provided which allows filtering of performance data and relating it back to the original sequential parallel code. Color-coded performance visualization enables the user to immediately identify hot spots in the parallel program. Performance information can be filtered and displayed at various levels of detail. On the one hand, every specific performance parameter can be optionally displayed or hidden, on the other hand, the performance information can be displayed at the statement, procedure and program level. The graphical user interface will be considered more detailed in the following section.

4 The $\mathbf{P^3T}$: A Static Parameter based Performance Prediction Tool

In this section we discuss the context in which P^3T is used to guide the parallelization and optimization process under the *VFCS* (cf. Figure 1) and describe a graphical user-interface which allows the user to view performance

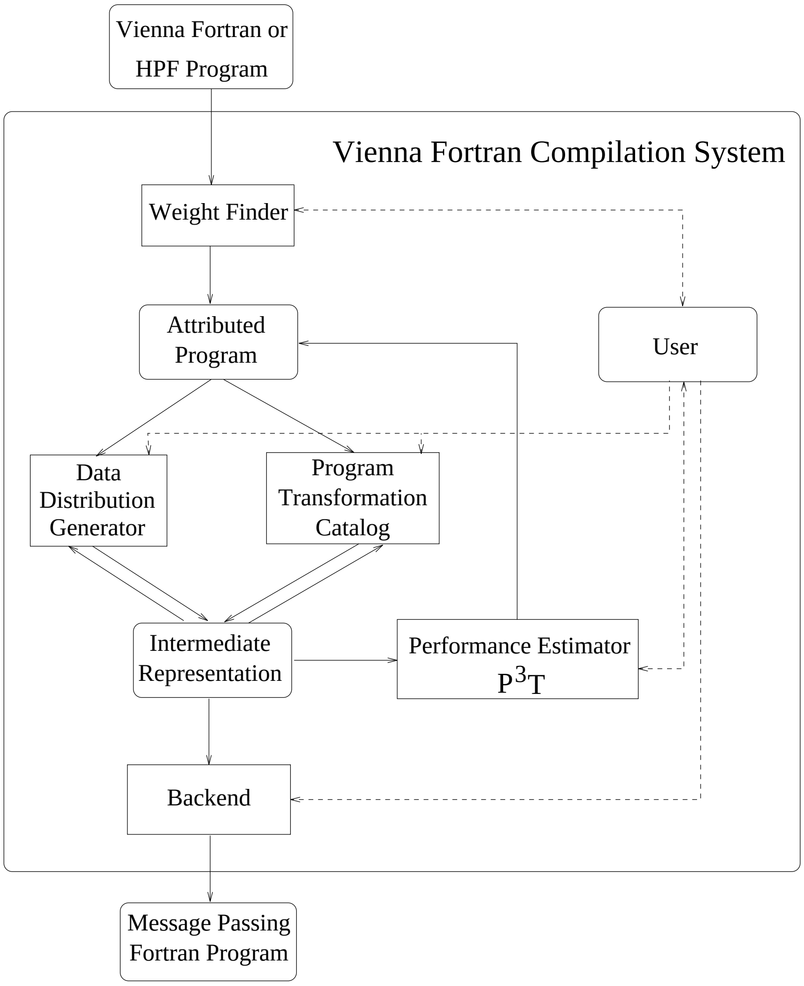

Figure 1: The Structure of P^3T as an integrated performance estimator of the Vienna Fortran Compilation System (VFCS). The programmer creates various parallel program versions under the VFCS through specifying different data distributions and applying code transformations. P^3T computes a variety of performance parameters for each program version and supports the user in finding effective strategies for performance tuning.

information from different levels of abstraction. We apply P^3T to an HPF JACOBI program version in order to demonstrate a parallelization and performance tuning session. This code is incorporated to approximate the solution of a partial differential equation discretized on a grid.

1. *VFCS Frontend*
   In the first step, the JACOBI program – written in HPF style – is run through the HPF frontend under *VFCS* ([2]), which involves extensive intra- and interprocedural analysis and program normalization and standardization. An intermediate code representation is used to make the program more readable.

2. *Weight Finder*
   In the next step, the Weight Finder (cf. Section 3.1) is applied. This tool automatically instruments and profiles a Fortran program in order to obtain concrete values for program unknowns (branching probabilities, loop iteration and statement execution counts) which is a prerequisite for P^3T.

