High Performance Computing in the Optimization of Software Test Plans

Karl Doerner[1] **Erwin Laure**[2]

[1]Department of Management Science
University of Vienna
Brünner Strasse 72, A-1210, Vienna, Austria
`Karl.Doerner@univie.ac.at`

[2]Institute for Software Science
University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
`erwin@par.univie.ac.at`

Abstract

Statistical software testing is an increasingly popular method in the software development cycle. An exact modeling of the usage profiles of a software system is an indispensable prerequisite for statistical testing. Recently, new techniques for obtaining optimal usage profiles even the presence of rarely used critical functions have been introduced. Although these techniques deliver unbiased dependability estimates with a single model (instead of using multiple models as it is the current practice) their applicability is hampered by their prohibitive computational complexity.

In this paper we demonstrate that these techniques can be effectively employed even for large software systems by exploiting high performance computing. We discuss the parallelism potential within these techniques and present a parallel implementation based on a high level parallel language—*High Performance Fortran*. Our techniques are general enough to allow an efficient optimization of medium to large scale test plans on high-end parallel machines as well as on more cost effective PC-clusters. A real world software system is studied in detail to prove our claims.

1 Introduction

In the past years statistical software testing has become more and more popular (see e. g. [12, 13, 15]) because it allows to incorporate empirical data in software test plans. Thereby uncertainty about the behavior of a software system is removed and an aid to decision-making regarding further testing is provided. A basic prerequisite for statistical software testing is the availability of suitable usage profiles of a software system. The Markov-chain modeling technique, developed by [19, 20, 18] is a reasonable technique to model such profiles because of its flexibility, easy comprehensiveness, and potential for deep formal analysis.

The operational profile described by a Markov usage model provides the input for random testing which eventually results in the computation of dependability estimates such as risk, reliability, safety, etc. However, these dependability measures may be biased, especially in the presence of highly mission-critical, yet rarely used functions.

Gutjahr [3, 4, 5] discussed an approach to overcome this problem by replacing the original empirical usage profile with a special test profile. In this approach, transition probabilities in test plans are optimized with respect to a specific dependability measure taking into account critical or error-prone operations. As a result of this optimization unbiased dependability measures can be obtained. However, the computational complexity of the optimization prevents its application to larger software systems. For instance, the optimization of a medium-sized system with 167 functions requires, depending on the computing system employed, up to 10 hours. Moreover, the optimization may have to be applied several times due to changes in the software system both during the engineering and the deployment phase. Because of the high costs involved this technique is not applied in production systems for the time being.

In this paper we show that by leveraging high performance computing techniques the runtime of the optimization phase can be reduced considerably; thus making the optimization of test plans a viable option for the software development process. The techniques we describe are general enough to deal with medium to large scale software test plans. Moreover, our techniques are designed to be platform independent such that they provide good efficiency on high-end parallel machines as well as on more cost effective PC-clusters. In describing the parallelization of the optimization we put emphasis on discussing in detail the rationale behind our choices, potential pitfalls, as well as the tools employed. This discussion is aimed to provide the reader with enough information to employ our techniques efficiently to related problems.

The remainder of this paper is organized as follows. In Section 2 we give a brief introduction on usage-based statistical software testing focusing on the optimization algorithm introduced by Gutjahr (cf. [3, 4, 5]). A prototype sequential implementation is presented in Section 3. Section 4 deals with the parallelization of this prototype implementation. We discuss the parallelism potential of the sequential algorithm, present a high level parallel language (HPF [6]), and give details on the parallel implementation. To prove the efficiency of our techniques we present experimental results in Section 5. We discuss the runtime behavior of a small sized synthetic problem as well as of a medium sized real world problem on both, a high-end parallel machine and a Beowulf PC-cluster. A numerical analysis of the optimization is also presented in this section. We conclude this paper with a summarizing discussion in Section 6.

2 A Markov Usage Model with Optimization of Test Transition Probabilities

In this section we develop the formal prerequisites for usage-based statistical software testing. Specifically, we introduce the formal graph-based definition of a Markov Usage Model (MUM) in Section 2.1 and outline an applied approach of MUM based statistical testing in Section 2.2. In Section 2.3 we summarize a novel approach that optimizes test transition probabilities.

