# N-MAP: A Virtual Processor Discrete Event Simulation Tool for Performance Prediction in the CAPSE Environment

A. Ferscha and J. Johnson
Institut für Angewandte Informatik und Informationssysteme, Universität Wien
Lenaugasse 2/8, A-1080 Vienna, Austria
Email: [ferscha | johnson]@ani.univie.ac.at

## Abstract

*The CAPSE (Computer Aided Parallel Software Engineering) environment aims to assist a performance oriented parallel program development approach by integrating tools for performance prediction in the design phase, analytical or simulation based performance analysis in the detailed specification and coding phase, and finally monitoring in the testing and correction phase.*

*In this work, the N-MAP tool as part of the CAPSE environment is presented. N-MAP covers the crucial aspect of performance prediction to support a performance oriented, incremental development process of parallel applications such that implementation design choices can be investigated far ahead of the full coding of the application. Methodologically, N-MAP in an automatic parse and translate step generates a simulation program from a skeletal SPMD program, with which the programmer expresses just the constituent and performance critical program parts, subject to an incremental refinement. The simulated execution of the SPMD skeleton supports a variety of performance studies. We demonstrate the use and performance of the N-MAP tool by developing a linear system solver for the CM-5.*

## 1 Introduction

The aim to accelerate the execution of computationally intensive applications using massively parallel hardware demands the availability of performance efficient implementations. As a consequence, performance orientation must be the driving force in the development process of parallel programs. In addition to the adherence to well-developed software engineering principles, only the application of a profound *performance engineering* [21] methodology can guarantee the development of successful parallel program codes.

Performance evaluation and engineering in the context of parallel processing has progressed over the past quarter of a century along different lines. Historically first, works on the so called mapping problem appeared [24, 4] which addressed assignment and scheduling issues in distributed memory MIMD systems. A large body of literature (see e.g. [20] for a classification of the particular problems studied) is available on the subject, however, most of the models have become obsolete due to the technological development of multiprocessor hardware now being in its third generation. The classical branch of computer systems performance evaluation has also recognized fields for research and application of analysis methods in parallel processing [2]. Model based performance analysis has recognized an overwhelming mass of contributions in the various paradigms: queueing network models [15, 16, 11], Petri Net modeling [1] and markovian/stochastic performance models [22] to give some pointers. Performance models have been devised that describe the interdependencies of hardware resources [17], that characterize the behavior of parallel program components and the workload for parallel systems [6], but also *integrated* performance models explicitly representing parallel program, multiprocessor hardware and mapping performance factors in a single model have been reported [8, 18]. Most of the results produced in this branch, however, are not related to the development process of parallel programs but do stand rather as contributions to the "art of modeling" for its own.

A methodological integration of performance and software engineering activities has been tried for certain purposes, and various different attempts have been made. Among the earliest approaches, we find the insertion of probes into parallel codes, generating trace information at runtime for the purpose of monitoring [19], performance tuning [3] or post-execution behavior visualization [13]. For the user-directed par-

1060-3425/95 $4.00 © 1995 IEEE

allelization of sequential codes in semiautomatic compilation systems [28, 7], parameters are computed from profiling the execution of a sequential run of the program, and the parallelization strategy is selected accordingly. Both approaches launch performance engineering activities at a time when operational (parallel or sequential) program code has already been produced, that is, at the end of the development cycle. Performance pitfalls detected in that phase, however, cannot be removed without major redesign and reimplementation efforts in general.

Performance engineering methods must therefore be applied at the very front of the development cycle in order to be able to avoid time-consuming and error-prone reimplementations. Several approaches for *performance prediction based on models* have been reported recently [18, 23, 26], some of which could be successfully applied in restricted application areas [27], but most applications fail to achieve the performance predicted by analytical models. The main reason for their limited practical relevance is the complexity of implementation details that have significant performance effects, the inaccuracy of hardware models and the unforeseen dynamics of interacting parts and layers of the application at run time. Performance prediction accurate enough to rank different implementation alternatives of an application based on program or workload *models* is just not feasible in general, so that the only practical evaluation method consists in the evaluation of the *actual* implementation.

This work systematically relates performance engineering activities to the parallel program development cycle. The activities can be classified roughly as: *performance prediction* applied in early development phases (design), *modeling* (*analytical modeling* and *simulation*) in the detailed specification and coding phase, and finally *monitoring* and *measurements* in the testing and correction phase. Particularly in this work, we present a performance/behavior prediction methodology based on the direct simulation of real (skeletal) codes, as opposed to traditional model based performance prediction (Section 2). We demonstrate how program skeletons representing just those program structures that have the highest impact on performance, can be provided very quickly by the application programmer within the the N-MAP (*N*-(virtual) processor map) tool environment (Section 3). The possibility to discover flaws that degrade performance in a parallel program as early as possible, and to reason about various implementation designs from a performance viewpoint is presented in a case study on the CM-5 (Section 4).