3. *VFCS Parallelization and Optimization Engine*
   Based on the program attributed by the concrete values for program unknowns, the user may select a data distribution and/or a sequence of program transformations ([2]) to parallelize and optimize the program. A data distribution imposes the parallelism on a program, while program transformations such as loop interchange, distribution, and tiling are used to introduce, discover, and exploit parallelism and data locality.

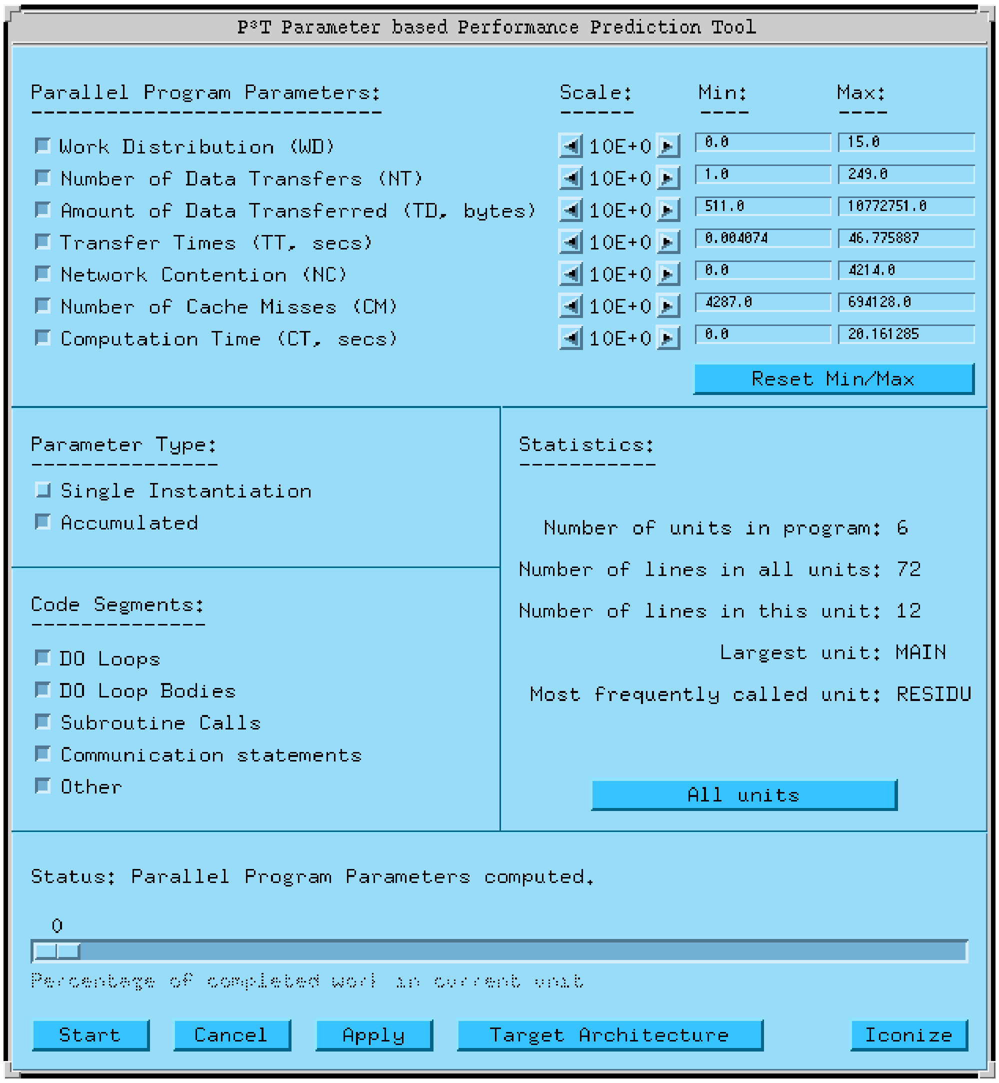

Figure 2: This window enables the user to select the parallel program parameters for specific code segments of interest. The resulting parameters can be scaled by a fixed order of magnitude and computed based on accumulated or single execution instance values.

Figure 4 shows the main program of an optimized parallel JACOBI version in the *VFCS* main window after several optimizing transformations have been applied. Program transformations can be automatically applied by using a separate *VFCS* menu.