2.1 Graph Representation of Markov Usage Models

A markov usage model is based on an attributed directed graph $\mathcal{G} = (V, A)$ where

- V is a set of n nodes $v_1, \ldots, v_n$ representing usage states (e. g. program invocation, program termination, informations screens, etc.) of a software system. v_1 is the initial state and v_n the terminal state;
- A is a set of arcs representing the state transitions which correspond to the operations of the application. Each arc (i, j) is attributed with a transition probability $p(i, j)$. This value indicates the probability of a transition from the current state v_i to state v_j. The transition probabilities satisfy the conditions $0 \leq p(i, j) \leq 1$ and the sum of the probabilities of all outgoing arcs of a node is 1.

A path X through the graph is denoted by the index list of the nodes traversed: $X = 1, \ldots, n$. The terminal state n is assumed to be an absorbing state, i.e. it cannot be left anymore: $p(n, n) = 1$ and $p(n, j) = 0$ for $j \neq n$. Furthermore, we assume that each node $v_i \in V$ is reachable from node v_1, i. e., there is a path in $\mathcal{G}$ from node v_1 to any node v_i.

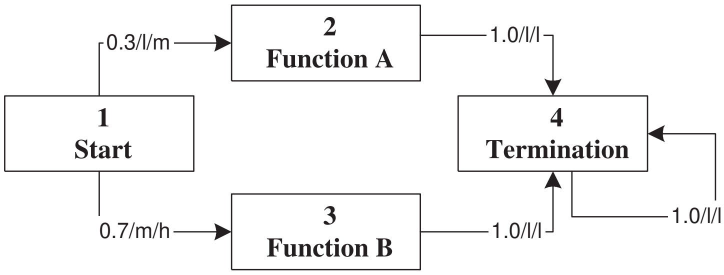


Figure 1: A Simple Markov Usage Model

2.2 Applied Usage-Based Statistical Testing

MUMs are considered in literature to be a viable tool for statistical software testing [12] and practically applied in *Cleanroom Software Engineering* [13]. Cleanroom Software Engineering is a theory based, team-oriented process for the economic production of high quality software.

In the Cleanroom process, a MUM is developed from a software system and random test cases are generated automatically from this usage model. Each test case is a "random walk" through the MUM, from the initial state to the terminal state. The test cases generated are used as instructions for human testers or as input sequences for automated testing.

Figure 1 shows a simple MUM that is comprised of four nodes. Based on such a MUM a number of test cases (for instance path $X = 1, 2, 4$) are generated automatically. The number of test cases generated depends on one hand on the available test budget and on the other hand on the required dependability. Dependability measures reflect the goodness of the software with respect to risk, reliability, safety, etc. (see for instance [9]).

A hierarchy of usage models with different transition probabilities is generated in order to model different usage profiles. Information needed to build these models can be derived from historical data, user experience, operational scenario documents and other sources.

The main drawback of Cleanroom testing is that the dependability of systems with highly critical, yet rarely used functions is overestimated. Therefore, separate MUMs have to be built which consider these functions appropriately. However, it is not an easy task to find appropriate transition probabilities for these rare events.

2.3 Optimization of Test Transition Probabilities

To overcome the drawbacks of classical Cleanroom testing outlined above, Gutjahr devised an approach that delivers unbiased dependability estimates without separate models for critical functions. In the remainder of this Section we summarize Gutjahr's approach. For a more detailed discussion the interested reader is referred to [3, 4, 5].

In contrast to the approach outlined above the generation of test cases in Gutjahr's approach is based on a *test profile* rather than on a real *usage profile*. The transition probabilities of the test profile take critical or error prone functions into account, appropriately.

In order to compute the transition probabilities for the test profile the original usage profile is augmented with additional information on failure rates and losses in case of failures. Let $f(i, j)$ denote the estimated failure probability of arc $(i, j) \in A$ and $l(i, j)$ the estimated loss due to failure, respectively. An arc $(i, j) \in A$ is attributed with $p(i, j)$ / $f(i, j)$ / $l(i, j)$ as depicted in Figure 1. We use the constants l (*low*), m (*medium*), and h (*high*) for denoting $f(i, j)$ and $l(i, j)$. The effective values of f are in the range of $0 \leq f \leq 1$; l denotes the loss a company experiences in terms of absolute currency unites. Note, that failures are non-blocking, i.e. the system may continue even in the presence of failures.

