$\mathbf{P^3T+}$: A Performance Estimator for Distributed and Parallel Programs *

T. Fahringer and A. Požgaj
Institute for Software Science, University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
[tf,alex]@par.univie.ac.at

Published in Scientific Programming, IOS Press, The Netherlands, Vol. 8, Nr. 2, Nov. 2000.

Abstract

Developing distributed and parallel programs on today's multiprocessor architectures is still a challenging task. Particular distressing is the lack of effective performance tools that support the programmer in evaluating changes in code, problem and machine sizes, and target architectures. In this paper we introduce P^3T+ which is a performance estimator for mostly regular HPF (High Performance Fortran) programs but partially covers also message passing programs (MPI). P^3T+ is unique by modeling programs, compiler code transformations, and parallel and distributed architectures. It computes at compile-time a variety of performance parameters including work distribution, number of transfers, amount of data transferred, transfer times, computation times, and number of cache misses. Several novel technologies are employed to compute these parameters: loop iteration spaces, array access patterns, and data distributions are modeled by employing highly effective symbolic analysis. Communication is estimated by simulating the behavior of a communication library used by the underlying compiler. Computation times are predicted through pre-measured kernels on every target architecture of interest. We carefully model most critical architecture specific factors such as cache lines sizes, number of cache lines available, startup times, message transfer time per byte, etc. P^3T+ has been implemented and is closely integrated with the Vienna High Performance Compiler (VFC) to support programmers develop parallel and distributed applications. Experimental results for realistic kernel codes taken from real-world applications are presented to demonstrate both accuracy and usefulness of P^3T+.

1 Introduction

Parallelizing and optimizing programs for multiprocessor systems with distributed memory is still a notoriously hard task. In most cases it is the programmer's responsibility to find parallelism, to distribute data (data parallelism) and computations (task parallelism) onto the target architecture, and to apply code transformations in order to improve performance. Programmers are faced with many problems when it comes to examine the performance of their codes:

- What is the effect of a code change in the performance of a program?
- What happens to the performance if problem and machine sizes are modified?
- How much performance can be gained by changing a specific machine parameter (e.g. communication bandwidth or cache size)?

Clearly this list is incomplete, but it shows, that tools providing accurate performance information to examine some of these effects are of paramount importance.

Historically there have been two classes of performance tools. On the one hand, there is extensive work done on monitoring distributed and parallel applications but these approaches have several drawbacks: availability of program and target architecture, long execution times, perturbation of measured performance data, and vast amounts of performance data. Monitoring, however, in principle can handle arbitrary complex and large codes and commonly also provides quite accurate results.

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

On the other hand, there is the class of performance estimators that try to statically examine a program's performance without executing it on a target architecture. This approach suffers mostly by restricting programs and machines that can be modeled as well as by less accurate results. Performance prediction does not require that the target architecture must be available. Moreover, the time needed to compute performance information can be very short.

In this paper we concentrate primarily on performance prediction which has seen many research efforts in the last several years. Traditionally, the quality of performance prediction has been hampered by modeling either programs or architectures with good accuracy but not both of them. Firstly, those methods that provided accurate predictions for applications suffered by some severe restrictions imposed on modeling architectures. Commonly these tools are unable to determine useful parameters reflecting computational and communication overhead. Secondly, performance prediction that concentrates on modeling architectures may not have enough information about the application that executes on this architecture. Statistical models are commonly used to assume a more or less virtual and often unrealistic application behavior. Moreover, very few performance estimators actually consider code transformations and optimizations applied by a compiler.

In this paper we introduce P^3T+, the successor tool of P^3T [22, 16, 17], which models programs, code transformations, and parallel and distributed architectures. The input programs of P^3T+ are written in High Performance Fortran [27, 2] which represents the de-facto standard of high-level data parallel programming. Moreover, P^3T+ analyzes Fortran90 message passing programs generated by the underlying compiler (VFC [3]) which can be executed on parallel and distributed machines such as network of workstations. P^3T+ models communication overhead, work distribution, computation times, and cache misses which is important for both distributed and parallel programs.

P^3T+ invokes a single profile run of the original sequential input program – ignoring all explicit parallel language constructs such as HPF directives – by using SCALA [21] in order to determine execution frequencies and branching probabilities. In order to achieve high estimation accuracy, we aggressively exploit compiler analysis and optimization information. P^3T+ computes a variety of parameters that reflect some of the most important performance aspects of a parallel program which includes: work distribution, number of transfers, amount of data transferred, transfer times, computation times, and cache misses.

Our estimation technology is based on modeling loop iteration spaces, array access patterns, and data distributions by employing highly effective symbolic analysis. Communication is estimated by simulating the behavior of the communication library as employed by the underlying compiler. Computation times are predicted through kernels which are pre-measured on every target architecture of interest. We carefully model most critical architecture specific factors such as cache lines sizes, number of cache lines available, startup times, message transfer time per byte, etc.

The rest of this paper is organized as follows: The following section discusses related work. In Section 3 we describe P^3T+ and its performance parameters. Section 4 reports on experimental results by using several realistic kernel codes taken from real-world applications. Finally, some concluding remarks are made and future work is outlined.

2 Related Work

J. Brehm et al. [6] built a user-driven performance prediction tool $PerPreT$ based on an analytical model to predict speedup, execution time, computation time and communication time for parallelization strategies. The tool examines application strategies without requiring a program. Communication and computation times are described by parameterized formulas where parameters describe the the application's problem size and the number of processors. The target machine is modeled by architectural parameters such as the setup times for computation, link bandwidth and sustained computing performance per node (expressed in MFLOP/s). The user can describe the application and machine model through a specific language called LOOP [29]. While $PerPreT$ offers an interesting possibility to evaluate the computation and communication times required by a parallel application, it does not provide information about work distribution or number of cache misses.

In [32] W. Kaplow et al. present a compile-time method for determining the cache performance of the loop nests in a program and a heuristic that uses this method for compile-time optimization of loop ranges in iteration-space blocking. The cache misses estimations are produced by applying the program's reference string of a loop nest, determined during compilation, to an architecturally parameterized cache simulator. Data reference strings are generated while parsing the source code as opposed to most hardware cache simulators where reference strings are generated at run-time. Data reference strings are then used by a simulator whose results are less accurate than hardware simulation. However, their approach appears to be effective enough for loop optimization techniques.

W. Kaplow and B. Szymanski [32] described an approach to estimate cache behavior for parallel programs based on realistic simulation of the input program for parallel architectures. Array reference traces are simulated at compile-time. The simulator can predict what is the next set of indices for the same array reference that will access the data beyond the cache line just loaded. The speed at which program execution is simulated is proportional to the cache miss rate of the simulated loop nest which is much slower than our analytical approach.

W. Meira et al. developed Carnival [31] which is a performance measurement and visualization tool for SPMD message-passing programs that automates the cause-and-effect inference process for waiting time. Carnival uses detailed event traces to gather performance information, which it presents both as global summary statistics and as localized performance profiles, facilitating top-down performance analysis. The user interface presents performance information together with the source code, creating a link between the observed phenomena and the code. Carnival supports waiting time analysis, an automatic inference process that explains each source of waiting time.

In [11, 10] M. Clement et al. present a compiler-generated analytical model for the prediction of cache behavior, CPU execution time, and message passing overhead for scalable algorithms implemented in high level data-parallel languages. The performance prediction requires a single instrumentation run of the program with a reduced problem size to generate a symbolic equation for execution time which includes the contributions of each basic block in a program expressed as a function of the problem size and the number of processors. Since the result of this model is an equation rather than a time estimate for a given problem size, the execution time can be differentiated with respect to a given system parameter. The resulting equation is used to determine the sensitivity of the application to changes in that parameter as the problem is scaled up. Their approach is more restricted in terms of program classes that can be handled (e.g. more restricted loops, no GOTOs, etc.) as compared to P^3T+.

M. Faerman et al. [15] introduced the *Adaptive Regression Modeling (AdRM)* which is a method for performance prediction of data transfer operations in network-bound distributed data-intensive applications. The presented technique predicts performance in multi-user distributed environments by employing small network bandwidth probes (provided by the Network Weather Service (NWS) [48]) to make short-term predictions of transfer times for a range of problem sizes. The NWS gathers performance probe data from a distributed collection of resources and catalogues that data as individual performance histories for each resource. It then applies lightweight time series analysis models to each performance history to produce short-term forecasts of future performance levels. *AdRM* combines the NWS measurements with instrumentation data taken from actual application runs to predict the future performance of the application. To capture the relationship between NWS probes and application benchmark data, regression models are used which calibrate the application execution performance to the dynamic state of the system measured by the NWS. The result is an accurate performance model that can be parameterized by "live" NWS measurements to make time-sensitive performance predictions which can be used to support adaptive scheduling of individual components of a distributed system.

In [23], W. Fang et al. present a method for the evaluation of the communication overhead in the SHRIMP multicomputer under a variety of workloads: analytic modeling and event-driven simulation. Using both methods, the authors study the behavior of the system under different communication patterns and report on system performance parameters such as message latency, occupancy of system buffers and network congestion. The purpose of their work is to learn about the behavior of the SHRIMP machine, and to explore the tradeoffs between analytic modeling and simulation as performance prediction techniques. Their analytic model is based on two assumptions: (i) packet inter-arrival times and service times at every component are exponentially distributed, and (ii) the states of any pair of components are independent random variables. While these assumptions do not match the way the system really operates, the authors believe they do not introduce a significant error in the model. Furthermore, the model assumes that each processor executes the same program, that all messages are of the same size and that messages are sent to uniformly distributed destinations.

In [44, 40], A. van Gemund presents a methodology that yields parameterized performance models of parallel programs running on shared-memory as well as distributed-memory (vector) machines. The aim of this research is to estimate performance degradation due to synchronization effects, covering both condition synchronization (task dependency) as well as mutual exclusion (resource contention). The author introduces an explicit, highly structured formalism called PAMELA together with an analysis technique that integrates an approximate analysis of mutual exclusion within a conventional condition synchronisation analysis technique.