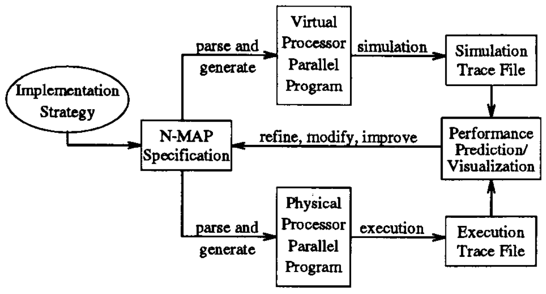


Figure 1: Parallel Program Development Cycle

## 2 Performance Oriented Parallel Program Development Cycle

The major ambition of the *performance oriented* parallel program development strategy [9] is to derive preliminary behavior and performance predictions from "rough" specifications of the parallel program. As a consequence, to express his implementation intent, the programmer should not be forced to provide detailed, operable program code, but should rather focus on specifying the constituent and performance critical program parts. A specification language, abstract enough to allow a quick denotation of the principal communication and computation pattern, but also syntactically detailed enough to support an automated code generation and performance analysis is demanded.

We consider (Figure 1) the "algorithmic idea" as it appears in the developers mind right after understanding the computational problem as the first step towards a parallel application. The N-MAP high level specification language is provided as means for the direct encoding of this algorithmic idea as a parallel SPMD program at the highest possible level of abstraction, the *task structure level.* Conveniently, the N-MAP language is based on the C programming language, providing language constructs for very intuitive notions in parallel programming: *tasks*, *processes* and (communication) *packets.* In a very abstract sense, a *task* refers to a contiguous, sequential block of program code. The functionality of a task in a parallel program is referred to as its *behavior*, and the quantification of its real or expected execution time is expressed with the *requirements* of a task. The behavior of a task can either be computation or communication, requirements are intuitively expressed in units of time. An ordered set of tasks defines a process, seen as a stream of computational or communication task calls. Since a process aggregates the behavior of the constituent tasks, the interaction among processes appears as the communication induced by tasks in different processes. The metaphor of "virtual processors" can be adequately used to refer to a process in the N-MAP language, and an arbitrary amount of virtual processors with a full interconnect can be assumed to be available for expressing algorithmic ideas. Communication objects, i.e. the actual data to be transferred among virtual processors, are referred to as *packets*. Analogous to tasks, packets are characterized by their *behavior* (functionality to gather and pack the data), and their *requirements* (quantification of the amount of data to be transferred among virtual processors, expressed in terms of the number of bytes).

Using the N-MAP language, programmers develop parallel applications by *incrementally* and *iteratively* going through various levels of explicitness. In each phase of program development, more and more detailed program functionality is added to the preliminary structure, until a complete executable parallel program is created. This process is supervised by performance predictions based on momentary information, guiding the developer through the various stages of development. With this strategy certain parts of the full specification may be intentionally absent at some stages of development, making performance predictions available right ahead of coding the details. Specifically, performance predictions can be invoked at any level of explicitness, with prediction precision correlated to the quality of available information.

## 3 The N-MAP Environment

An N-MAP specification that can be automatically processed in the N-MAP environment (Figure 2) consists of a mandatory program description at the task structure level, and other (optional) sources of text detailing functionalities and requirements:

**TSS** (Task Structure Specification) The TSS contains a skeletal specification of the program, describing it in terms of *processes*, *tasks* and *packets*. The language used in the TSS is C with a few extensions (see [10] for a more detailed description of the language extensions used).

**TRS** (Task Requirements Specification) The requirements of each task may be specified in the TRS which returns an execution time estimate for the task. If no TRS is present for a given task, unitary execution time is assumed. Probabilistic requirements (e.g. random requirements following the normal distribution) are provided and may be used for tasks with non-deterministic execution times.

**TBS** (Task Behavior Specification) The actual code to be performed in a task may be specified in the TBS which may contain any valid C function. If no TBS is given, the task is assumed to return `NULL` (in the case of a non-void function) and only the TRS is evaluated during simulation.

**PRS** (Packet Requirements Specification) Analogous to the TRS, the number of bytes to be transferred in the packet may be specified in the PRS. If no PRS is given, a default unitary packet size is assumed.

**PBS** (Packet Behavior Specification) The PBS contains the code necessary to pack the data and returns a pointer to the memory address where the data is located. If no PBS is given, only the PRS will be evaluated during simulation.

### 3.1 Discrete Event Simulation of Virtual Processor Programs

The N-MAP tool (Figure 2) operates in two phases. In the first phase, the program sources (`fn.tss`, `fn.trs`, ...) are parsed and translated into either a discrete event simulation source program (`fn.des.c`) or a target machine source program (`fn.sys.c`). In the second phase, these are in turn compiled, linked and executed producing either a trace of the program's simulated execution (`fn.sim.trf`) or of its true execution on the target machine (`fn.sys.trf`). (In Figure 2, `fn` stands for the name of the program to be simulated/executed and `sys` stands for the name of the particular target machine (e.g. `cm-5`).) All steps in the large dotted box are fully automated and usually not visible to the user.