From the augmented usage profile the test profile with transition probabilities that allow an unbiased dependability estimate is derived. Specifically, the transition probabilities of the original

```
procedure OptimizeTest {
  set 𝒯 := 𝒫;
  draw a sample of paths according to 𝒫;
  for s := 1 to max_s {
    for i := 1 to n {
      for each j with t(i,j) > 0 {
        compute from the sample an estimate D(i,j) of the
        partial derivative of G(T) to the variable t(i,j);
      }
      determine that j = j_min for which D(i,j) is minimal;
      for each j with t(i,j) > 0 {
        t(i,j) := (1 − steps/s)t(i,j);
        if (j = j_min)t(i,j) := t(i,j) + steps/s;
      }
    }
  }
}
```

Figure 2: Optimization Algorithm

usage profile are modified to reflect the impact of critical functions.

Let $\mathcal{P}$ denote an augmented usage profile and $\mathcal{T}$ the test profile derived from $\mathcal{P}$. $X = 1, \ldots, n$ is a randomly sampled path according to $\mathcal{P}$. The probability that path X is traversed in a given program execution according to $\mathcal{P}$ is given by

$$P(X) \;=\; \prod_{k \in X \wedge k > 1} p(k-1, k). \tag{1}$$

Similarly, the probability of path X according to $\mathcal{T}$ is denoted by $T(X)$.

$H(X)$ denotes a specific dependability measure such as risk, safety, or reliability with respect to path X. For instance, risk under the static model [5] can be approximated by

$$H(X) \approx \sum_{k \in X \wedge k > 1} f(k-1, k)\,[l(k-1, k)]^2, \tag{2}$$

Since $H(X)$ is computed based on a test profile $\mathcal{T}$ instead of the original usage profile $\mathcal{P}$ this change has to be compensated by assigning the weights $P(X)/T(X)$, called *likelihood ratio*, to the test results.

With these prerequisites the goal of finding a test profile $\mathcal{T}$ with minimum variance, i.e. with the highest precision of the estimate, can be achieved through a stochastic process.

A test profile $\mathcal{T}$ is a minimum variance profile if and only if the expected dependability value according to $\mathcal{P}$, weighted by the likelihood ratio, is minimized [5]. The expected weighted dependability value can be approximated by

$$G(T) \approx -\frac{1}{np} \sum_{i=1}^{np} \frac{P(X_i)}{T(X_i)} \cdot H(X_i). \tag{3}$$

where $X_i, 1 \leq i \leq np$ is a randomly sampled path according to $\mathcal{P}$. The precision of this estimate correlates with the number of sampled paths (np).

Minimizing (3) is a stochastic optimization problem which can be solved, for instance, by applying a stochastic optimization algorithm of Frank-Wolfe type [11]. Figure 2 outlines such an algorithm. The idea of the algorithm is the following: First, we set $\mathcal{T}$ equal to $\mathcal{P}$. This means that the transition probabilities in $\mathcal{T}$ denoted by $t(i,j)$ are initialized with the corresponding values in $\mathcal{P}$. A sample of np independent paths according to $\mathcal{P}$ is drawn. An approximation of the weighted expected dependability value (cf. Equation 3) obtained by the sample is minimized. This is done iteratively: in max_s iterations the transition probabilities in $\mathcal{T}$ are modified by

1. determining an arc (i, j_{min}) for any node v_i such that an augmentation of $p(i, j_{min})$ leads to the maximal improvement of the estimated $G(T)$, and
2. augmenting the probability for arc (i, j_{min}) and reducing the probability of all other outgoing arcs of v_i. This modification is represented by a factor *steps* which depends on the number of iterations.

The improvement of $G(T)$ with respect to an arc (i, j) can be computed by estimating the partial derivative

$$\frac{\partial G(T)}{\partial t(i,j)} = D(i,j) \approx -\frac{1}{np} \sum_{k=1}^{np} P(X_k) H(X_k) \cdot \frac{M(i,j,X_k)}{t(i,j)} \prod_{l,m \in A} [t(l,m)]^{-M(l,m,X_k)} \tag{4}$$

where $M(i, j, X_k)$ denotes the number of times arc (i, j) is traversed in path X_k. Arc (i, j) for which $D(i, j)$ yields the minimum value is finally chosen to be augmented.