4. P^3T

   Selecting the P^3T entry in the performance pulldown menu of the *VFCS* main window presents the P^3T *Options* window (cf. Figure 2). This window enables the user to selectively choose the parallel program parameters for code segments (for instance, DO loops, subroutine calls, etc.) of interest. The resulting parameter values can be scaled by multiplying them with a factor $v \in \{10^{-9}, \ldots, 10^{0}, \ldots, 10^{9}\}$. This can be useful for large parameter values, for instance, if the amount of data transferred should be displayed as Mbytes instead of bytes. Selecting the single instantiation parameter type results in the display of the parameters based on a single instantiation of the selected code segments, while the accumulated type yields accumulated figures with respect to the entire program. After the parallel program parameters have been computed, the minimum (best case) and maximum (worst case) parameter values as found in the entire program are displayed in the upper right section of the P^3T *Options* window (cf. Figure 2).

   In order to achieve good estimation accuracy the computation of the parallel program parameters depends on various machine parameters (cf. Section 3.2). After clicking on the *Target Architecture* button in the P^3T *Options* window, the user is enabled changing the architecture specific parameters via the *Target Architecture*

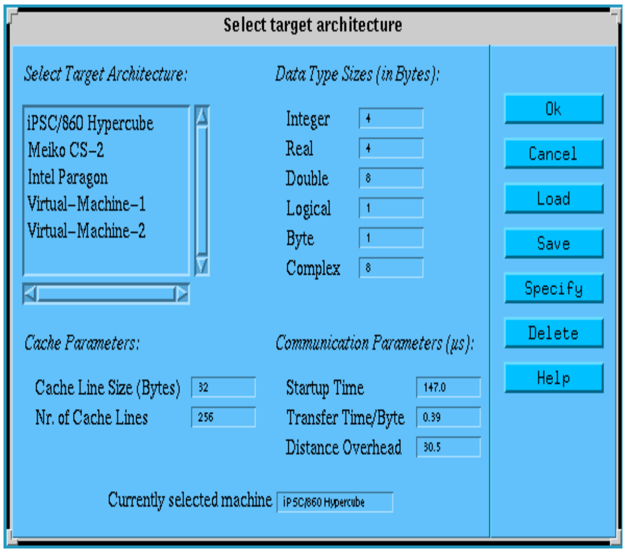

Figure 3: This window allows the user to change target architecture specific parameters, on which the computation of the parallel program parameters is based on. The user can either pre-select a target architecture (e.g. iPSC/860, Meiko CS-2, or Intel Paragon) or create a virtual machine by arbitrary defining the machine parameters and storing them under a virtual machine name. Architecture parameters can be changed and virtual machines can be stored, loaded and deleted.

window (cf. Figure 3). In addition to a set of pre-defined parameters for existing architectures (such as iPSC/860, Meiko CS-2 and Intel Paragon) the user can also define virtual machines by arbitrary specifying the machine parameters. This feature has been shown to be very useful, for instance, to evaluate a codes' performance for larger caches or better communication hardware, which are not yet available.

Selecting the *Start* button in the P^3T Options window initiates the computation of the parallel program parameters by P^3T. Based on the parallel program parameters the user is able to identify the location and the nature of performance losses. Figure 4 shows the *VFCS* main window with color-coded performance bars for all selected source code lines. A performance bar is defined by a sequence of sub-bars, each of which corresponds to a specific parameter outcome. The performance goodness is indicated by a range of colors. Each sub-bar contains a percentage value which indicates the performance outcome relative to the worst performance outcome as found in the entire program. For instance, the *NTA* value of source code line 13 indicates that this line accounts for 7.26 % of the accumulated number of transfers with respect to the entire program. The color-coded bar of the first source code line (*TASK PROGRAM MAIN*) corresponds to the performance of the entire program.

Clicking on an arbitrary source code line – for instance, line 19 in Figure 4 – will present a *Performance Data* window (cf. Figure 5), which visualizes detailed performance information for that line.

A global performance overview is presented by selecting the *All Units* button in the P^3T *Options* window. A window (see Figure 6) pops up, which displays a sorted list of all program units with respect to performance. The main program, all functions and subroutines define a unique program unit.

$P^3T's$ graphical user interface permits the user to immediately identify the hot spots of a program. Displaying the parallel program parameters transparently, as opposed to hiding them in a single estimated runtime figure, allows drastic narrowing of the transformation sequence and data partition search space. The user controls the subsequent application of program changes under *VFCS* and the estimation of the resulting performance outcome by P^3T, which implies a program parallelization and optimization cycle (cf. Figure 1).

5. *Backend*
   Once P^3T reports reasonable performance gains across all parallel program parameters or the code development time limit is reached, the user selects a specific backend for the desired target architecture, based on which *VFCS* generates an explicitly parallel and optimized message passing program.