There is a variety of related projects which focus on performance analysis based on real executions of the parallel program on the target architecture.

B. Miller et al. developed the *Performance Consultant* (PC) as part of the PARADYN project [37] which searches for performance bottlenecks according to the W^3 Search Model. An automatic online performance analysis is conducted by using dynamic instrumentation for monitoring. Hypotheses can be defined to determine the occurrence of a set of performance bottlenecks which is currently predefined. It includes *CPUbound*, *Excessive Sync Waiting Time*, *ExcessiveIOBlockingTime*, and *TooManySmallIOOps*.

F. Wolf and B. Mohr have built EARL [46] which enables description of performance event patterns in message passing programs in a procedural fashion as scripts in a high-level event trace analysis language. Frequently used, higher-level events

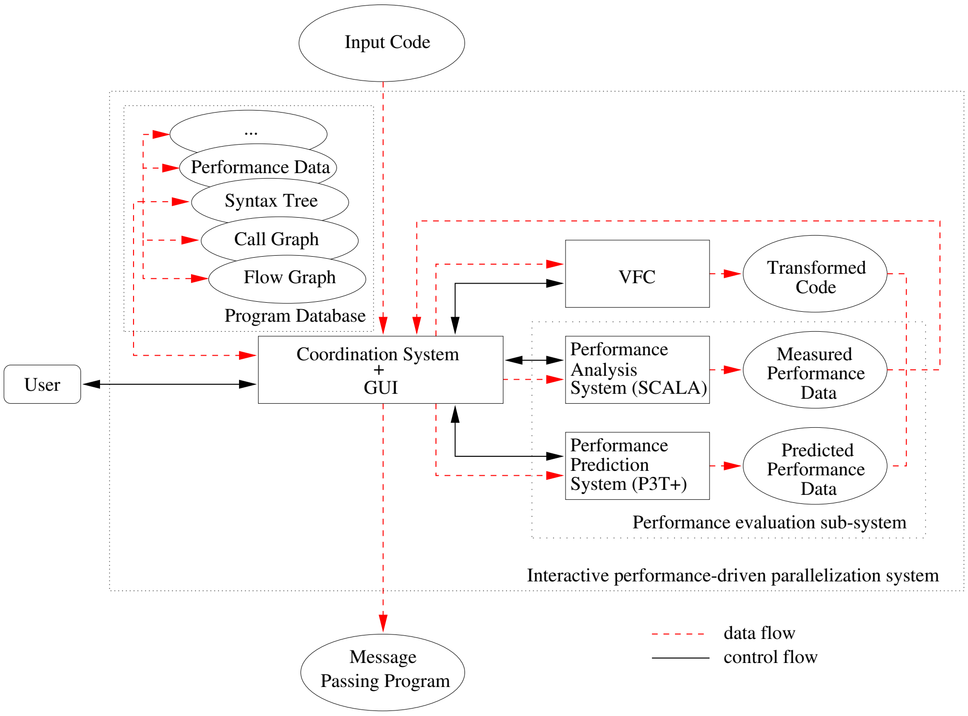

Figure 1. Performance-driven development of distributed and parallel programs

like region instances or message transfers are represented by links between their constituent events, which can be easily traversed by a script.

A. Espinosa et al. developed KAPPA-PI [1] which is an automatic performance analyzer for PVM-programs. It is a post-execution tool, implemented in PERL, that evaluates traces generated by the Tape/PVM monitoring library or by the VAMPIR MPI trace library. Based on a predefined list of performance bottlenecks, it searches for performance problems and their causes. In addition to trace data, it analyzes the source code using pattern matching.

3 P^3T+: A Performance Estimator for Distributed and Parallel Programs

P^3T+ is a state-of-the-art performance estimator that targets both distributed and parallel programs. Figure 1 shows P^3T+ as part of a program development and optimization system. Input programs are parsed and analyzed by VFC which generates syntax trees, call graphs, flow graphs, etc. and stores them in a program database. VFC applies various code transformations and optimizations to the program with/without user control. The programmer can invoke a performance analysis system (SCALA) to instrument, compile, and execute a distributed or parallel program on the target architecture. Based on the instrumented program execution, performance data is gathered and stored in the program database. Moreover, P^3T+ can be employed to predict the performance behavior of the code transformations and optimizations applied by VFC. P^3T+'s performance data is also stored in the program database. All three tools (VFC, SCALA, and P^3T+) are coordinated and controlled through a coordination system that also includes a graphical user interface (GUI) for displaying source code and performance data and for enabling user interaction. Finally, as a result of performance-driven program development, an optimized distributed or parallel program is created by VFC.

The programs which are estimated by P^3T+ are based on the underlying compilation and programming model of VFC [3] which is a source-to-source parallelization system that translates Fortran90/HPF programs to Fortran90/MPI message-passing SPMD programs. Moreover, P^3T+ also models Fortran90 message-passing programs. The parallelization strategy of *VFC* is based on data decomposition in conjunction with the Single-Program-Multiple-Data (SPMD) programming model. With this method, 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 *VFC* through message passing.

P^3T+ currently supports mostly regular HPF programs which restricts array subscript expressions and loop bounds to linear functions of loop variables. Irregular codes with indirect array references (array subscript expressions contain array references) are excluded.

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 execution time. 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 also play an important role. Without making such information transparent, performance tuning is extremely difficult. P^3T+ at compile-time computes a set of performance parameters each of which reflects a different performance aspect. In the following all P^3T+ performance parameters are described.

3.1 Work Distribution

It is well known [7, 45, 43, 30, 41, 14, 42, 34, 25] that the work distribution has a strong influence on the cost/performance ratio of a parallel system. An uneven work distribution may lead to a significant reduction in a program's performance. Therefore, providing both programmer and parallelizing compiler with a work distribution parameter for parallel programs is critical to steer the selection of an efficient data distribution.

Two problems must be solved in order to compute the work distribution of a parallel program: first, how much work is contained in a program and second, how much work is being processed by every individual processor. We first consider these problems for loops and then extend our approach to full programs. Consider the following loop nest with a statement S included in a conditional statement.

```
    DO J1=1,N1
       DO J2=1,N2*J1
          IF (J1 <= N2) THEN
S :          A = A + ...
             ...
          ENDIF
       ENDDO
    ENDDO
```

Computing how many times S is executed is equivalent to counting the number of integer solutions of $\mathcal{I} = \{1 \leq J_1 \leq N_1,$ $1 \leq J_2 \leq N_2 * J_1,\ J_1 \leq N_2\}$. J_1 and J_2 are (loop) *variables* and N_1, N_2 are *parameters* (loop invariants). Note that we consider $J_2 \leq N_2 * J_1$ to be non-linear, although N_2 is loop invariant. The statement execution count for S is given by:

$$\sum_{J_1=1}^{min(N_1,N_2)} \sum_{J_2=1}^{N_2*J_1} 1 = \begin{cases} \frac{N_1^2*N_2}{2} + \frac{N_1*N_2}{2} & : \text{ if } 1 \leq N_1 \leq N_2 \\ \frac{N_2^3}{2} + \frac{N_2^2}{2} & : \text{ if } 1 \leq N_2 < N_1 \end{cases}$$

In general, every loop implies at least two constraints on its loop variable, one for its upper and one for its lower bound. Additional constraints on both parameters and variables can be implied, for instance, by conditional statements, minimum and maximum functions, data declarations, etc.

We briefly describe a symbolic algorithm which computes the number of integer solutions of a set of linear and non-linear constraints $\mathcal{I}$ defined over $\mathcal{V} \cup \mathcal{P}$ where $\mathcal{P}$ is the set of parameters and $\mathcal{V}$ the set of variables. Every $I \in \mathcal{I}$ is restricted to be of the following form:

$$p_1(\vec{P}) * v_1 + \ldots + p_k(\vec{P}) * v_k \;\; REL \;\; 0 \tag{1}$$

where REL $\in \{\leq, \geq, <, >, =, \neq\}$ represents an equality or inequality relationship. $\vec{P}$ is a vector defined over parameters of $\mathcal{P}$. $p_i(\vec{P})$ are linear or non-linear expressions over $\mathcal{P}$, whose operations can be addition, subtraction, multiplication, division, floor, ceiling, and exponentiation. Minimum and maximum functions are substituted where possible by constraints free of minimum and maximum functions.

SUM($\mathcal{I}$,$\mathcal{V}$,$\mathcal{P}$,E,$\mathcal{R}$)

- **INPUT:**
  $\mathcal{I}$: set of linear and non-linear constraints defined over $\mathcal{V} \cup \mathcal{P}$
  $\mathcal{V}$: set of variables
  $\mathcal{P}$: set of parameters
  E: symbolic expression defined over $\mathcal{V} \cup \mathcal{P}$
- **INPUT-OUTPUT:**
  $\mathcal{R}$: set of solution tuples $(\mathcal{C}_i, E_i)$ where $1 \le i \le k$. $\mathcal{C}_i$ is a conjunction of linear or non-linear constraints defined over $\mathcal{P}$. E_i is a linear or non-linear symbolic expression defined over $\mathcal{P}$.
- **ALGORITHM:**

S1: Simplify $\mathcal{I}$

S2: **if** $\mathcal{I}$ is inconsistent (no solution) **then**
  return
 endif

S3: **if** $\mathcal{V} = \phi$ **then**
  $\mathcal{R} := \mathcal{R} \cup \{\mathcal{I}, E\}$
  return
 endif

S4: Split $\mathcal{I}$

S4.1: Choose variable $v \in \mathcal{V}$ for being eliminated

S4.2: $\mathcal{I}''$:= subset of $\mathcal{I}$ not involving v
 $\mathcal{L}$=$\{l_1,...,l_a\}$:= set of lower bounds of v in $\mathcal{I}$
 $\mathcal{U}$=$\{u_1,...,u_b\}$:= set of upper bounds of v in $\mathcal{I}$
 a := cardinality of $\mathcal{L}$
 b := cardinality of $\mathcal{U}$