The solution obtained by procedure `OptimizeTest` is only an approximation, even after a large number of iterations (`max_s`). This drawback may be overcome by repeating, in an outer loop, the process of sampling paths and modifying the current test profile $\mathcal{T}$.

3 An Implementation of the Optimization

In this section we discuss a prototype implementation of the optimization presented in Section 2.3. The implementation was done in Fortran 90 [7] and we assume some familiarity with Fortran 90 in the remainder of this paper.

3.1 Data Structures

As discussed in Section 2.3 the input data for the program is an augmented usage profile $\mathcal{P}$ which is transformed to an optimized test profile $\mathcal{T}$. For representing $\mathcal{P}$ and $\mathcal{T}$ we elected an adjacency matrix representation. Specifically, $\mathcal{P}$ is represented by three adjacency matrices `P`, `F`, and `L` storing the transition probabilities (`P`), failure probabilities (`F`), and estimated losses (`L`). $\mathcal{T}$ is represented by a single adjacency matrix `T` which stores the optimized transition probabilities.

An adjacency matrix representation of graphs typically results in very sparse matrices. For an efficient representation of sparse matrices several storage formats have been proposed in literature (see e.g. [17]). In our implementation we employ the *Compressed Row Storage* (CRS) format. In this format a vector P' stores the non-zero values of matrix P in row-major order. Another vector C stores the column index of each non-zero element, while a third vector B marks the beginning of each row in P' and C. B and C are called auxiliary arrays since they do not contain actual data; they only provide information about the structure of the matrix. Matrix elements are identified using a (i, j) index set, where i denotes the i-th row of the matrix and j denotes the j-th non-zero element in that row. The matrix element referred to by (i, j) is the one at row number i, column number $C(B(i)+j-1)$ and has the non-zero value stored in $P'(B(i)+j-1)$. Figure 3 illustrates the CRS representation of the transition probability matrix P derived from the Train Info System depicted in Figure 5 (cf. Section 5).

Apart from the four matrices `P`, `F`, `L`, and `T`, each of which is stored in CRS format, `NP` sampled paths are stored in a two dimensional matrix `M`. Element `M(i,j)` denotes how often state transition `j` (remember that we use a CRS format) took place in sample `i`. Summarizing, the main data structures employed are declared as follows:

P

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1.0	0	0	0	0	0	0	0	0	0	0
2	0	0	.99	0	0	0	0	0	0	0	0	.01
3	0	.01	0	.94	0	0	0	.02	0	0	0	.03
4	0	0	.10	0	.30	.20	.20	0	0	0	0	.20
5	0	0	0	1.0	0	0	0	0	0	0	0	0
6	0	0	0	1.0	0	0	0	0	0	0	0	0
7	0	0	0	1.0	0	0	0	0	0	0	0	0
8	0	0	.30	0	0	0	0	0	.30	.10	.30	0
9	0	0	0	0	0	0	0	1.0	0	0	0	0
10	0	0	0	0	0	0	0	1.0	0	0	0	0
11	0	0	0	0	0	0	0	1.0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	1.0

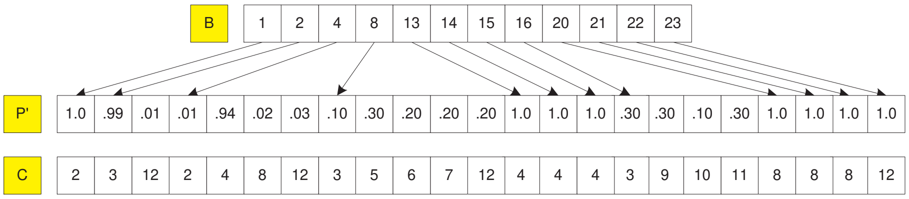


Figure 3: Sparse Matrix Representation

```
INTEGER                          :: N      ! number of nodes in the graph
INTEGER                          :: A      ! number of edges in the graph
INTEGER                          :: NP     ! number of paths sampled

DOUBLE PRECISION, ALLOCATABLE :: P(:)    ! original transition probabilities
DOUBLE PRECISION, ALLOCATABLE :: F(:)    ! failure probabilities
DOUBLE PRECISION, ALLOCATABLE :: L(:)    ! estimated losses
DOUBLE PRECISION, ALLOCATABLE :: T(:)    ! optimized transition probabilities
INTEGER, ALLOCATABLE             :: M(:,:) ! paths sampled

INTEGER, ALLOCATABLE             :: C(:)    ! column index
INTEGER, ALLOCATABLE             :: B(:)    ! row start
```