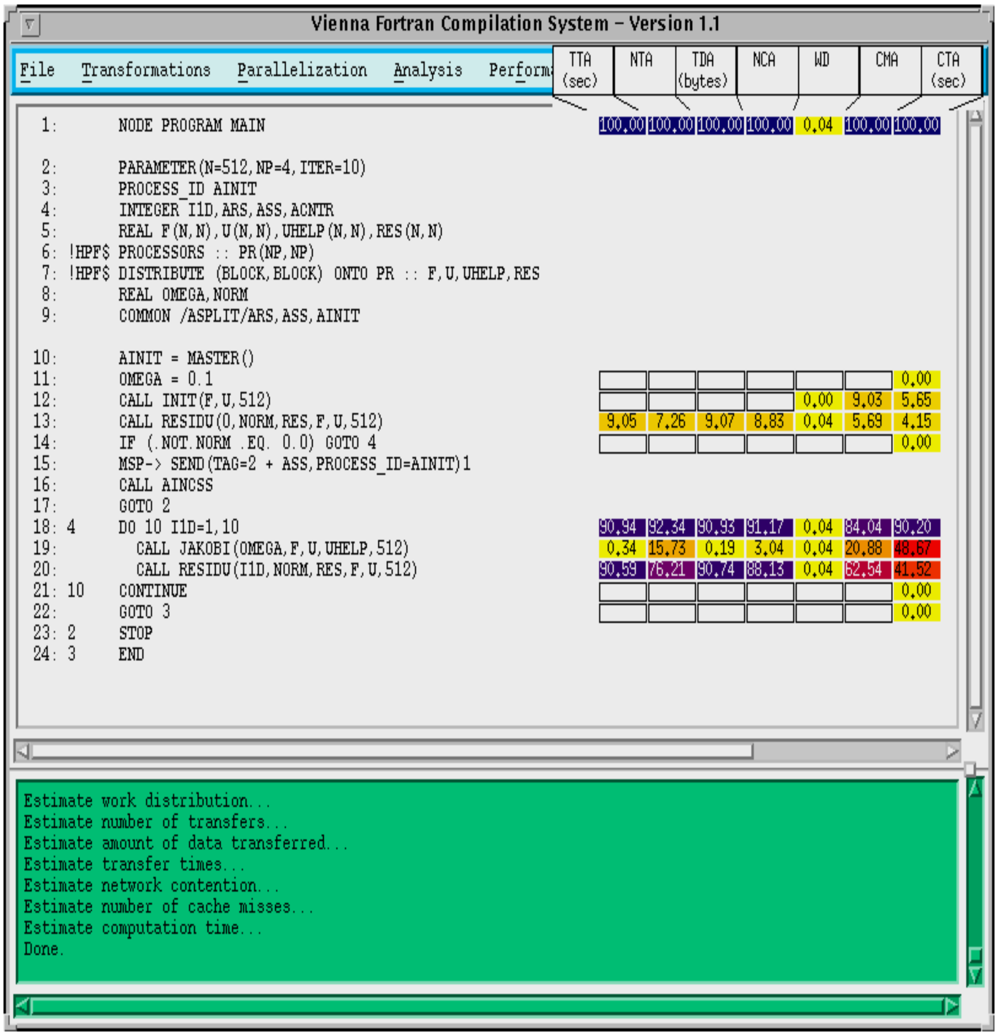

Figure 4: HPF JACOBI main program version with P^3T performance data in *VFCS* main window. Performance data is displayed by color-coded performance bars (set of sub-bars) for each program statement of interest. Each sub-bar corresponds to a specific parameter outcome: transfer times (TTA), number of transfer (NTA), amount of data transferred (TDA), network contention (NCA), work distribution (WD), number of cache misses (CMA), and computation time (CTA). The performance goodness is indicated by a range of colors; the brighter the color the better the performance outcome. Dark colors reflect performance losses (hot spots). Each sub-bar contains a percentage value which indicates the performance outcome relative to the worst performance outcome found in the entire program. Blank sub-bars indicate non-applicable performance parameters – for instance, pure computation statements cannot contain any communication.