S4.3: **for each** $(l_i, u_j) \in \mathcal{L} \times \mathcal{U}$ **do**
 $\mathcal{I}'_{i,j} := \mathcal{I}'' \cup \{\, l_1 < l_i, .., l_{i-1} < l_i, l_{i+1} \le l_i, .., l_a \le l_i\}$
 $\cup \{u_1 > u_j, .., u_{j-1} > u_j, u_{j+1} \ge u_j, .., u_b \ge u_j\} \cup \{l_i \le u_j\}$
 $E_{i,j} := \sum_{v=l_i}^{u_j} E$
 SUM($\mathcal{I}'_{i,j}, \mathcal{V} - \{v\}, \mathcal{P}, E_{i,j}, \mathcal{R}$)
 endfor

S5: **return**

Figure 2. Symbolic sum algorithm for computing the number of solutions of a set of constraints $\mathcal{I}$

Figure 2 shows the algorithm for counting the number of solutions to a set of constraints, given $\mathcal{I}$ (set of constraints), $\mathcal{P}$, $\mathcal{V}$, E, and $\mathcal{R}$. E is an intermediate result (symbolic expression) for a specific solution E_i of the symbolic sum algorithm. The result $\mathcal{R}$ is a set of tuples $(\mathcal{C}_i, E_i)$ where $1 \le i \le k$. Each tuple corresponds to a conditional solution of the sum algorithm. Note that the conditions $\mathcal{C}$ (satisfying (1)) among all solution tuples are not necessarily disjoint. The result has to be interpreted as the sum over all E_i under the condition of $\mathcal{C}_i$ as follows:

$$\sum_{1 \le i \le k} \gamma(\mathcal{C}_i) * E_i \tag{2}$$

where γ is defined as

$$\gamma(\mathcal{C}) = \begin{cases} 1 & : \ \text{if } \mathcal{C} = \text{TRUE} \\ 0 & : \ \text{otherwise} \end{cases} \tag{3}$$

E and $\mathcal{R}$ must be respectively set to 1 and ϕ (empty set) at the initial call of the algorithm.

In each recursion the algorithm (see Figure 2) is eliminating one variable $v \in \mathcal{V}$. First, all lower and upper bounds of v in $\mathcal{I}$ are determined. Then the maximum lower and minimum upper bound of v are searched by generating disjoint subsets of constraints based on $\mathcal{I}$. For each such subset $\mathcal{I}'$, the algebraic sum of the current E over v is computed. Then the sum algorithm is recursively called for $\mathcal{I}'$, the newly computed E, $\mathcal{V}-\{v\}$, $\mathcal{P}$, and $\mathcal{R}$. Eventually at the deepest recursion level, $\mathcal{V}$ is empty, then E and its associated $\mathcal{I}$ represent one solution tuple defined solely over parameters. More details about this algorithm are given in [20].

In what follows we demonstrate how the symbolic sum algorithm can be used to determine the work contained in a loop nest as well as the work to be processed by a generic processor.

The following code shows a High Performance Fortran - HPF code excerpt with a processor array *PR* of size P.

```
      INTEGER A(N2)
      !HPF$ PROCESSORS :: PR(P)
      !HPF$ DISTRIBUTE (BLOCK) ONTO PR :: A
      DO J1=1,N1
         DO J2=1,J1 * N1
            IF ( J2 <= N2 ) THEN
S:             A(J2) = ...
            ENDIF
         ENDDO
      ENDDO
```

The loop nest contains a write operation to a one-dimensional array A which is block-distributed onto P processors. Let k ($1 \leq k \leq P$) denote a specific processor of the processor array. Computations that define the data elements owned by a processor k are performed exclusively by k. For the sake of simplicity we assume that P evenly divides N_2. Therefore, a processor k is executing the assignment to A based on the underlying block distribution if $\frac{N_2*(k-1)}{P}+1 \leq J_2 \leq \frac{N_2*k}{P}$. The precise work to be processed by a processor k is the number of times k is writing A, which is defined by *work(k)*.

The problem to estimate the amount of work to be done by processor k can now be formulated as counting the number of integer solutions to $\mathcal{I}$ which is given by:

$$\begin{array}{rcccl} 1 & \leq & J_1 & \leq & N_1 \\ 1 & \leq & J_2 & \leq & J_1 * N_1 \\ & & J_2 & \leq & N_2 \\ \frac{N_2*(k-1)}{P}+1 & \leq & J_2 & \leq & \frac{N_2*k}{P} \end{array} \tag{4}$$

In the following we substitute $\frac{N_2*(k-1)}{P}+1$ by LB and $\frac{N_2*k}{P}$ by UB.

By applying our algorithm we can automatically determine that statement S is approximately executed

$$work(k) = \sum_{1 \leq i \leq 3} \gamma(C_i) * E_i(k)$$

times by a specific processor k ($1 \leq k \leq P$) for the parameters $N_1,\ N_2$ and P. $\gamma(C_i)$ is defined by (3) and

- $C_1 = \{UB \leq N_1^2,\ P \leq N_2\}$ with $E_1(k) = \frac{(N_1+UB-LB)*(LB-2*N_1+2*LB*N_1+UB)}{2*N_1^2}$.
- $C_2 = \{\frac{UB}{N_1} > N_1,\ \frac{LB}{N_1} \leq N_1\}$ with $E_2(k) = (N_1 - \frac{LB}{N_1}+1)*(\frac{N_1^2}{2} - \frac{LB}{2}+1)$
- $C_3 = \{N_2 \geq P,\ N_1^2 \geq UB+1\}$ with $E_3(k) = \frac{N_2}{P}*(N_1 - \frac{UB+1}{N_1}+1)$

Note that by omitting the last two inequalities in (4), we can use the same symbolic sum algorithm to compute the overall work contained in the HPF code excerpt shown above.

Most conventional performance estimators must repeat the entire performance analysis whenever the problem size or the number of processors used are changing. However, our symbolic performance analysis provides the solution of the above problem as a symbolic expression of the program unknowns (P,N_1,N_2, and k). For each change in the value of any program unknown we simply re-evaluate the result, instead of repeating the entire performance analysis.

Having clarified the algorithm for computing how much work is contained in a program and how much work is being processed by every individual processor, we finally present the definitions for work distribution goodness of array assignment and loops, procedures and programs. Let S be an array assignment statement inside of a loop L, where A is the left hand-side array. P^A is the set of processors onto which A is distributed.

Definition 3.1 Optimal amount of work

The arithmetic mean: $owork(S) = work(S, P^A)/|P^A|$ *defines the optimal amount of work to be processed by every single processor in* P^A.

Based on the optimal amount of work a goodness function for the *useful work distribution* of an array assignment statement in a loop L is defined.

Definition 3.2 Useful work distribution goodness for an array assignment

The goodness of the useful work distribution with respect to an array assignment statement S *is defined by*

$$wd(S) = \frac{1}{owork(S)} \sqrt[2]{\frac{1}{|P^A|} \sum_{p \in P^A} \left(work(S,p) - owork(S)\right)^2}$$

The above formula is the standard deviation (σ) divided by the arithmetic mean ($owork(S)$), which is known as the variation coefficient in statistics [5]. In [17] we have presented a proof for the lower and upper bound of $wd(S)$ with the following result: $0 \leq wd(S) \leq |P^A| - 1$. Best-case and worst-case work distribution are, respectively, given by $wd(S) = 0$ and $wd(S) = |P^A| - 1$.

Based on Definition 3.2, a work distribution goodness function for loops, procedures, and programs can be defined.

Definition 3.3 Work distribution goodness for loops, procedures, and programs

Let E *be a loop, procedure or an entire program with* $\varrho(E)$ *the set of array assignment and procedure call statements in* E*, and* $freq(S)$ *is the execution time frequency of* S*, then the work distribution goodness for* E *is defined by:*

$$wd(E) = \sum_{S \in \varrho(E)} \frac{freq(S)}{\sum_{S' \in \varrho(E)} freq(S')} wd(S)$$

If S *represents a call to a procedure* E*, then* $wd(S) := wd(E)$.

3.2 Communication Parameters

The overhead to access nonlocal data from remote processors on distributed memory architectures is commonly orders of magnitude higher than the cost of accessing local data. P^3T+ estimates this critical performance aspect of a distributed or parallel program by simulating on a von Neumann architecture the associated communication behavior and computing the following performance parameters: the number of transfers (NT), the amount of data transfered (TD), and the overall communication time (TT). In this paper we describe how P^3T+ models communication caused by Fortran 90 array assignments in the context of regular data distributions. Predicting communication based on Fortran 77 array references has been described in detail in [18].

In what follows, we briefly sketch how *VFC* generates parallel code for Fortran 90 array assignments. Then, we outline the computation of the communication parameters for Fortran 90 array assignments based on a modified VFC runtime system and associated communication libraries

3.2.1 Modeling Fortran 90 Array Assignment Statements (*VFC*)

Distributed arrays, when referenced in a Fortran 90 array assignment statement, can introduce a considerable amount of communication, depending on the data distribution of the arrays involved in the assignment, access patterns implied by array subscripts, and problem and machine size chosen.