3.2 Control Structures

Figure 4 shows a high level view of the implementation of the optimization algorithm given in Figure 2. Within the outermost approximation loop `K` three main parts can be identified:

Part 1 First `NP` paths are sampled and stored in matrix `M`. At the same time the probability `PROB`, the dependability measure `H`, and the first term of Equation 4

$$...P(X_k)H(X_k)... \tag{5}$$

denoted by `PH` is computed for each path.

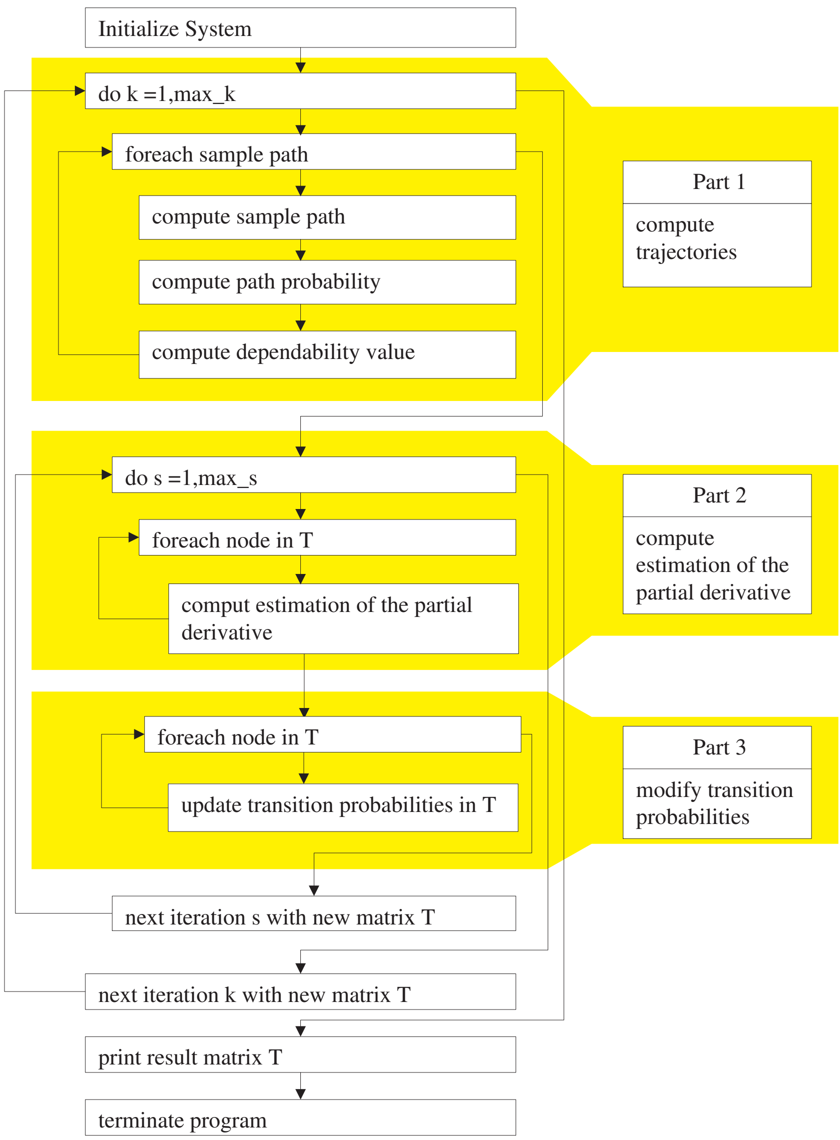


Figure 4: Flow Chart

```
! initialize M
   M = 0.D0
   DO I = 1,NP     ! loop over paths
      PROB = 1.D0 ! probability of the path

      DO WHILE (sample_new_element()) ! sampling loop
         NEW_POSITION = ! position of a newly generated path element
         PROB = PROB * P(NEW_POSITION)
         M(I,NEW_POSITION) = M(I,NEW_POSITION)+1
      END DO
      H = ! dependability measure
      PH(I) = PROB * H
   END DO
```

After all the necessary paths have been sampled the second approximation loop `S` is entered which comprises part 2 and part 3.