The parser/translator itself (developed using `yacc`) takes the source files and converts them into standard C using the directives contained in the Translation Template files (`sim.t` and `sys.t`). These allow for the fully automated instrumentation of the program and the conversion of the (generic) communication calls of the TSS language into the syntax of the communication calls of the target machine. The user may also provide new translation templates, allowing N-MAP to produce source code for various target machines.

The source generated by the parser/translator is then, in the case of simulation, compiled and linked with the discrete event simulator (`des.a`) and with routines necessary for producing the trace files themselves (`trace.sim`). In the case of executable code,

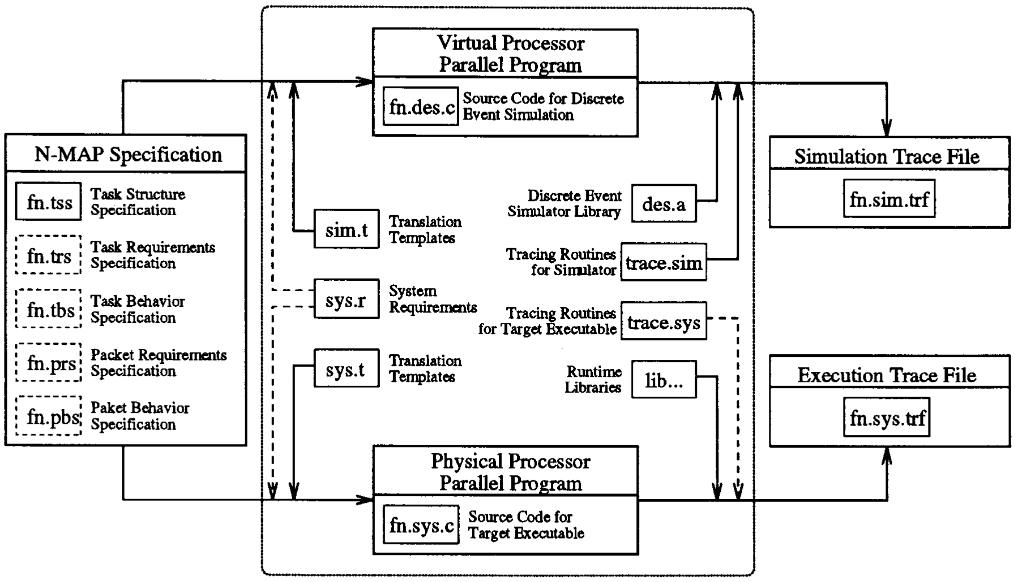

Virtual Processor Parallel Program
fn.des.c
Source Code for Discrete Event Simulation
N-MAP Specification
fn.tss
Task Structure Specification
fn.trs
Task Requirements Specification
fn.tbs
Task Behavior Specification
fn.prs
Packet Requirements Specification
fn.pbs
Paket Behavior Specification
sim.t
Translation Templates
sys.r
System Requirements
sys.t
Translation Templates
Discrete Event Simulator Library
des.a
Tracing Routines for Simulator
trace.sim
Tracing Routines for Target Executable
trace.sys
Runtime Libraries
lib...
Simulation Trace File
fn.sim.trf
Execution Trace File
fn.sys.trf
Physical Processor Parallel Program
fn.sys.c
Source Code for Target Executable

Figure 2: N-MAP Compilation and Virtual Processor Simulation Environment

the source generated is compiled and linked with the necessary libraries and tracing routines (if tracing is to be performed).

The discrete event simulator uses a threads concept to implement the behavior of the virtual processors. The threads approach was chosen over other possible implementations (e.g. simulating program behavior by the use of an interpreter) because of the speed of execution attainable with this method. Furthermore, threads can be implemented very easily and efficiently on various platforms, making them a feasible and tractable choice. The overhead introduced by the use of threads is low and due primarily to the cost of the context switch (`setjmp()` and `longjmp()`) which is necessary each time control is transferred from one virtual processor to another. Also, because each virtual processor is executing in its own context, no further adjustments must be made to the program code to be simulated in order to allow for concurrent operation.

Besides the trace files generated for the visualization and animation of the simulated virtual processor program execution, the discrete event simulator also gathers statistics about the simulation run on-the-fly in a file `fn.sim.stats` (not shown in Figure 2). The data contained here includes event occurrence frequencies allowing the computation of busy, overhead and idle time for each virtual processor, as well as the number of messages/bytes sent/received. The trace files generated by both the simulation and execution adhere to a standard trace file format (PICL [12]), defining the interface to subsequent analysis and visualization tools (e.g. ParaGraph [14]).

## 4 Application Development in N-MAP

With the following case study, the development of a parallel Housholder reduction (HR) [5] application, we show the basic usage of the N-MAP methodology and toolset. We follow the development process from the TSS until a full implementation is reached, and compare early performance predictions with empirical performance observations on the CM-5.