As shown in Figure 3, a parallel program generated by *VFC* contains calls to the *VFC* Run Time System (*RTS*) which manages distributed data structures (including redistribution of arrays) and provides an interface to communication libraries such as Adlib library [8]. A VFC generated parallel program contains calls to the *RTS* for any kind of communication. *RTS* requires allocation of a runtime descriptor (RD) for every array in a program. The RD is updated during runtime, for instance, when changing the shape of an array or its distribution. Let an array assignment statement S consist of a left-hand side array reference (LHS_ref) and several right-hand side array references (RHS_ref). VFC compiles Fortran 90 array assignment statements as follows:

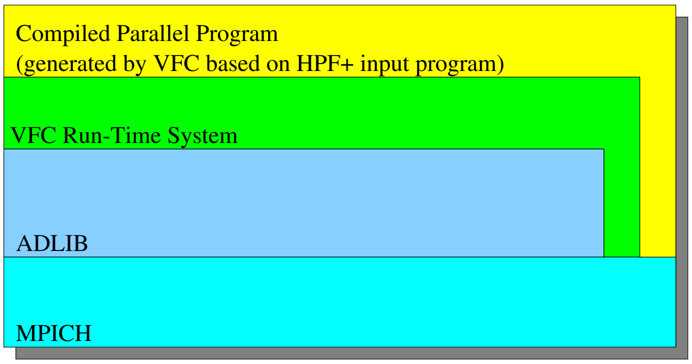

Figure 3. The structure of the compiled parallel program

1. For every array reference in S, a section descriptor (SD) is allocated and initialized. SD describes the array elements (specified by an array section with lower, upper bound and stride for every array dimension) that are touched by a given array reference.
2. Communication buffers are allocated for every different distributed RHS_ref of S
3. For every distributed RHS_ref of S, a call to a RTS routine is inserted with the following parameters: RD and SD of LHS_ref and RHS_ref, and a communication buffer of RHS_ref. The RTS routine is responsible to transfer non-local data to the communication buffer by invoking an Adlib library call which is implemented on top of MPI [26].
4. The subscript expressions of RHS_ref are modified so as references to non-local data are redirected to their associated communication buffers.

For further information on the parallelization of Fortran 90 array assignments in *VFC*, the reader may refer to [3].

3.2.2 Modifying VFC *RTS* and Adlib library

P^3T+ estimates the communication behavior of VFC generated distributed or parallel programs by simulating the behavior of the VFC RTS and associated Adlib library calls on a von Neumann architecture. This means that at compile-time P^3T+ partially executes calls to the VFC RTS and Adlib library for every processor suppressing any actual communication. Only those code sections that compute the sending processor and size of messages are executed. This is achieved by integrating P^3T+ with a modified VFC RTS and Adlib library (see Figure 4) and executing them for every processor of the parallel program at compile-time on a von Neumann architecture.

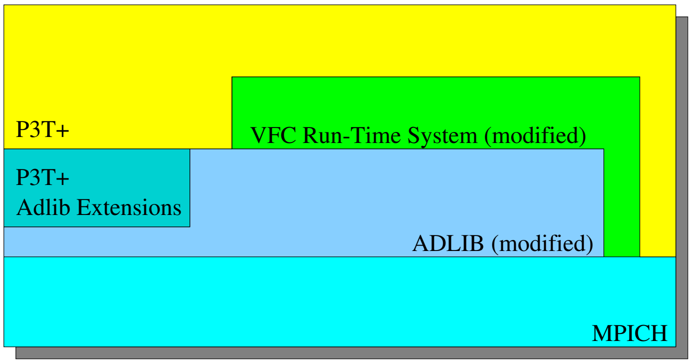

Figure 4. Modified VFC RTS and Adlib library as part of P^3T+.

In *RTS* we suppressed initialization code where the number of processors available on a given parallel architecture is compared with the number of processors requested by the parallel program. The Adlib library has been modified as follows:

- Three global variables have been introduced: *NoOfProcessors* holds the number of processors onto which the parallel program is being executed (as defined by the HPF PROCESSORS directive). *CurrentProcessor* ($1 \leq$ CurrentProcessor $\leq$ *NoOfProcessors*) defines the identification of the current processor that is being simulated by P^3T+ . *CurrentLineNumber* holds the line number of the currently analyzed source code line.
- All calls to functions MPI_COMM_SIZE and MPI_COMM_RANK are, respectively, replaced with a reference to NOOFPROCESSORS and CURRENTPROCESSOR.
- A new data structure – P3T_COMM_SEQUENCE – is introduced which records all SEND operations of a unique statement S. Every entry in P3T_COMM_SEQUENCE holds information about a unique SEND operation by specifying the size of the message in bytes, the sending processor, and the number of the currently analyzed source code line.
- All $send$ operations are suppressed except computation of their parameters which are used to update P3T_COMM_SEQUENCE.
- All $receive$ and $wait$ operations are suppressed.

We use a preprocessor together with conditional code in VFC RTS and Adlib library thus both VFC and P^3T+ can use the same sources. The conditional code is only activated for P^3T+ .

3.2.3 Computing P^3T+ Communication Parameters

In order to estimate the communication behavior of all Fortran 90 array assignments in a parallel program, P^3T+ proceeds as follows:

1. Invoke VFC to generate message passing code based on input program.
2. Traverse VFC generated message passing code and execute pre-compiled communication code – based on modified VFC RTS and Adlib library – for every call to a communication routine R of VFC RTS.
   (a) Update P3T_COMM_SEQUENCE for every processor p that executes R.
3. Compute communication parameter for all code regions (e.g. statements, loops, procedures, and program) of interest based on P3T_COMM_SEQUENCE entries.

Some of the input parameters of RTS and Adlib library calls may require user interaction. For instance, in order to determine number of processors and program unknowns appearing in array subscript expressions or loop bounds, the user may be requested for realistic values.

Definition 3.4 Communication parameters for an array assignment

Let $\mathcal{S}$ denote the set of array assignments in a program and $\mathcal{F}(S)$ the set of procedures referenced within a statement $S \in \mathcal{S}$. Furthermore, let $\mathcal{K}(S)$ denote the set of communication records (stored in P3T_COMM_SEQUENCE*) associated with S. Then the number of transfers $ntS(S)$, the amount of data transfered $tdS(S)$, and the transfer time $ttS(S)$ for S statement are defined as*

$$ntS(S) = freq(S) * \left(|\mathcal{K}(S)| + \sum_{q \in \mathcal{F}(S)} \frac{ntE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

$$tdS(S) = freq(S) * \left(\sum_{k \in \mathcal{K}(S)} data(k) + \sum_{q \in \mathcal{F}(S)} \frac{tdE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

$$ttS(S) = freq(S) * \left(\sum_{k \in \mathcal{K}(S)} (\alpha + data(k) * \beta) + \sum_{q \in \mathcal{F}(S)} \frac{ttE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

where $freq(S)$ is the execution frequency of S, $calls(q)$ denotes the set of statements calling procedure q in the program, $ntE(q)$ denotes the overall number of transfers for a procedure q, $card(q, S)$ is the number of calls to procedure q in S, $data(k)$ is the amount of data (in bytes) transfered by a message k, and α and β, both measured on the target machine, denote the message startup time and the transfer time per message byte respectively.

The *nesting level* of a statement S is defined as the number of loops enclosing that statement. If S is not enclosed in a loop then S has loop nesting level 0.

Definition 3.5 Communication parameters for a loop nest

Let L denote a loop at the nesting level i, $\mathcal{S}_L$ the set of all statements (excluding nested loop statements and their bodies) appearing in the body of L. Furthermore, let $\mathcal{L}_L$ denote the set of all loops at the nesting level $i+1$, occurring in the body of L. Then the number of transfers $ntL(L)$, the amount of transfered data $tdL(L)$, and the transfer time $ttL(L)$ for L are recursively defined as

$$\begin{aligned} ntL(L) &= \sum_{s \in \mathcal{S}_L} ntS(s) + \sum_{l \in \mathcal{L}_L} ntL(l) \\ tdL(L) &= \sum_{s \in \mathcal{S}_L} tdS(s) + \sum_{l \in \mathcal{L}_L} tdL(l) \\ ttL(L) &= \sum_{s \in \mathcal{S}_L} ttS(s) + \sum_{l \in \mathcal{L}_L} ttL(l) \end{aligned}$$

Definition 3.6 Communication parameters for a procedure or a program

Let E be a procedure or an entire program, $\mathcal{L}_E$ the set of loop nests with nesting level 0 (correspond to loop nests without enclosing loop) in E. Furthermore, let $\mathcal{S}_E$ denote the set of statements (excluding loop nests) in E, outside of loops. Then, number of transfers $ntE(E)$, amount of transfered data $tdE(E)$, and transfer time $ttE(E)$ implied by all statements $S \in \mathcal{S}_E$ and loop nests $L \in \mathcal{L}_E$, are defined as

$$\begin{aligned} ntE(E) &= \sum_{s \in \mathcal{S}_E} ntS(s) + \sum_{l \in \mathcal{L}_E} ntL(l) \\ tdE(E) &= \sum_{s \in \mathcal{S}_E} tdS(s) + \sum_{l \in \mathcal{L}_E} tdL(l) \\ ttE(E) &= \sum_{s \in \mathcal{S}_E} ttS(s) + \sum_{l \in \mathcal{L}_E} ttL(l) \end{aligned}$$

3.3 Computation Times

The computation time parameter reflects the time required by a processor to execute local computations of the program excluding communication. By local computations we mean those computations assigned to a processor according to the SPMD programming model and the "owner computes paradigm" (see Section 3). This parameter can be useful to

- analyze the communication/computation relationship by incorporating also communication parameters described in Section 3.2
- evaluate whether there is enough computation contained in a code region, thus parallelizing the code region may be effective.
- identify the most time-consuming code regions of the program (*hot spots*) which are often hard to isolate without the help of a profiling tool.