Part 2 In this part the estimate for the partial derivative (cf. Equation 4) is computed. This can be done in three steps: first, the dot product

$$\dots \prod_{l,m} [t(l,m)]^{-M(l,m,X_k)} \dots \tag{6}$$

which is the third part of Equation (4) is computed for every path in `M` and stored in auxiliary array `DP`. Thereafter, the dot product is multiplied with `PH` already computed in part 1. This leads to a partial result which is the same for every path. We keep track of these partial results within the auxiliary array `HELP`. Finally, the computation of the partial derivative for every node of `T` is completed by weighing the partial result of a path with the arc traversal frequency stored in `M` and the transition probability `T`.

Note that due to the Fortran storage format (which is column-major order) it is advantageous to iterate over the rows of matrix `M` in a blocked manner in order to reduce cache misses.

```
! computation of dot product
    DP = 1.D0
    DO I = 1,A       ! loop over edges
       DO J = 1,NP   ! loop over paths
          DP(J) = DP(J) * EXP(((-1)*M(J,I)) * LOG(T(I)))
       END DO
    END DO

! computation of partial result for every path
    DO I = 1,NP      ! loop over paths
       HELP(I) = DP(I)*PH(I)
    END DO

! complete computation of the estimation of the
! partial derivative
    D = 0.D0
    DO I = 1,A       ! loop over edges
       DO J = 1,NP   ! loop over paths
          D(I) = D(I) - HELP(J)*M(J,I)/T(I)
       END DO
    END DO
```

Part 3 Subsequently, the transition probabilities in `T` need to be modified. This requires that for each node the outgoing arc with the minimum value of `D` (denoted by `J_MIN`) is determined. Finally, the test transition probability for the minimum arc is increased by a value `STEPS` while all the other transition probabilities are reduced by `1 - STEPS`. `STEPS` depends on the values of both approximation indices `S` and `K`. It is guaranteed that after applying this modification the sum of the test transition probabilities of all outgoing arcs of a node equals one.

```
DO I = 1,N    ! loop over nodes

   J_MIN =    ! outgoing arc with minimum D for any node
   STEPS =    ! compute modification value



! modify transition probabilities
   DO J = B(I),B(I+1)-1
      T(J) = (1.D0 - STEPS) * T(J)
   END DO
   T(J_MIN) = T(J_MIN)+STEPS

END DO
```

The program sketched above is able to compute an optimized test profile from any augmented usage profile. However, the time complexity of this program is unsatisfactory. Neglecting the two approximation loops (`max_k` and `max_s`) the complexity of part 1 is `O(NP*|X|)` where `|X|` is the average length of a sampled path; the complexity of part 2 is `O(NP * A)` and that of part 3 is `O(N+A)`. Since `A` is typically strictly larger than `N` and `NP` is several orders of magnitude larger than `A` the complexity of part 2 is most significant in the program.

The solution of a typical medium-sized problem, such as the Outbound Call Center discussed in Section 5, takes already several hours on a state of the art desktop computer, thus rendering the solution of bigger problems almost impossible within the narrow bounds of a testing budget. Moreover, test plans may be modified very often during the development phase but even in the deployment phase of a software product. Hence, the optimization would have to be applied after every change in the software.

In the following section we discuss how the complexity of part 2 can be (theoretically) reduced to `O(A)` through the exploitation of high performance computing techniques. This reduced complexity allows an application of the optimization even to large problems.

4 Parallelization

4.1 Parallelization Strategy

Measurements conducted with the sequential program presented in Section 3 have shown that up to 98 % of the elapsed time is spent in part 2 where the estimate of the partial derivative is computed. Therefore we focus our initial parallelization effort on this part of the code.

The execution time spent in this part is mainly due to the computation of the dot product of Equation 4. Since this computation is applied to every path and all the paths are independent from each other, we can apply the computation of the dot product to each of the paths in parallel. The computation of the partial results (`HELP`) can be parallelized accordingly. The final step in computing the estimate of the partial derivative requires the accumulation of these partial results from all the paths. Although any processor may accumulate the values of all the paths that are mapped to it independently, an information exchange among all processors is required. This collective operation may result in expensive communication and synchronization and therefore requires special care as we describe below.