### 4.1 Case Study: Problem Statement

Householder reduction (HR) as a procedure to transform a set of $n$ linear equations to upper triangular form that can be solved by back substitution is known as an an unconditional numerically stable method. In contrast to e.g. Gaussian elimination, it is robust against the scaling of equations as a source of numerical errors. The HR operation is a transformation of $A\vec{x} = \vec{b}$ into $(HA)\vec{x} = H\vec{b}$, where $A$ is $n \times n$. $A$ is triangularized in $n - 1$ iterative steps over eliminations in the respective first column $\vec{a_1}$ and according transformations of the columns $\vec{a_2} \dots \vec{a_n}$ and the right

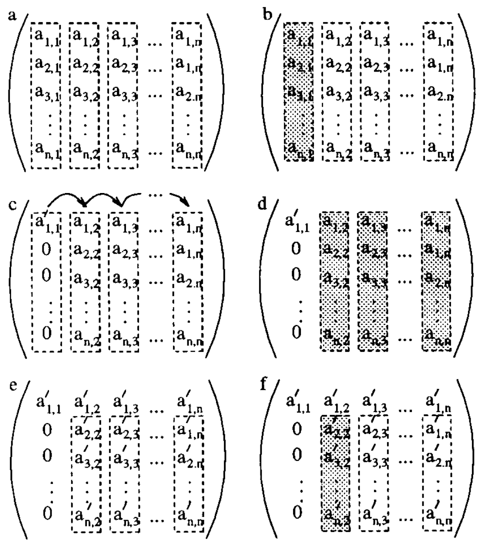


Figure 3: Implementation Strategy for HR

hand side $\vec{b}$ into $H\vec{b}$. The following two steps describe the HR operation more formally:

**eliminate**

$$\begin{aligned} a'_{1,1} &:= \text{sign}(a_{1,1})\|\vec{a_1}\| \\ q_1 &:= \sqrt{-2\,(a_{1,1} - a'_{1,1})\,a'_{1,1}} \\ H\vec{a_1} &:= [a'_{1,1}, 0, \ldots 0]^T \\ \vec{v} &:= [a_{1,1} - a'_{1,1}, a_{2,1}, \ldots, a_{n,1}]/q_1 \end{aligned}$$

**transform**

$$\begin{aligned} H\vec{a_i} &:= \vec{a_i}(1 - 2\,\vec{v}^T\,\vec{v}), \qquad \text{for } 1 < i \leq n \\ H\vec{b} &:= \vec{b}(1 - 2\,\vec{v}^T\,\vec{v}) \end{aligned}$$

The eliminate step eliminates nonzeros in the first column of $A$ and produces $\vec{v}$, whereas the transform step uses $\vec{v}$ to transform the remaining columns of $A$. Clearly, the transform step could be executed on the respective columns in parallel. The first HR operation applied to $A$ creates $HA$ with zeros below the diagonal element in the first column, and $\vec{b}$ into $H\vec{b}$; the second HR then operates on the remaining $(n-1) \times (n-1)$ submatrix of $HA$ and the $(n-1) \times 1$ subvector of $H\vec{b}$; etc.

## 4.2 Implementation Strategy

At a first glance, the transformation rules above bring up an "algorithmic idea" of a dedication of one virtual processor per column of $A$ (Figure 3 *a*)) with the following processor behavior: in the first column (processor) perform an eliminate step (Figure 3 *b*)), broadcast the resulting vector $\vec{v}$ to all the other columns (processors) (Figure 3 *c*)), and perform respective transform steps (using $\vec{v}$) in parallel (Figure 3 *d*)). After that, one HR operation is completed, and the virtual processor which has assigned the second column of $A$ can start the elimination step of the second HR operation (Figure 3 *e*)), etc. (Figure 3 *f*))

N-MAP, by supporting a processor topology independent possibility of expressing algorithmic ideas, does not require any explicit representation of the virtual processor interconnection topology. Referencing virtual processors by their index implicitly defines the communication pattern. This is a very powerful option for developing parallel algorithms by experimentation within N-MAP, since no specific hardware assumptions perturb the analysis. For our example, a ring interconnection scheme could be assumed as direct correspondence to the algorithmic idea: column (virtual processor) $i$ has to accept vectors $\vec{v}$ from column $(i-1)$, pass it on to column $(i+1)$ and perform the local *transform* with $\vec{v}$.

## 4.3 The Task Structure Specification

Thinking in terms of virtual processors that execute tasks, we can immediately provide process identifiers as `column[i]` and task identifiers as `eliminate[i]` and `transform[i][j]`. With those, the algorithmic idea developed above can now be coded directly as an N-MAP Task Structure Specification (TSS):

```
/* HR.tss: Task Structure Specification */
#define    N 16
int        i,j;
task       readblock[N],eliminate[N],
           transform[N][N];
packet     v[N];
process column[i] where { i=0..N-1; }
{
  readblock[i];
  for (j=0; j<i; j++)
   {
    recv(column[i-1],v[j]);
    if (i<N-1) send(column[i+1],v[j]);
    transform[i][j];
   }
  eliminate[i];
 if (i<N-1) send(column[i+1],v[i]);
}
```

The TSS first declares tasks for reading a column of $A$ (`readblock[N]`, `N` data items), for the *eliminate* operation denoted by `eliminate[N]`, and for the