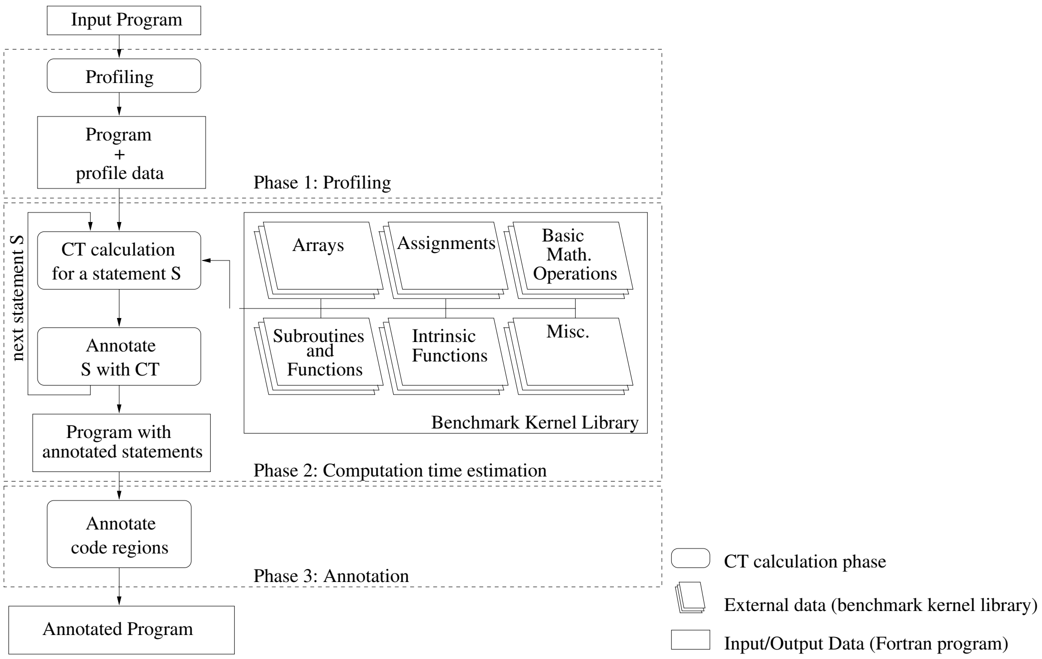

Figure 5. Estimating computation times under P^3T+

Our method for predicting computation times employs statement execution frequencies and branching probabilities as well as pre-measured kernel codes. Pre-measured kernel codes are used to associate statements and small code sections in the input program with pre-measured execution times for a specific target machine. A large set of kernel codes are pre-measured for every target machine of interest and stored in a benchmark kernel library.

Figure 5 shows the architecture of the CT parameter. Given the Fortran program and the profiling information for a specific set of input data, the computation time parameter is estimated for each statement separately by pattern matching against pre-measured kernels stored in the benchmark kernel library.

In what follows we describe the set of kernels upon which our techniques are based on. Then, we will discuss the training phase of the benchmark kernel library which measures all kernels once for every different target machine of interest. Finally, we describe how to estimate computation times based on pre-measured kernels and profiling data.

3.3.1 Benchmark Kernel Library

The benchmark kernels of the computation time parameter can be classified as follows:

1. **Assignments**
   Scalar assignment operations considering several cases where the data types of left-hand side and right-hand side scalars are identical or different. Different data types may cause additional overhead due to type conversion.

2. **Basic mathematical operations**
   Basic mathematical operations, such as +, -, *, /, **.

3. **Procedures (subroutines and functions)**
   Subroutine call and function reference overheads for varying numbers of parameters.

4. **Intrinsic functions**
   Standard intrinsic functions, like SIN, COS, MOD, LOG, etc. and implicit reduction functions included in Fortran such as MIN, MAX, SUM, and INDEX.

5. **Arrays**
   Kernels for array reference address calculations.

6. **Miscellaneous**
   All other kernels comprising, for instance, boolean operations, IF-THEN-ELSE constructs, loop headers, etc.

3.3.2 Training Phase

The performance estimator has to be trained once for all different target machines of interest in order to determine computation times for each different kernel in the kernel library. This is achieved by a training phase. Primitive statements and most primitive operations - except array operations - are measured for different data types and stored in the benchmark kernel library as numeric values. Computation times for array operations and intrinsic functions depend not only on the data types involved but also on the data size of arrays and the number of parameters which are considered by the measurements. Based on these measurements a set of functions describing the computation times for different access patterns, data types, and problem sizes is constructed using the chi-square fit method and stored in the benchmark kernel library.

3.3.3 Estimating computation times

Obtaining computation times for a program essentially involves 3 phases as shown in Figure 5.

1. The input program is instrumented and executed once on a von Neumann architecture. The profile data is used to annotate the program with execution frequencies and branching probabilities.
2. For every statement, a kernel pattern matching in combination with a performance evaluation algorithm is invoked. Primitive operations, primitive statements and intrinsic functions are simply detected by their syntax tree node representation. The computation times for every statement are then weighted by their execution frequencies or branching probabilities (in case of conditional statements) which yields the overall execution time for a statement. Every statement is annotated with the estimated computation time as obtained from this phase.
3. Estimated computation times for larger code regions (e.g. loops, procedures, and programs) are obtained by summing up the corresponding computation times of all statements in this region. Larger code patterns (e.g. matrix multiplication) may require more advanced pattern matching techniques such as those mentioned in [12]. The current implementation of our pattern matcher handles all kernels in the benchmark kernel library except code patterns. The output of phase 3 is the program annotated with computation times for all code regions.

In the following, we define the computation time for a single statement, loop nest, procedure and entire program.

Definition 3.7 Computation time for a program statement

Let $\mathcal{S}$ denote the set of statements of a program and $\mathcal{F}(S)$ the set of procedures referenced within a statement $S \in \mathcal{S}$, then the accumulated time $ctS(S)$ for this statement is defined as

$$ctS(S) = freq(S) * \left(ctS_{simple}(S) + \sum_{q \in \mathcal{F}(S)} \frac{ctE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

where $ctS_{simple}(S)$ denotes the computation time required by the single instantiation of S excluding the computation time required by any procedures referenced in that statement. The set of statements referencing procedure q in program is given by $calls(q)$. $ctE(q)$ denotes the overall computation time of a procedure q, $freq(S)$ the execution frequency of the statement S, and $card(q,S)$ the number of references to procedure q in S.

Definition 3.8 Computation time for a loop nest

Let L denote a loop at the nesting level i, $\mathcal{S}_L$ the set of all statements (excluding nested loop statements and their bodies) appearing in the body of L. Further, let $\mathcal{L}_L$ denote the set of all do loops at the nesting level $i+1$ occurring in the body of L. Then the computation time for L denoted by $ctL(L)$ is defined as

$$ctL(L) = \sum_{s \in \mathcal{S}_L} ctS(s) + \sum_{l \in \mathcal{L}_L} ctL(l)$$

Definition 3.9 Computation time for a procedure or a program

Let E be a procedure or an entire program and $\mathcal{L}_E$ the set of loop nests with nesting level 0 in E. Further, let $\mathcal{S}_E$ denote the set of statements in E outside of loops. Then the accumulated computation time $ctE(E)$, implied by all statements $S \in \mathcal{S}_E$ and loop nests $L \in \mathcal{L}_E$, is defined as

$$ctE(E) = \sum_{s \in \mathcal{S}_E} ctS(s) + \sum_{l \in \mathcal{L}_E} ctL(l)$$

3.4 Number of Cache Misses

It is well known [19, 47, 36, 33, 24] that inefficient memory access patterns and data mapping into the memory hierarchy (data locality problem) of a single processor cause major program performance degradation. P^3T+ estimates the number of accessed cache lines which correlates with the number of cache misses. This parameter is derived for loops, procedures, and entire programs.

The main idea of our estimation approach for cache misses is that array references are grouped into array access classes such that all arrays in a specific class exploit reuse of array elements in the same set of array dimensions. The definition of array access classes is based on a specific number of innermost loops of a not necessarily perfectly nested loop L. Two array references are in the same array access class for a loop nest if they actually access some common memory location in the same array dimensions and reuse occurs in L. The common accesses occur on either the same or a different iteration of L.

In the following we define the number of cache misses for a loop nest, procedure, and entire program.

Definition 3.10 Number of cache misses for a loop nest

Let P define the set of processors executing the loop nest L and $\mathcal{F}(L)$ the set of procedures referenced within L. Furthermore, let $cmL^p(L)$ define the number of cache misses induced by a single instantiation of L with respect to a processor $p \in P$, excluding the cache misses implied by procedure calls within L. Then the overall number of cache misses induced by L with respect to all processors in P is defined as

$$cmL(L) = freq(L) * \left(\frac{1}{|P|} \sum_{p \in P} cmL^p(L) + \sum_{q \in \mathcal{F}(L)} \frac{cmE(q) * card(q, L)}{\sum_{S \in calls(q)} freq(S) * card(q, S)} \right)$$

where $calls(q)$ denotes the set of statements calling procedure q in the program, $cmE(q)$ denotes the accumulated number of cache misses implied by procedure q (see Definition 3.11), $freq(S)$ and $freq(L)$ denote the execution frequency of statement S and loop nest L respectively, and $card(q, S)$ is the number of calls to procedure q in S.

The first sum in Definition 3.10 describes the mean value of cache misses implied by a single instantiation of L across all processors in P executing L. The second sum is explained as follows: in order to take procedure calls into account, the parameter outcome for a single procedure call instantiation is supposed to be independent of the call site. This means that the parameter outcome at a particular call site is the same as the parameter outcome of the procedure over all call sites, which is a common assumption made for performance estimators. The estimated number of cache misses for every specific loop is weighted by its execution count ($freq$) in order to reflect its impact on the overall program performance.

All call graphs are supposed to be acyclic. Note that Definition 3.10 is also applicable to a sequential program iff $|P| = 1$.

Extending the cache cost function to a procedure or a program is straight forward:

Definition 3.11 Number of cache misses for a procedure or a program

Let E be a procedure or an entire program, and $\mathcal{L}_E$ and $\mathcal{S}_E$, respectively, denote the set of loop nests and statements with procedure calls at nesting level 0 in E. Furthermore, let $\mathcal{F}(S)$ denote the set of procedures referenced within a statement $S \in \mathcal{S}$. Then the number of cache misses induced by all loop nests $L \in \mathcal{L}_E$ and statements $S \in \mathcal{S}_E$ is defined as follows:

$$cmE(E) = \sum_{l \in \mathcal{L}_E} cmL(l) + \sum_{S \in \mathcal{S}_E} \sum_{q \in \mathcal{F}(S)} \frac{cmE(q) * card(q, S)}{\sum_{S' \in calls(q)} freq(S') * card(q, S')}$$

where $calls(q)$, $freq(S)$ and $card(q, S)$ are defined as in Definition 3.10.