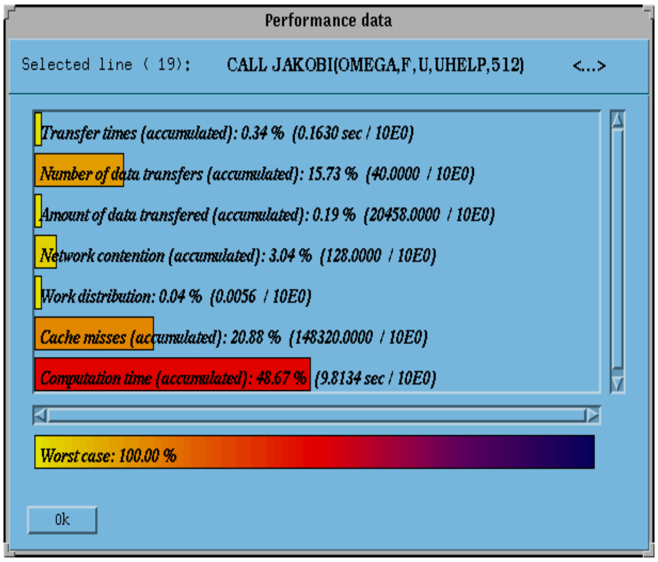

Figure 5: Performance visualization for a single statement. For each parameter the precise absolute outcome is visualized. The worst case performance outcome is established by the color-coded bar at the bottom of this window. All parameter bars are compared to this bar. In addition to the color also the length of the bar reflects the performance goodness, which is indicated relative to the worst performance outcome found in the entire program.

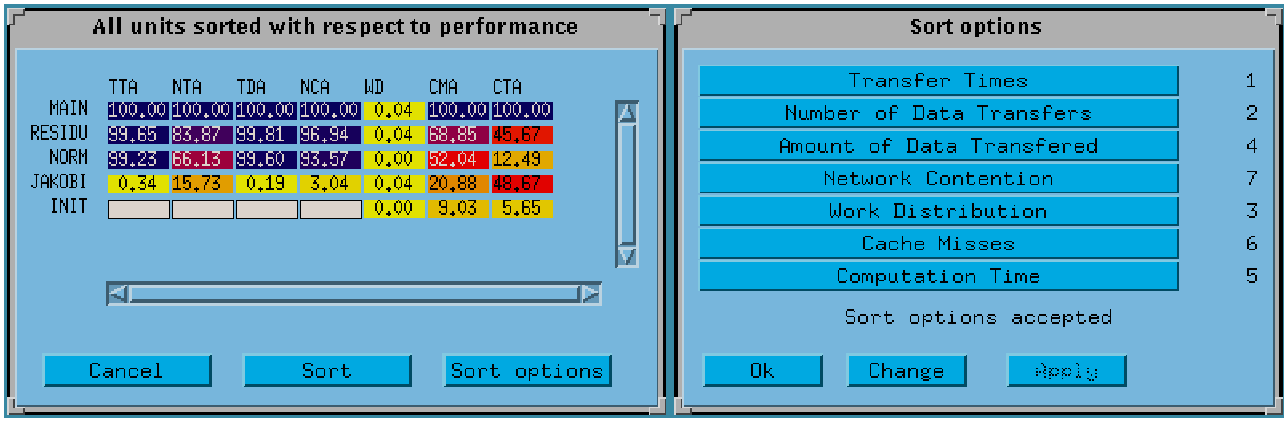

Figure 6: Sorted list of program units with respect to performance. The sorting is based on a list of parameter priorities, which can be optionally specified via the sort options window. Each parameter can be given a priority value between 1 and 7, where 1 defines the highest and 7 the lowest priority, respectively. As a consequence, the MAIN program reflects the worst and the INIT subroutine the best performance outcome.

5 Usefulness of P^3T

In this section we demonstrate how the parallel program parameters of P^3T can be effectively used to support the programmer in performance tuning of the ADI (Alternating Direction Implicit) code under the *VFCS*. ADI is a well known and effective method for solving partial differential equations in two or more dimensions. For all experiments we used the *VFCS* Version 1.1. to generate parallel program versions by applying a variety of program transformations and data distribution strategies, P^3T to statically compute the parallel program parameters, and the iPSC/860 hypercube (16 processors) to execute the parallel programs and measure their corresponding runtimes for validation of our predicted results. In the following we describe a series of code changes applied to the ADI code based on the performance outcome of $P^3T's$ parallel program parameters.

5.1 Selecting a data distribution strategy

The runtime of the original sequential ADI code for a specific problem (array) size N=512 was measured to be 9.459 seconds (cf. Figure 7) on a single node of the Intel hypercube. The original code is parallelized by applying three different data distribution strategies:

v1: Distribute the first dimension of all arrays onto 16 processors (row-wise distribution).

v2: Distribute the arrays 2-dimensional block-wise onto a 4x4 processors array.

v3: Distribute the second dimension of all arrays onto 16 processors (column-wise distribution).