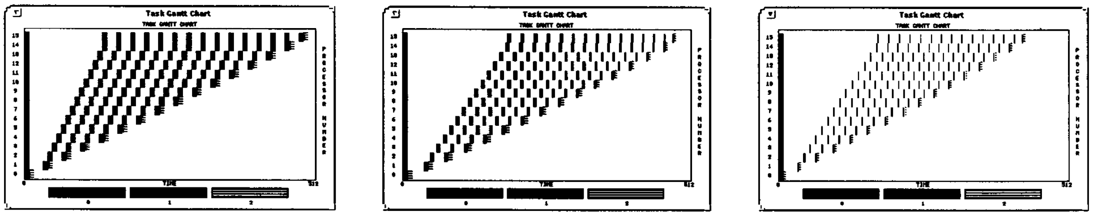


Figure 4: Task occurrences simulated from TSS (left), simulated from TSS, TRS and PRS (middle), and observed from real execution on CM-5 (right)

*transform* operation denoted by **transform[N][N]**. Note that nothing is said about the implementation (TBS) of these tasks yet. A **packet** is defined as the data item subject to transfer (**v[N]**). Finally **N** successively indexed virtual column processors are declared (**column[i] where { i=0..N-1; }**). The index **i** here serves as the virtual processor index, i.e. **transform[i][j]**; tells to conduct a transform step in the j'th iteration on processor i; **eliminate[i]** shall eliminate on processor **i**. Note, however, that the processor index can simultaneously define an actual parameter for the TRS, TBS, PRS and PBS.

Without providing – for the time being – a detailed workload characterization in terms of a TRS and PRS, or a specific target architecture hardware model (processor interconnection topology, mapping, etc.), we can immediately generate the anticipated behavior of the SPMD program in an idealized, virtual multiprocessor environment.

Figure 4 (left) shows the Gantt chart for the occurrences of the specified tasks in the simulated execution of the TSS, generated by N-MAP for $n = 16$ virtual processors. The chart mainly expresses the concurrent invocation of the individual **readblock[]** tasks (color code 0) by the 16 processors (lines in the diagram) at the very left side of the chart, and the dependencies among **transform[][]**-tasks (color code 1) and **eliminate[]**-tasks (color code 2), i.e. the availability of vectors generated by **eliminate[]** to be "piped" to other **column[]**s. White spaces in the lines of the diagram indicate that the corresponding processor is idle at the respective points in time.

The creation of this chart only required a few lines of code in the TSS and a few seconds of CPU time for the simulation on a SunSparc workstation – but gives a very clear intuition on the performance quality of the algorithm in the implementor's mind: It is easily seen that expected processor utilization is rather poor.

## 4.4 Adding Requirements

Naturally the vectors $\vec{v}$ arriving in some virtual processor become shorter and shorter with each iteration, and the amount of data to be transferred, as also the computational complexity of vector transformations, reduces linearly with the increase of the iteration count. A column $i$ stops transforming after $(i-1)$ vectors $\vec{v}$, performs a local eliminate step generating a new $\vec{v}$, propagates this $\vec{v}$ to column $(i+1)$ and finally terminates.

This additional quantitative information can be added to the TSS in terms of the TRS and PRS as a matter of refinement of the characterization of the quantitative behavior of the program. Assuming the requirements of **eliminate[]** and **transform[][]** to be linearly dependent to the iteration level j, since the vectors to be manipulated reduce in size with each iteration, we could write the respective TRSs as:

```
long REQ_transform(i,j) int i,j;
{ return alpha_transform+beta_transform*j; }

long REQ_eliminate(i) int i;
{ return alpha_eliminate+beta_eliminate*i; }
```

The communication related requirements in terms of the packet sizes defining the amount of data to be transferred in the respective iterations of the algorithm are written in the PRS as:

```
long REQ_v(i) int i;
{ return (N-1-i) * sizeof(float); }
```

Using those TRSs and the PRS, the N-MAP tool would generate simulated task occurrences reflecting the reduced computational complexity towards the higher iteration levels, as compared to the simulation with unitary requirements. Target hardware performance characteristics, however, are (intentionally) not included in our preliminary N-MAP specification. To turn from the idealized target system to a real environment, we have to include a characterization of the performance influences induced by the specific target system. As an example, we shall demonstrate here, how the effects of the CMMD message passing library [25] provided by the operating systems of the CM-5 can be integrated and used by the N-MAP simulation engine.

The "communication model" implicitly assumed so far is a constant one, i.e. the behavior of a blocking send communication together with a constant message propagation time is simulated by N-MAP. To provide a more detailed communication model for the **CMMD_send_block** and **CMMD_receive_block** synchronous message passing calls to be used in the final implementation, we subdivide the execution time of each communication operation into three distinct phases: The *init* phase represents the time needed to create a communication object, that is, to allocate buffer space, move data to the buffer and deliver a pointer to the message buffer. In the *start* phase, the pointer is subsequently passed to the communication subsystem which then initiates the send/receive. The *wait* phase is the time necessary for the communication subsystem to complete the call, that is, to confirm that the message has been forwarded and that the sender/receiver buffers may be safely accessed. In the case of locally blocking communication, *wait* follows *start* immediately, whereas in the case of locally non-blocking communication, computation may intercede in order to mask latency.