The first sum in Definition 3.11 corresponds to the loops contained in E. The second sum models procedure calls outside of loops in E. It is assumed that the same cache behavior is implied by every instantiation of L. A more accurate modeling of cmE requires separate values regarding $freq(L)$ for every instantiation of L at the price of a considerably larger computational effort.

More details about our cache modeling approach can be found in [17].

4 Experiments

P^3T+ has been implemented and is currently used to support development of parallel and distributed programs. P^3T+ is primarily used to guide the selection of profitable data distributions and program transformations under VFC. Note that the programmer can specify data data distributions (e.g. through HPF directives) and select transformations under VFC. The compiler automatically compiles HPF directives and applies transformations to the program.

In order to demonstrate the usefulness of P^3T+ we present three different experiments on two different target machines. First, Cholesky factorization – a code for factoring a *n x n* symmetric positive-definite matrix into the product of a lower triangular matrix and its transpose – is used to examine the accuracy of P^3T+'s computation time, work distribution and cache misses parameters on a Meiko CS-2 and a NEC Cenju-4 multiprocessor system. The performance outcome of various code versions and data distributions is compared by using $P^3T+$and second, an application about pricing of derivate products which is an important field in finance theory, is evaluated. Among others, we examine the accuracy of P^3T+ for predicting execution times of important parallel reduction operations. Third, we apply P^3T+ to WIEN97 which is a code for quantum mechanical calculations of solids. We compare predicted against measured performance parameters for number of transfers, amount of data transferred, transfer times, and work distribution for changing problem and machine sizes.

4.1 Cholesky Factorization

Cholesky factorization [9] factors a *n x n*, symmetric, positive-definite matrix into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^TU$, where U is upper triangular). It is assumed that the lower triangular portion of A is stored in the lower triangle of a two-dimensional array and that the computed elements of L overwrite the given elements of A. Cholesky factorization is a key kernel used by the material science code (see Section 4.3) which has a significant impact on the performance of this code.

The following code excerpt shows the main portion of a Cholesky factorization:

```
      ...
      DOUBLE PRECISION :: A(N,N)
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (CYCLIC,*) ONTO PR :: A
      ...
      A = 2*N
      DO 10 I=1,N
          A(I,I) = SQRT(A(I,I))
          A(I+1:N,I)=A(I+1:N,I)/A(I,I)
          DO 20 K=I+1,N
              DO 20 J=I+1,N
                  IF (K .GE. J) THEN
                      A(K,J)=A(K,J)-A(K,I)*A(J,I)
                  ENDIF
20          CONTINUE
10     CONTINUE
       ...
```

We have used P^3T+ to predict several performance parameters of Cholesky factorization. Figure 6 shows the predicted and measured computation times of the Cholesky factorization code on a single processor of the Meiko CS-2 distributed memory multiprocessor system. Note that the computation time parameter refers to the sequential computation time overhead. In the worst case predicted computation times are off the measured values by 10 %.

Figure 7 shows estimated and measured work distribution values for two different parallel versions of the Cholesky factorization (BLOCK and CYCLIC distribution [27] of the first dimension of array A) that has been executed on 16 processors. For BLOCK distribution, the predicted work distribution values are off the measured results in the worst case by 0.6 %. The Cholesky factorization based on CYCLIC distribution can yield estimation errors of up to 35 % for very small problem sizes (N) due to inaccurate division operations in our symbolic sum algorithm. However, for increasing problem sizes, the estimation error is almost negligible (less than 1 %).

Figure 8 displays the estimated cache misses as obtained by P^3T+ for various problem and machine sizes, and loop nestings of the Cholesky factorization. We tested two different loop nestings: first, K-loop is the outer and J-loop the inner loop. Second, J-loop is the outer and K-loop the inner loop. In the first case, three array references are traversed in the first dimension by the K-loop which results in a better cache behavior than the second case due to the column-major storage layout of Fortran. Increasing the number of processors also increases the available cache memory which in turn improves the overall cache performance. P^3T+ clearly detects these effects as shown by Figure 8. It is very difficult to measure the number of cache misses without hardware support. For this reason, we measured the corresponding execution times (see Figure 9) for each code version of Figure 8. The performance ranking are identical for each different code version both in terms of predicted cache behavior as well as measured execution times. Although the code versions display only a rather small difference in execution times, we believe that it is the cache behavior that causes the differences which is correctly modeled by P^3T+.

4.2 Pricing of Financial Derivatives

In this experiment we apply P^3T+ to an application about pricing of derivate products which is an important field in finance theory. A *derivative* (or *derivative security*) is a financial instrument whose value depends on other, so called underlying securities [28]. Examples are stock options and variable coupon bonds, the latter paying interest rate dependent coupons. The pricing problem can be stated as follows: what is the price today of an instrument which will pay some cash flows in the future, depending on the development of an underlying security, e.g. stock prices or interest rates? For simple cases analytical formulas are available, but for a range of products, whose cash flows depend on a value of a financial variable in the past - so called *path dependent* products - Monte Carlo simulation techniques have to be applied [39, 35]. By utilizing massively parallel architectures very efficient implementations can be achieved [49].

The parallel pricing system has been encoded as an HPF program [13] by the group of Prof. Dockner, Department of Business Administration, University of Vienna. This program comprises approximately 1000 lines of code. This program has been executed on the NEC Cenju-4 [38] distributed memory multiprocessor system:

```
...
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (BLOCK) ONTO PR :: VALUE
    ...
   TYPE(BOND) :: B                                              ! the bond to be priced
   REAL(DBLE) :: VALUE(1:N)                                     ! all path results
   REAL(DBLE) :: PRICE

   !HPF$ INDEPENDENT, REDUCTION(PRICE), ON HOME(VALUE(I))
   DO I = 1, N
      VALUE(I) = DISCOUNT(0,CASH_FLOW(B,1,N),FACTORS_AT(RANDOM_PATH(0,0,N)))
      PRICE = PRICE + VALUE(I)                                  ! reduction over PRICE
   END DO
   PRICE = PRICE/N                                              ! mean value
    ...
```

Array VALUE has been block-wise distributed onto the maximum number of processors – by using the HPF intrinsic function NUMBER_OF_PROCESSORS() – that are available on a given architecture. The HPF DO-Independent directive specifies that each iteration of the main simulation loop can be executed simultaneously. Every iteration of the simulation loop is executed by the processor that owns array element VALUE(I) based on the owner-computes paradigm [3]. The summation of the path results over variable PRICE has been realized by an HPF reduction directive which is compiled to an efficient machine specific function.

We first used P^3T+ to predict the work distribution of this code based on BLOCK distribution for 16 processors. Figure 10 shows the estimated and measured work distribution for various problem sizes N. It clearly shows that we achieve a best-case work distribution (WD= 0.0). The estimates are identical with measurements.

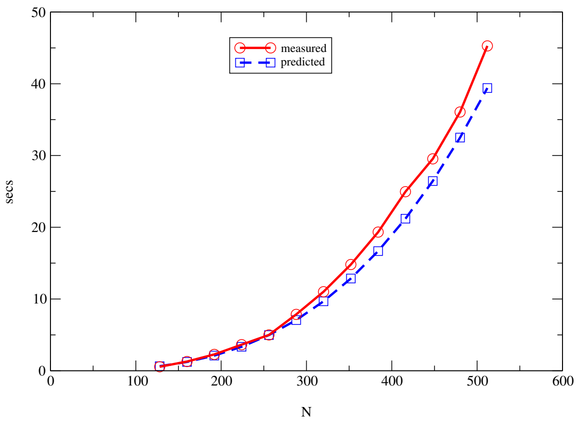

Figure 6. Measured versus predicted computation times of the Cholesky factorization for various problem sizes (N) on the Meiko CS-2

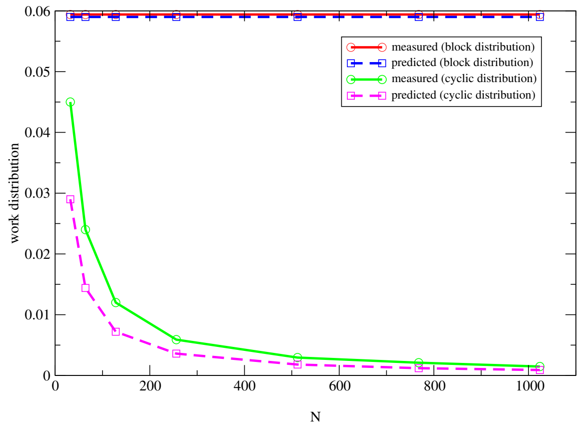

Figure 7. Measured versus predicted work distribution of the Cholesky factorization on 16 processors for various problem sizes (N) and data distributions (BLOCK and CYCLIC).