In a typical problem the number of paths is in the range of several thousands, hence this parallelization strategy offers a degree of parallelism that is sufficient for an efficient exploitation of standard parallel machines that are composed of several hundreds of processors. Assuming that `NP` processors are available we can reduce the complexity of this part from `O(NP*A)` to `O(A)`. This is a drastic reduction since `NP` is typically several orders of magnitude bigger the `A`. Note however, that the communication overhead implied by the reduction operation may degrade the performance gains. Moreover, the number of available processors will typically be significantly lower than `NP`.

Another source of parallelism is the random generation of the paths in matrix `M` performed in part 1. Again, the generation of paths is independent and can be done in parallel.

In order to exploit parallelism on any given parallel architecture it is necessary to distribute the program's work (and if the processors do not have access to a shared memory also the program's data) to the processors on which the program will execute. A common approach is the *Single Program Multiple Data (SPMD)* or *data parallel* paradigm where all the processors execute the same program however on different parts of the data. The data of a program is distributed over the available processors and any processor owning data elements that are defined (i.e. they are assigned a value) perform the associated computation. This is known as the *owner-computes-rule*. For a more detailed discussion see for instance [2, 21].

A data parallel implementation of the program discussed in Section 3 along these lines requires the distribution of the set of all the paths over the available processors. With this data distribution the parallelism potential of the first and second phase can be fully exploited. However, the final computation of the estimation of the partial derivative including the summation of the partial results from all the sampled paths requires non-local data accesses that need to be minimized in order not to trade off parallelism gains for communication time which may be extremely expensive, in particular on loosely coupled Beowulf style clusters [14]. Apart from this data exchange all computation can be performed locally on a processor; no further communication is required. A parallel execution of the modification of transition probabilities is not enabled by this distribution because the path matrix is not defined in this part of the code. Since this part of the program consumes less than 0.5 % of the total computation time and no distributed data that might cause expensive communication is required in this part, we decided not to parallelize it.

4.2 Implementing the Parallel Algorithm in HPF

A common procedure for implementing parallel programs (especially on distributed memory machines) is to employ low level message passing systems such as MPI [10] in conjunction with a standard sequential language such as Fortran or C. With this approach explicit communication calls for non local data accesses have to be inserted into the sequential program. Manual parallelization by means of MPI is however a cumbersome and error prone task since the programmer has to deal with all details of parallelism and synchronization explicitly. This involves data distribution, index conversion to take data distribution into account, communication, synchronization, etc. Moreover, code maintenance is exacerbated since two version of the code, a sequential and a parallel one which might differ significantly, have to be maintained. In light of the disadvantages of manual parallelization we decided to employ a high level data parallel language: *High Performance Fortran (HPF)* [6].

4.2.1 HPF Overview

HPF is a data parallel extension of Fortran 90 [7] that was designed to provide the user with a high-level interface for programming scientific applications, while delegating to the compiler the task of generating an explicitly parallel (in most cases message-passing) program. HPF has an SPMD execution model and provides the programmer with a logical single thread of control, a global name space, and loosely synchronous parallel computation. The main extensions of HPF to Fortran 90 are *data distribution features*, *data parallel execution features*, and an *extended intrinsic functions and standard library*.

In particular, HPF defines compiler directives for mapping data on arrays of virtual processors, data parallel directives, and data parallel language constructs, some of which have been adopted in Fortran 95 [8] (for instance `FORALL` loops). Since most HPF constructs are compiler directives that are ignored by a sequential compiler only one code version needs to be maintained.

In the following we describe the features of HPF employed within our program in more detail:

Data Distribution. HPF employs a two-stage data distribution model where data arrays are *aligned* with *templates* which are subsequently distributed onto virtual *processor arrays*. A one dimensional model omitting templates is supported, as well. Virtual processor arrays are specified with the `PROCESSORS` directive. An intrinsic function `NUMBER_OF_PROCESSORS()` may be used to determine the size of a processor array dynamically at program start. An `ALIGN` directive allows to align data arrays with templates or with other data arrays and a `DISTRIBUTE` directive finally specifies a distribution mechanism for mapping templates or data arrays to processor arrays. The most important distribution mechanisms provided by HPF are `BLOCK` and `CYCLIC(M)` distributions. A `BLOCK` distribution partitions a data array into (almost) equally sized blocks of contiguous data where the number of blocks is equal to the number of available processors. Each such block is mapped to one processor. In a `CYCLIC(M)` distribution a data array is partitioned into chunks of size `M` which are distributed in a round-robin fashion to the processors. The default value of `M` is 1.