As an input to the N-MAP simulation tool, time requirements for the respective phases are defined in the *System Requirements File* (SRF) (<sys>.r). The time a sender virtual processor is involved in communication is simply $s_{init}()+s_{start}()+s_{wait}()$. Additional delay is induced for the receiving processor due to idling until the sender completes the *start* phase, and due to the inherent latency caused by the communication network (*latency*). The excerpt from the SRF cm-5.r below reflects the requirements related to the CMMD specific communication model:

```
long REQ_ssend(packLEN) long packLEN;
{ return REQ_ssend_overhead +
        packLEN * REQ_ssend_byte; }

long REQ_recv(packLEN) long packLEN;
{ return REQ_recv_overhead +
        packLEN * REQ_recv_byte; }

long REQ_propagation_latency(packLEN) long packLEN;
{ return REQ_data_transfer_setup +
        REQ_propagation_delay*packLEN; }
```

With the additional CM-5 specific SRF cm-5.r, a "CM-5 specific" sample trace is generated by N-MAP, giving the predicted task occurrences as in Figure 4 (middle). The chart explains, that for the CM-5 the communication will dominate computations (gaps between **transform** blocks in the same iteration on different **columns**). Compared to previous predictions based on less specific information, a more detailed behavior could be generated.

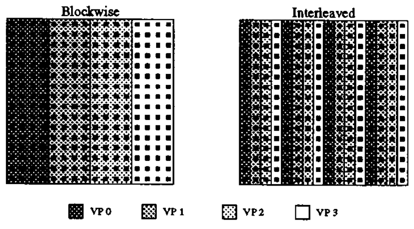


Figure 5: Blockwise vs. interleaved decomposition

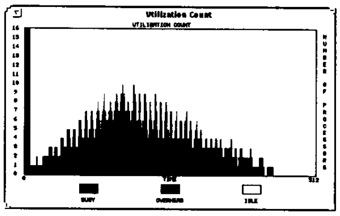
## 4.5 Adding Behaviors

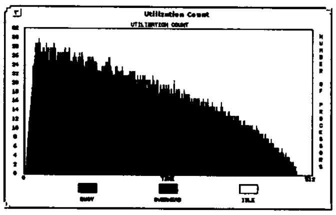
Given that the programmer is now willing to fully implement parallel Householder transformations based on the preliminary algorithmic idea, he would now provide TBSs and PBSs containing the source code for the tasks declared in the TSS. The N-MAP tool can then automatically generate an executable for the CM-5 (from the TSS, TBS, TRS, PBS and PRS), using CMMD message passing library call substitutions for the **send()** and **recv()** tasks in the TSS.

The instrumented version running on 16 nodes of a CM-5 produced the behavior in Figure 4 (right) and Figure 6 (left). From a direct comparison of the behaviors generated on the basis of the TSS (exclusively) and incremental extensions up to the final SPMD program we see, that it was sufficient to characterize the program behavior in terms of TSS to obtain the main performance characteristics of the algorithm. We could derive this program behavior within a few minutes of programming efforts, while the full implementation of the algorithm would have taken hours of coding, debugging, etc. Thus – without wasting programming efforts – we can make performance critical implementation decisions in the early program development phases.

## 4.6 Implementation Alternatives

From the very first chart in Figure 4 the programmer might have recognized the inefficiency of the algorithmic idea based on a "one column of $A$ to one processor mapping". Already at that point, various other algorithmic ideas might have been studied by providing further TSSs.





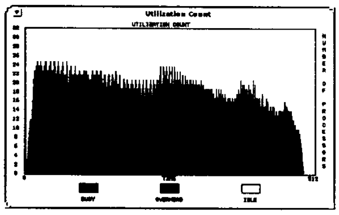


Figure 6: Processor utilization observed from execution on CM-5: original implementation strategy (left), blockwise (middle), interleaved (right)

```
/* hh.contblo.tss: contiguous blockwise */
#define P 32      /* No of processors   */
#define N 128     /* Dimension of A     */
#define B 4       /* columns/processor  */
int i,j,k;

task readblock[N],transform[N][N],eliminate[N];
packet v[N];

process column[i] where { i=0 .. P-1; }
{
for (j=0; j<B; j++)
   {
   readblock[i*B+j];
   }

/* pivot column not in my block */
for (j=0; j<i*B; j++)
   {
   recv(column[i-1],v[j]);
   if (i<P-1)
       {
       send(column[i+1],v[j]);
       }
   for (k=0; k<B; k++)
      {
      transform[i*B+k][j];
      }
   }

/* pivot column in my block */
for (j=0; j<B; j++)
   {
   eliminate[i*B+j];
   if (i<P-1)
      {
      send(column[i+1],v[i*B+j]);
      }
   for (k=j+1; k<B; k++)
      {
      transform[i*B+k][i*B+j];
      }
   }
}
```

```
/* hh.interl.tss: interleaved           */
#define P 32       /* No of processors   */
#define N 128      /* Dimension of A     */
#define B 4        /* columns/processor  */
int i,j,k,s;

task readblock[N],transform[N][N],eliminate[N];
packet v[N];

process column[i] where { i=0 .. P-1; }
{
for (j=0; j<B; j++)
   {
   readblock[i+j*P];
   }

s=0;
for (j=0; j<N; j++)
   {

   if (j%P==i)  /* eliminate in my block */
      {
      eliminate[j];
      send(column[(i+1+P)%P],v[j]);
      s++;
      }
   else  /* pivot not in my block */
      {
      recv(column[(i-1+P)%P],v[j]);
      if ((i+1)%P!=j%P)
         {
         send(column[(i+1+P)%P],v[j]);
         }
      }

   for (k=s; k<B; k++)
      {
      transform[i+k*P][j];
      }
   }
}
```