Figure 7 displays the three program versions – one for each different data distribution – with their associated measured runtime MT (indicated in seconds) and the most relevant parallel program parameters. Note that the MT values are not estimated but reflect the actually measured runtime as taken on the iPSC/860 hypercube.

According to the parameter priorities of Section 3.2 – the transfer time (TT) parameter has highest priority – P^3T estimates that program version $v1$ has the best performance outcome of all three different data distributions. This is validated by the corresponding measured runtime figures. Note that the estimated transfer times are larger than the associated measured runtimes. This is due to the fact that P^3T sums up the transfer times for all communication statements inside of loops. The important point, however, is that the transfer time parameter is about two orders of magnitude higher than the computation time parameter. This clearly indicates that $v1$, $v2$, and $v3$ imply a significant communication overhead. There are two reasons for that. First, most of the communication cannot be hoisted out of the two main ADI loops due to data dependences. Second, scalar assignments, which have to be executed by all processors, require the exchange of all local array segments across all processors. Note also that all 3 different data distributions imply an excellent work distribution (close to zero which is optimal).

5.2 Loop distribution of the main loops

At this point, P^3T detects that there is significant communication inside of the two main loops L1 and L2 in the ADI code and proposes loop distribution which may break data dependences and consequently allows pulling communication out of loops. Loop distribution ([2]) is used to expose partial parallelism by separating statements which may be parallelized from those that must be executed sequentially. The user now selects loop distribution of L1 and L2 which is automatically applied by *VFCS*. This transformation reduces the amount of communication drastically as indicated by the associated transfer time figures for $v1'$, $v2'$ and $v3'$. The resulting improvement in performance is clearly confirmed by the associated measured runtimes. Note that also the cache behavior has improved due to this transformation. The similarity of both row-wise and column-wise distribution regarding all parameters is due to the symmetric performance behavior of loop L1 and L2 with respect to these data distribution strategies. Loop L1 favors column-wise distribution while loop L2 is predestined for row-wise distribution. For this reason, only row-wise and 2Dblock-wise distribution are compared for all subsequent optimizations.

After loop distribution the 2D block-wise distribution outperforms the row-wise distribution, which is correctly reported by the transfer time parameter.

5.3 Scalar forward substitution

In order to reduce redundant computation in the ADI code, scalars are incorporated as temporary variables to store common subexpressions. The underlying compilation model requires all processors to execute all instantiations of scalar assignment statements. This may induce extensive communication, if distributed arrays are referenced on the right hand-side of a scalar assignment statement. In order to prevent this fall-off in performance, the VFCS