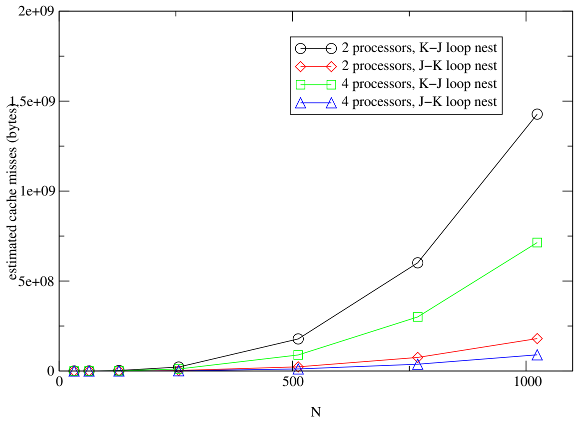

Figure 8. Predicted number of cache misses of the Cholesky factorization for various machine and problem sizes, and loop nestings on a NEC Cenju-4

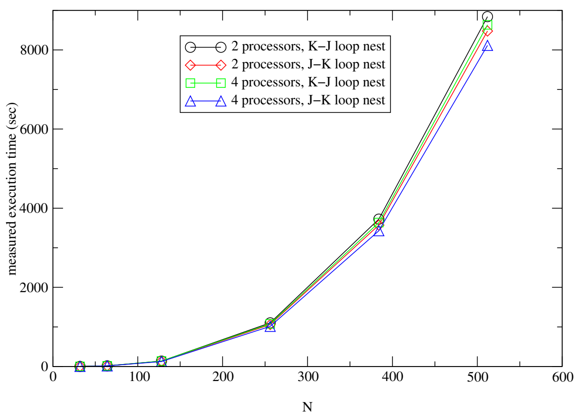

Figure 9. Measured execution times for the Cholesky factorization for various machine and problem sizes, and loop nestings on a NEC Cenju-4

Furthermore, we used P^3T+ to predict the execution time behavior of the SUM reduction operation of this code. Figures 11 and 12, respectively, show the measured and predicted execution times for the reduction operation for various problem (size of reduction data) and machine sizes on a NEC Cenju-4. Note that although our computation time parameter is restricted to local computations excluding communication, for this experiment, we employed the same technique (pre-measured kernel codes) to predict the execution time of an an explicitly parallel reduction operation. The given reduction operation (SUM) is defined over a scalar (double precision variable PRICE). However, our implementation is more general and covers reductions for both scalars as well as replicated arrays. Figure 13 displays the estimation error rate (all predictions are within 12 % of the real results) which is given by the following formula:

$$\frac{|\text{measured value} - \text{predicted value}|}{\text{measured value}} \tag{5}$$

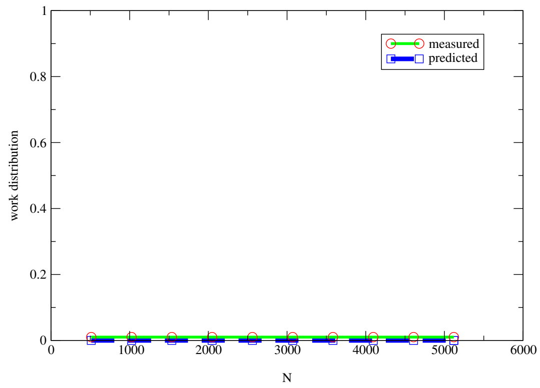

Figure 10. Estimated versus predicted work distribution of the pricing code on 16 processors for various problem sizes N

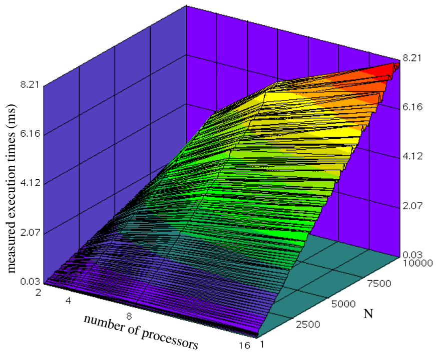

Figure 11. Measured execution times of a summation reduction operation based on replicated data for varying number of processors and reduction data size (N) on the NEC Cenju-4.

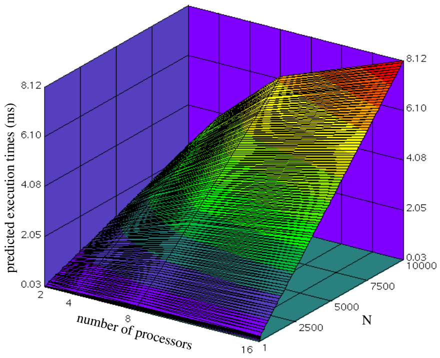

Figure 12. Predicted execution times of a summation reduction operation based on replicated data for varying number of processors and reduction data size (N) on the NEC Cenju-4.

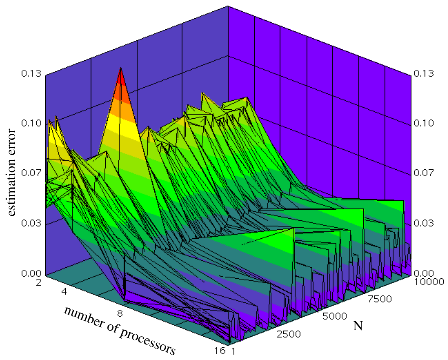

Figure 13. Estimation error rate of a summation reduction operation based on replicated data for varying number of processors and reduction data size (N) on the NEC Cenju-4.

4.3 Quantum Mechanical Calculations of Solids

In our final experiment we applied P^3T+ to WIEN97 [4] which is a system for the calculation of the electronic structure of solids that is being used by several 100 institutions world-wide. P^3T+ has been employed to predict the performance behavior of HNS that comprises 500 lines of code and is a core routine of WIEN97. HNS defines a symmetric (hermitian) matrix (the Hamiltonian). Radial and angular dependent contributions are pre-computed 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.

The following code shows the main loop nests of the HNS code based on HPF/Fortran 90 array operations:

```
...
!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
...
```

The array operations are executed in parallel based on the owner-computes-paradigm and the HPF distribution directives [27]. Arrays are mapped onto the maximum number of processors (HPF intrinsic function NUMBER_OF_PROCESSORS) that are available on a given architecture.

Figures 14 displays the time needed to compute all P^3T+ performance parameters for a specific problem size and for varying machine sizes on a Sun Ultra 10 workstation. The timings do not include the profile rune of the original sequential HNS program to determine execution frequencies and branching probabilities. Note that the time needed to compute any performance parameter is invariant with respect to the problem size of a given program.

Figures 15 - 17 show the predicted and measured values for the P^3T+ parameters: number of transfers, amount of data transferred, and transfer times. The experiments have been conducted for various number of processors and problem sizes on a NEC Cenju-4 machine. Note that for small problem and machine sizes the estimation errors are almost negligible whereas for larger problem and machine sizes (more than 8 processors) the estimation errors can be more severe. P^3T+ replaces (only in its performance model not in the actual code) I in H(I,1:I) by $N/2$ (determined by the enclosing loop bound and array access pattern) in the HNS code. This assumption results in a decrease of the predicted number of transfers for larger machine sizes. In reality more processors are involved in the communication which causes higher number of transfers. The predicted amount of data transferred is very close to the measured values (see Figure 16). As transfer times are influenced by both number of transfers and amount of data transferred, we observe a slightly better estimation accuracy than for number of transfers.

Figure 18 and 19, respectively, show the estimated and measured work distribution behavior of HNS based on BLOCK and CYCLIC distribution of H in the second dimension. It can be clearly seen that CYCLIC distribution outperforms BLOCK distribution due to the triangular loop iteration space of the HNS loop nests. The estimation errors for CYCLIC distribution are mostly due inaccurate division operations in our symbolic sum algorithm. All experiments for the HNS code have been conducted for problem sizes up to N=497 which reaches the memory capacity of the target architecture.

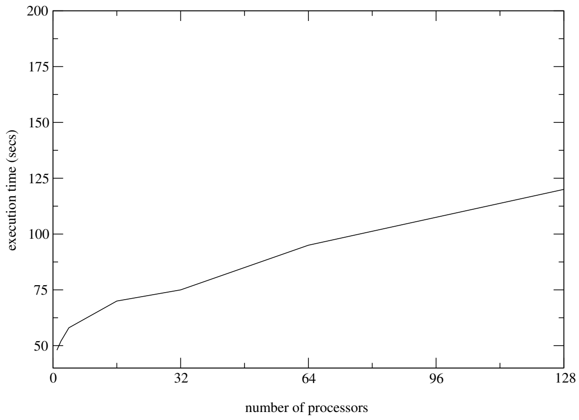

Figure 14. Execution times to obtain all P^3T+ performance parameters for HNS with varying machine sizes.

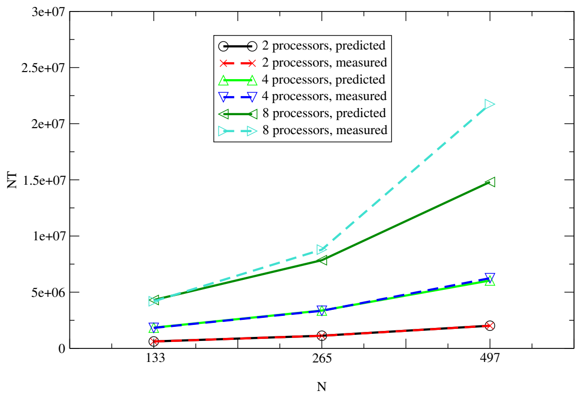

Figure 15. Measured versus predicted number of transfers of HNS for various problem (N) and machines sizes on a NEC Cenju-4.

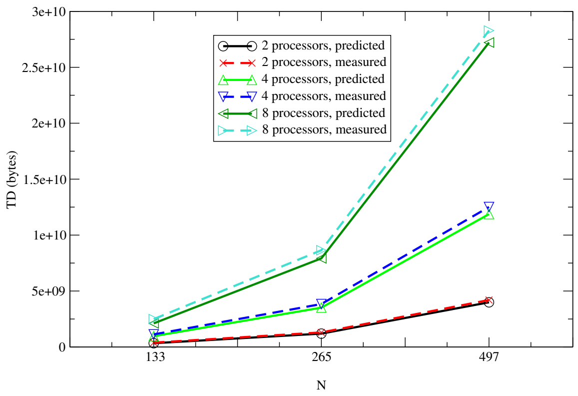

Figure 16. Measured versus predicted amount of data transferred (in bytes) of HNS for various problem (N) and machines sizes on a NEC Cenju-4.