Figure 7: Variations of the Algorithmic Idea for the CM-5: TSSs

A contiguous blockwise partitioning of $A$ could significantly reduce the communication invocations by a single processor and would allow the mapping of a set of consecutive (in terms of their index) virtual processors to a set of physical processors. An improvement of this partitioning strategy is to interleave the columns (virtual processors) when mapping them to physical processors in such a way, that the amount of blocking time in the propagation of orthogonal vectors is minimized. Every physical processor $j$ out of a set of $M$ processors in this case is assigned columns $j + iM$, $(i = 0, 2, \ldots, \frac{N}{M} - 1)$ of $A$.

These two implementation strategies can be quickly specified in the form of a TSS (see Figure 7). By re-using the behavior and requirements specifications already developed in the previous implementation, we immediately obtain performance characteristics for the two new strategies which are shown in the form of a Utilization Count Chart in Figure 6. We see that both interleaved and blockwise mapping are superior in performance to the original one-to-one mapping (while processing more matrix columns). True execution performance on the CM-5 is compared with predictions of N-MAP for the contiguous blockwise (Figure 8) and the interleaved decomposition (Figure 9) of $A$. Naturally, the regular structure (deterministic behavior) of Householder reductions, together with our communication model are responsible for the high prediction precision achieved by N-MAP. We wish to note however, that prediction quality is not the main claim of N-MAP. The major goal addressed with the N-MAP tool is rather to demonstrate that from within most common programming environments (UNIX) it is possible to generate very quickly and with minor programming efforts execution behaviors of parallel programs as they would execute in a real multiprocessor environment. Earliest possible behavior visualizations and performance predictions (even if rather rough) are the primary concern of N-MAP.

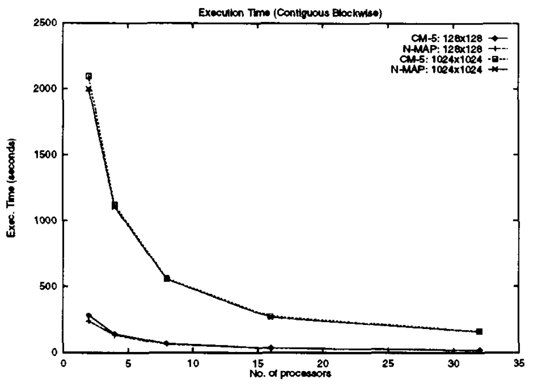


Figure 8: Blockwise decomposition of $A$: CM-5 execution vs. N-MAP prediction

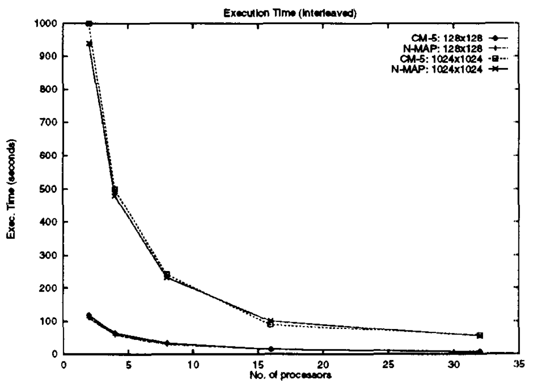


Figure 9: Interleaved decomposition of $A$: CM-5 execution vs. N-MAP prediction

## 5 Conclusions

This work proposed an incremental, performance oriented development process for parallel programs. Performance/behavior prediction has been treated as a performance engineering activity in the early development stages, supporting the evaluation of implementation strategies far ahead of the actual coding. Specifically, an extension to the C programming language has been made to allow a quick specification of parallel SPMD programs at the highest level of abstraction, the task structure level. Mainly communication patterns among tasks comprised in processes (metaphorically seen as *virtual processors*) are expressed at this level, and performance impacts of the preliminary implementation skeleton can already be predicted. As program functionality and a quantification of the anticipated program execution behavior is incrementally added in refinement steps, the quality of the predictions is successively improved. In this way, starting from a virtual processor, target architecture independent program specification, a performance efficient implementation is reached for a specific target platform after a few refinement iterations.

Methodologically, task structure specifications (TSSs), parametrized with quantitative workload parameters and target specific performance characteristics as an option, are parsed by the N-MAP tool and translated into a virtual processor SPMD programs, the discrete event simulation of which generates execution statistics and traces. All relevant program performance characteristics can be deduced from the statistics/traces in a subsequent analysis. In the final development phase, N-MAP can automatically generate "physical processor" SPMD program source code for the dedicated target software environment (message passing libraries).