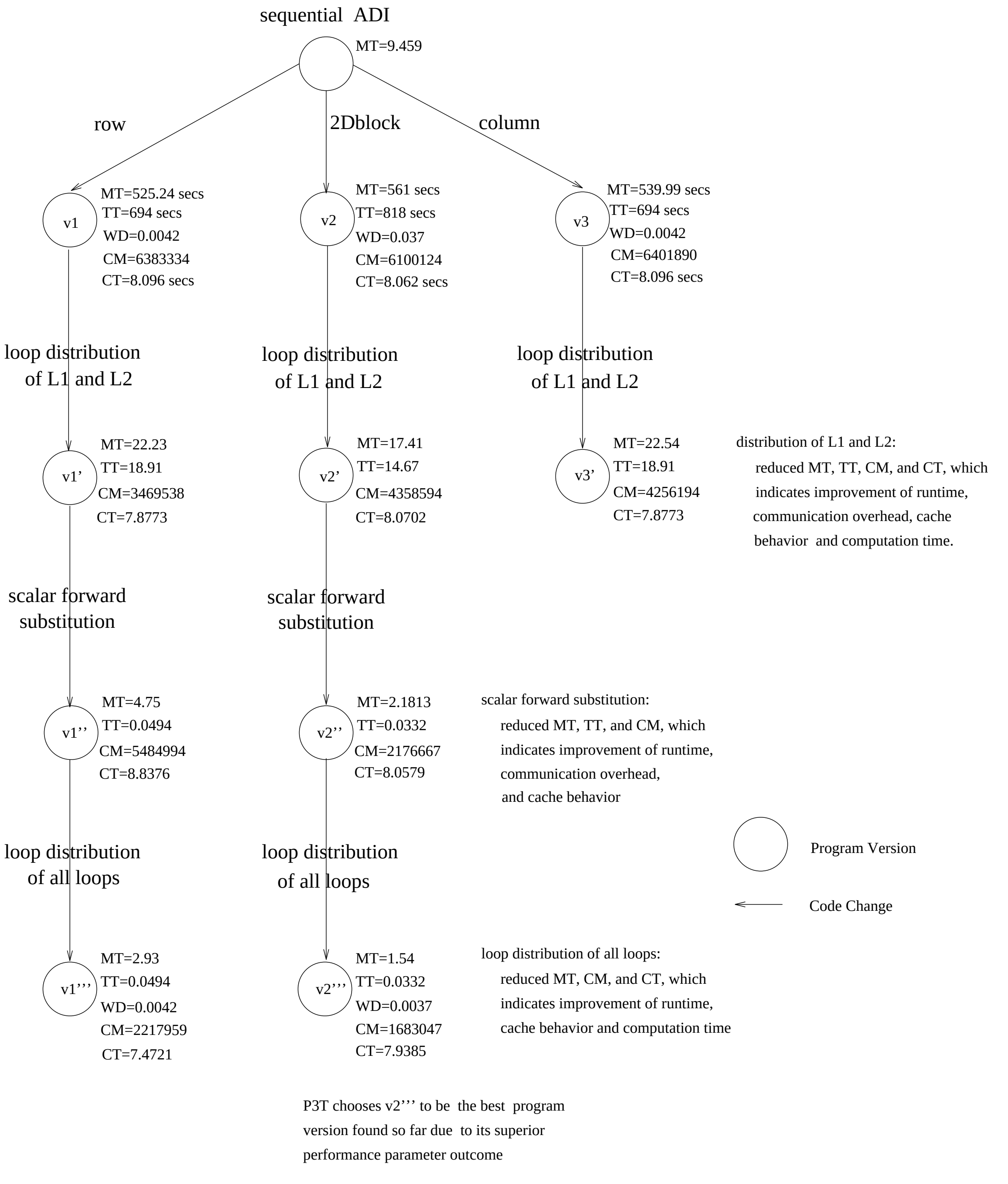

Figure 7: Using the P^3T to support the performance tuning of the ADI program for an iPSC/860 hypercube, which spawns a search tree of code changes (data distributions and program transformations). Measured parameters: runtime (MT); Estimated parameters: transfer time (TT), work distribution (WD), number of cache misses (CM), computation time (CT). Column and row-wise distribution have almost identical performance behavior, therefore, only one of these strategies (column-wise distribution) is followed beyond the step where L1 and L2 are distributed.

provides the user with a *Scalar Forward Substitution* transformation, which replaces – under appropriate conditions ([2]) – an applied occurrence of a scalar variable by the expression on the right-hand side of an assignment to that variable. After applying scalar forward substitution to the ADI code, the measured runtime (MT) improves by an approximate factor of 4.5 and 8 for $v1''$ and $v2''$ relative to $v1'$ and $v2'$, respectively. P^3T confirms that by a vast reduction of the associated transfer times.

5.4 Loop distribution of all loops

P^3T detects that the communication overhead is reasonable small as compared to CT. At this point code changes should be investigated to improve other performance parameters such as the high number of cache misses. Loop distribution frequently allows exploiting spatial locality, in particular, if several large arrays are accessed inside of a loop nest. This transformation permits also optimizing of the loop nest for the only statement contained in it which results in less complex loop bounds and array assignment masks. After application of this transformation, P^3T reports an improved cache miss ratio and computation time parameter for both $v1'''$ and $v2'''$, which is again validated by a reduced runtime figure. Loop distribution increased the number of loops. This drawback is compensated by less complex loops and array assignment masks, which is validated by decreased computation time (CT) values.

P^3T chooses $v2'''$ to be the best program version of the entire search tree due to its superior performance behavior with respect to transfer time, work distribution and cache performance. This program version uses 16 processors and achieves an approximate speedup of 6 with respect to the runtime of the sequential ADI code.

The computation time parameter accounts for the original computation contained in the input program plus the extra overhead implied by parallelizing the code. Communication time is not considered by this parameter but modeled separately by the communication time parameter. CT should, therefore, be compared to MT of the sequential ADI version as opposed to the MT value of the associated parallel ADI version. In addition, this parameter is sensitive to the improved cache behavior due to the application of loop distribution.

This experiment demonstrated that P^3T successfully guides the programmer in the selection of an efficient data distribution strategy and in the application of various profitable program transformations.