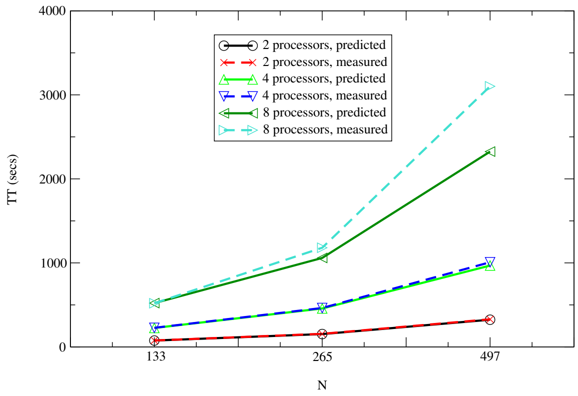

Figure 17. Measured versus predicted transfer time (secs) of HNS for various problem (N) and machines sizes on a NEC Cenju-4.

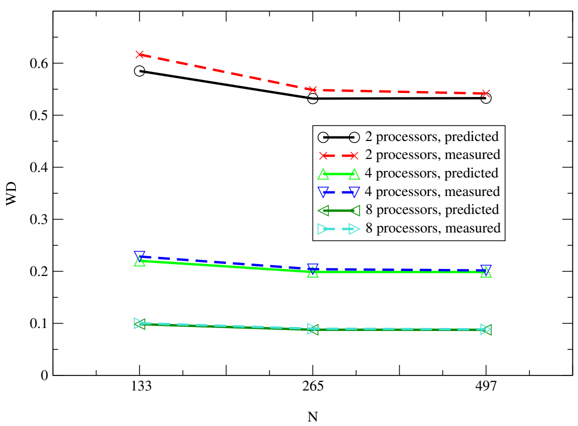

Figure 18. Measured versus predicted work distribution values of HNS for BLOCK distribution and various problem (N) and machine sizes.

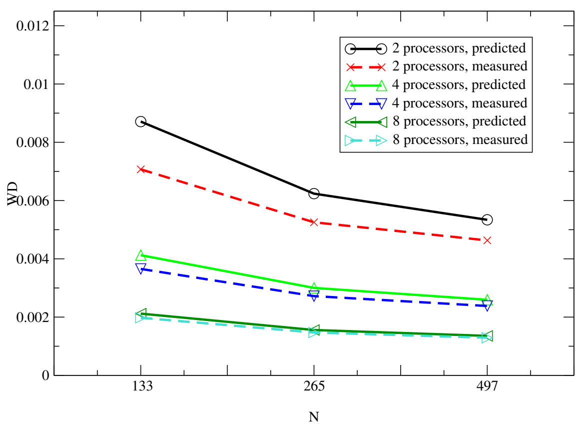

Figure 19. Measured versus predicted work distribution values of HNS for CYCLIC distribution and various problem (N) and machine sizes.

5 Conclusions

In this paper, we have described P^3T+, a performance prediction tool for parallel and distributed programs. P^3T+ is closely integrated with a parallelizing compiler (VFC) and thus supports the programmer during development of parallel programs under this compiler. Traditionally, the quality of performance prediction has been hampered by modeling either programs or architectures with good accuracy but not both of them. Moreover, very few performance estimators actually consider code transformations and optimizations applied by a compiler.

In contrast to most other performance estimators P^3T+ models programs, code transformations, and parallel and distributed architectures. The transformations and optimizations are selected by the programmer and automatically performed by VFC. At any stage in the parallelization effort P^3T+ can be invoked to examine the performance of a given code version in the parallelization search space of code versions generated by VFC. P^3T+ computes a variety of performance parameters including work distribution, number of transfers, amount of data transferred, transfer times, computation times, and number of cache misses. P^3T+ supports the programmer in finding efficient code transformations and optimizations by comparing different code versions with respect to the outcome of the performance parameters.

Several novel technologies are employed to compute these parameters: loop iteration spaces, array access patterns, and data distributions are modeled by employing highly effective symbolic analysis. Communication is estimated by simulating the behavior of a communication library used by the underlying compiler. Computation times are predicted through pre-measured kernels on every target architecture of interest. We carefully model most critical architecture specific factors such as cache lines sizes, number of cache lines available, startup times, message transfer time per byte. P^3T+ has been implemented and is currently evaluated by several application developers. Experimental results with realistic kernel codes taken from real-world applications demonstrate the accuracy and usefulness of P^3T+.

Various open issue will be followed by future work. We want to extend P^3T+ by extensive symbolic analysis to handle programs with unknowns, irregular applications, and linear and non-linear symbolic expressions. Performance information should be provided as functions over program unknowns for all performance parameters. Moreover, symbolic evaluation should be employed to aggressively collect constraints about program unknowns throughout a program. We also plan to extend P^3T+ covering object-oriented multi-threaded programs which exploit both data and task parallelism. Finally, we examine approaches to describe parallel programs at a higher level than specific programming languages which should alleviate portability and reusability of P^3T+ for various other transformation and development systems.

References

[1] E. L. A. Espinosa, T. Margalef. Automatic Performance Evaluation of Parallel Programs. In *IEEE Proc. of the 6th Euromicro Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, January 1998.

[2] S. Benkner. HPF+: High Performance Fortran for advanced industrial applications. *Lecture Notes in Computer Science*, 1401, 1998.

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

[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] J. Bleymüller, G. Gehlert, and H. Gülicher. *Statistik für Wirtschaftswissenschaftler*. Verlag Vahlen, München, 1985. WiSt Studienkurs.

[6] J. Brehm, M. Madhukar, E. Smirni, and L. Dowdy. PerPreT — A performance prediction tool for massively parallel systems. *Lecture Notes in Computer Science*, 977, 1995.

[7] B. Carlson, T. Wagner, L. Dowdy, and P. Worley. Speedup properties of phases in the execution profile of distributed parallel programs. In *Computer Performance Evaluation '92: Modeling Techniques and Tools*, pages 83–95, 1992. (Ed.) R. Pooley and J. Hillston.

[8] 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.

[9] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. Report ORNL/TM-12470, Oak Ridge National Laboratory, Oak Ridge, TN, 1994. LAPACK Working Note 80.

[10] M. Clement and M. Quinn. Symbolic Performance Prediction of Scalable Parallel Programs. In *Proc. of 9th International Parallel Processing Symposium*, St. Barbara, CA, April 1995.

[11] M. J. Clement and M. J. Quinn. Dynamic performance prediction for scalable parallel computing. Technical Report 95-80-04, Oregon State University.

[12] B. DiMartino. Algorithmic Concept Recognition Support for Automatic Parallelization: A Case Study for Loop Optimization and Parallelization. *Journal of Information Science and Engineering, Special Issue on Compiler Techniques for High-Performance Computing*, to appear in March 1998.

[13] 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.

[14] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408 – 423, March 1989.

[15] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. Technical Report CS1999-0619, University of California, San Diego, Computer Science and Engineering, May 18, 1999.

[16] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11):47 – 56, November 1995.

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

[18] T. Fahringer. Compile-Time Estimation of Communication Costs for Data Parallel Programs. *Journal of Parallel and Distributed Computing, Academic Press*, 39(1):46–65, Nov. 1996.

[19] T. Fahringer. Estimating cache performance for sequential and data parallel programs. In *Proc. of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN'97), Vienna, Austria*, pages 840–850. Lecture Notes in Computer Science, Springer Verlag, 1997.

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

[21] T. Fahringer, P. Blaha, A. Hössinger, J. Luitz, E. Mehofer, H. Moritsch, and B. Scholz. Development and Performance Analysis of Real-World Applications for Distributed and Parallel Architecture. AURORA Technical Report TR1999-16, http://www.vcpc.univie.ac.at/aurora/publications/, University of Vienna, August 1999.

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

[23] W. Fang, E. W. Felten, and M. Martonosi. Contention and queueing in an experimental multicomputer: Analytical and simulation-based results. Technical Report TR-508-96, Princeton University, Computer Science Department, Jan. 1996.

[24] J. Ferrante, V. Sarkar, and W. Trash. On estimating and enhancing cache effectiveness. In *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, Aug 1991.

[25] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

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

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

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

[29] J. Brehm et al. A Multiprocessor Communication Benchmark, Users Guide and Reference Manual. *Public Report of the ESPRIT III Benchmarking Project*, 1994.

[30] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, 1991.

[31] W. M. Jr., T. J. LeBlanc, and A. Poulos. Waiting Time Analysis and Performance Visualization in Carnival. In *ACM SIGMETRICS Symp. on Parallel and Distributed Tools*, pages 1–10, May 1996.

[32] W. K. Kaplow and B. K. Szymanski. Program optimization based on compile-time cache performance prediction. *Parallel Processing Letters*, 6(1):173–184, Mar. 1996.

[33] K. Kennedy and K. McKinley. Optimizing for Parallelism and Data Locality. In *International Conference on Supercomputing 1992*, pages 323–334, Washington D.C., July 1992.

[34] M. Kumar. Measuring parallelism in computation intensive scientific/engineering applications . *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

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

[36] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara,CA, April 1991.

[37] 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.

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

[40] A. van Gemund's *PAMELA* project webpage. http://dutepp0.et.tudelft.nl/~gemund/Pamela/pamela.html.

[41] K.-H. Park. *Dynamic Processor Partitioning for Multiprogrammed Multiprocessor Systems*. PhD thesis, Vanderbilt University, Nashville, TN, Aug 1990.

[42] K. Sevcik. Characterization of parallelism in applications and their use in scheduling . *Performance Evaluation Review*, 17(1):171–180, 1989.

[43] C. Siddhartha. *Compiling data-parallel programs for efficient execution on shared-memory multiprocessors*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, October 1991.

[44] A. van Gemund. *Performance Modeling of Parallel Systems*. Delft University Press, 1996.

[45] S. Venugopal and V. K. Naik. SHAPE: a parallelization tool for sparse matrix computations. Research report rc 17899, IBM Research Division, T.J. Watson Research Center, Yorktwon Heights, NY 10598, July 1992.

[46] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In *Proc. of 7th International Conference, HPCN Europe 1999*, pages 503–512, Amsterdam, The Netherlands, April 1999.

[47] M. Wolf and M. Lam. A data locality optimizing algorithm. In *In Proceedings of the SIGPLAN 91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[48] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6), 1999.

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