With a case study, the implementation of parallel Householder transformations on the CM-5, we have demonstrated how a full implementation is incrementally developed from the TSS. Variations of implementation strategies could be analyzed based on fast SPMD skeletons, supporting performance decisions without time-consuming and error-prone coding and debugging of the full program version.

## References

[1] M. Ajmone Marsan, G. Conte, and G. Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93 - 122, May 1984.

[2] I. F. Akyildiz. Performance Analysis of a Multiprocessor System Model with Process Communication. *The Computer Journal*, 35(1):52–61, 1992.

[3] Thomas E. Anderson and Edward D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. *Performance Evaluation Review, Special Issue, 1990 ACM SIGMETRICS*, 18(1):115–125, May 1990.

[4] Shahid H. Bokhari. On the Mapping Problem. *ACM Transactions on Computer Systems*, C-30(3):207–214, March 1981.

[5] P. Brinch Hansen. Householder Reduction of Linear Equations. *ACM Computing Surveys*, 24(2):185 - 194, June 1992.

[6] Maria Calzarossa and Guiseppe Serazzi. Workload Characterization: A Survey. In *Proceedings of the IEEE*, 1993.

[7] T. Fahringer and H.P. Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. In *Proc. 1993 ACM Int. Conf. on Supercomputing, July 1993, Tokyo, Japan*, 1993.

[8] A. Ferscha. Modelling Mappings of Parallel Computations onto Parallel Architectures with the PRM-Net Model. In C. Girault and M. Cosnard, editors, *Proc. of the IFIP WG 10.3 Working Conf. on Decentralized Systems*, pages 349 - 362. North Holland, 1990.

[9] A. Ferscha. A Petri Net Approach for Performance Oriented Parallel Program Design. *Journal of Parallel and Distributed Computing*, 15(3):188 - 206, July 1992.

[10] A. Ferscha and J. Johnson. Performance Oriented Development of SPMD Programs Based on Task Structure Specifications. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR94-VAPP VI*, LNCS 854, pages 51-65. Springer Verlag, 1994.

[11] Cynthia A. Funka-Lea, Tasos D. Kontogiorgos, Robert J. T. Morris, and Larry D. Rubin. Interactive Visual Modeling for Performance. *IEEE Software*, pages 58–68, September 1991.

[12] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A Users' Guide to PICL: a Portable Instrumented Communication Library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, August 1990.

[13] G. Haring and G. Kotsis, editors. *Performance Measurement and Visualization of Parallel Systems*, volume 7 of *Advances in Parallel Computing, G. R. Joubert, Udo Schendel (Series Eds)*. North Holland, 1993.

[14] Michael T. Heath and Jennifer A. Etheridge. Visualizing Performance of Parallel Programs. Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, May 1991.

[15] P. Heidelberger and K. S. Trivedi. Queueing Network Models for Parallel Processing with Asynchronous Tasks. *IEEE Transactions on Computers*, C-31(11):1099–1109, November 1982.

[16] P. Heidelberger and K. S. Trivedi. Analytic Queueing Models for Programs with Internal Concurrency. *IEEE Transactions on Computers*, C-32(1):73–82, January 1983.

[17] I. O. Mahgoub and A. K. Elmagarmid. Performance Analysis of a Generalized Class of $m$-Level Hierarchical Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):129 - 138, March 1992.

[18] V. W. Mak and S. F. Lundstrom. Predicting Performance of Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–269, July 1990.

[19] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433 - 450, July 1992.

[20] M. G. Norman and P. Thanisch. Models of Machine and Computation for Mapping in Multicomputers. *ACM Computing Surveys*, 25(3):261 - 302, September 1993.

[21] Connie U. Smith. *Performance Engineering of Software Systems*. Addison Wesley, 1990.

[22] R.M. Smith and K.S. Trivedi. The Analysis of Computer Systems Using Markov Reward Processes. In H. Takagi, editor, *Stochastic Analysis of Computer and Communication Systems*, pages 589 - 630. North-Holland, 1990.

[23] H. V. Sreekantaswamy, S. Chanson, and A. Wagner. Performance Prediction Modeling of Multicomputers. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 278–285, Los Alamitos, California, 1992. IEEE Computer Society Press.

[24] Harold S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[25] Thinking Machines Corporation, Cambridge, Massachusetts. *CMMD Reference Manual, Version3.0*, May 1993.

[26] A. J. C. van Gemund. Performance Prediction of Parallel Processing Systems: The PAMELA Methodology. In *Proc. 1993 ACM Int. Conf. on Supercomputing, July 1993, Tokyo, Japan*. ACM, 1993.

[27] H. Wabnig and G. Haring. PAPS - The Parallel Program Performance Prediction Toolset. In *Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation.*, LNCS 794, pages 284–304. Springer-Verlag, 1994.

[28] H.P. Zima, H.-J. Bast, and H. M. Gerndt. SUPERB - a Tool for Semiautomatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1–18, 1988.