6 Conclusion

The most important motivation for writing parallel programs is to achieve high performance. Unfortunately, the techniques and tools needed to aid in program performance evaluation and tuning are lagging behind the development of parallel programs. Many programmers frequently hand-instrument, compile and execute their codes on the target architecture to evaluate the corresponding performance. Performance monitors substantially alleviate this effort, however, they still require to execute the code which may take excessive time to arrive at performance results. Programmers are frequently inundated with irrelevant details of vast amount of performance data obtained. Program perturbation due to instrumentation overhead is yet another disadvantage of runtime monitoring. On the other hand, a major advantage of monitors is their ability to handle arbitrary large codes. Performance estimators are the right choice for reasonable sized regular codes or code sections whose performance behavior can be analyzed at compile time. A usable yet accurate performance estimator for parallel programs is needed to build second generation parallelizing compilers, which require guidance for performance tuning of parallel programs. P^3T is, to the best of the authors knowledge, the first performance estimator, which steers both the selection of efficient data distribution strategies and the application of profitable program transformations. This can be done for both Vienna Fortran and a subset of HPF programs. P^3T statically computes a set of parameters that characterize the behavior of parallel programs. This includes work distribution, number of data transfers, amount of data transferred, transfer time, computation time, network contention, and number of cache misses. These parameters can be selectively determined for statements, loops, procedures, and the entire program; in addition, their effect with respect to individual processors can be examined. The computational complexity of the parallel program parameters is independent of the program's problem size, statement execution and loop iteration counts. As a consequence, estimating the parallel program parameters is considerably faster than simulating or actually compiling and executing the program.

P^3T does not only report on the performance outcome of parallel programs, but going beyond that, allows the user to apply well-directed program transformations to eliminate or alleviate each specific performance drawback as reported by the parallel program parameters. For each parallel program parameter the underlying compiler provides a set of program transformations to improve the corresponding performance outcome.

A graphical user interface is provided which allows filtering of performance data and relating it back to the original parallel code. Color-coded performance visualization enables the user to immediately identify hot spots in the parallel program. The program's performance data can be filtered and displayed at various levels of detail.

Ongoing work to fine tune the estimator for a larger set of optimizing transformations and to evaluate it for several other distributed memory architectures will further enhance the usefulness of P^3T.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *3rd ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Williamsburg, VA, April 21-24 1991.

[2] B. Chapman, S. Benkner, R. Blasko, P. Brezany, M. Egg, T. Fahringer, H.M. Gerndt, J. Hulman, B. Knaus, P. Kutschera, H. Moritsch, A. Schwald, V. Sipkova, and H.P. Zima. *VIENNA FORTRAN Compilation System - Version 1.0 - User's Guide*, January 1993.

[3] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, University of Vienna, Institute of Software Technology and Parallel Systems, October 1993.

[4] T. Fahringer. Using the P^3T to Guide the Parallelization and Optimization Effort under the Vienna Fortran Compilation System. In *IEEE Proc. of the Scalable High Performance Computing Conference 1994, Knoxville, TN*, May 1994.

[5] T. Fahringer and H. Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. In *Invited Paper, In Proc. of the 7th ACM International Conference on Supercomputing* , Tokyo, Japan, July 1993.

[6] A. Ferscha. A petri net approach for performance oriented parallel program design. *Journal of Parallel and Distributed Computing*, 15:188–206, 1992.

[7] F. Hartleb and V. Mertsiotakis. Bounds for the Mean Runtime of Parallel Programs. In *Computer Performance Evaluation '92: Modeling Techniques and Tools*, pages 197–210, 1992. (Ed.) R. Pooley and J. Hillston.

[8] High Performance FORTRAN Language Specification. Technical Report, Version 1.0, Rice University, Houston, TX, May 1993.

[9] H. Jonkers. Queueing Models of Parallel Applications: The Glamis Methodology. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools*, pages 123–138, Vienna, Austria, May 1994. Lecture Notes in Computer Science, Springer Verlag.

[10] K.Y.Wang. A Framework for Static, Precise Performance Prediction for Superscalar-Based Parallel Computers . In H. Sips, editor, *In Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, pages 413–427, Delft, The Netherlands, December 1993.

[11] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessor*. The MIT Press, Cambridge, Massachusetts, 1989.

[12] K. Trivedi, R. Haverkort, A. Rindos, and V. Mainkar. Techniques and Tools for Reliability and Performance Evaluation: Problems and Perspectives. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools*, pages 1–24, Vienna, Austria, May 1994. Lecture Notes in Computer Science, Springer Verlag.

[13] N. Yazici-Pekergin and J.M. Vincent. Stochastic Bounds on Execution Times of Parallel Programs. *IEEE Transactions on Software Engineering*, 17(10):197 – 217, October 1991.