Parallelism. The main source of parallelism in HPF are parallel loops. Since static compiler analysis cannot detect parallelism in all cases, the user can guide parallelization by giving appropriate hints to the compiler. So the `INDEPENDENT` directive indicates that a loop can be parallelized. This directive can be used in conjunction with `NEW` and `REDUCTION` clauses. The `NEW` clause asserts that the specified variables are private to each iteration of the loop; that is, a separate copy is used within each iteration. The scope of `NEW` variables is restricted to the loop body. Reduction operations, such as summation, need to be marked explicitly. This gives the compiler the possibility to replace the involved parts of the code by optimized system calls.

4.2.2 Data Structures

The parallelization strategy outlined above requires to distribute all paths (i.e. the first dimension of matrix `M`) to the available processors. Although the computational effort for modifying paths may differ significantly due to the random path lengths, several hundreds to thousands of paths are mapped to a processor and we can expect the accumulated time spent in computing on all the local paths not to differ that much among processors. Therefore we decided to distribute `M` in a block-wise manner.

This distribution implies an according distribution of the temporary arrays that are associated with the paths, in particular `PH`, `DP`, and `HELP`. If those arrays were replicated instead of distributed all the distributed data of `M` would have to be broadcasted which would introduce a large communication overhead. We will touch this issue below.

The following code excerpt illustrates the HPF directives for specifying the data distribution. Note that we do not make use of templates but distribute data directly to the processors. For specifying the distribution of the temporary arrays we use the `ALIGN` directive to reflect their connectivity with matrix `M`.

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
      ...
      INTEGER, ALLOCATABLE :: M(:,:)
      DOUBLE PRECISION, ALLOCATABLE :: PH(:)
      DOUBLE PRECISION, ALLOCATABLE :: DP(:)
      DOUBLE PRECISION, ALLOCATABLE :: HELP(:)
!HPF$ DISTRIBUTE(BLOCK,*) ONTO P :: M
!HPF$ ALIGN PH(:) WITH M(:,*)
!HPF$ ALIGN DP(:) WITH M(:,*)
!HPF$ ALIGN HELP(:) WITH M(:,*)
```

4.2.3 Control Structures

Part 1 In order to parallelize the first part of the algorithm we need to mark the loop over all the paths as `INDEPENDENT`. Auxiliary variables required within this loop (such as `PROB`, `H`, and

NEW_POSITION) need to be marked as NEW so that each iteration of the loop may work with a private copy of this data. An ON HOME clause is used to force the owner-computes-rule.

```
       M = 0.D0
!HPF$ INDEPENDENT, ON HOME(M(I,:)), NEW(PROB, H, NEW_POSITION)
       DO I = 1,NP
          PROB = 1.D0
          DO WHILE (sample_new_element())
             NEW_POSITION = ! position of a newly generated path element
             PROB = PROB * P(NEW_POSITION)
             M(I,NEW_POSITION) = M(I,NEW_POSITION)+1
          END DO
          H = ! dependability measure
          PH(I) = PROB * H
       END DO
```

An HPF compiler will be able to fully parallelize this code without any synchronization and communication since all distributed data is aligned perfectly with the variable specified in the ON HOME clause (M) and all distributed data is accessed in the same manner. Note, however, that procedures called from within INDEPENDENT loops (such as sample_new_element) are required to be PURE in the Fortran 95 sense, i.e. they are not allowed to have certain side effects.

If in this loop array PH, for instance, would have been replicated instead of distributed perfectly aligned with M the result of computing PROB * H would have to be broadcasted in each loop iteration. This communication overhead would probably completely outweigh the performance gains obtained through parallel execution.

Part 2 The first two loops of part two can be parallelized accordingly to part 1 by marking the loops over the paths as INDEPENDENT:

```
       DP = 1.D0
       DO I = 1,A        ! loop over edges
!HPF$ INDEPENDENT, ON HOME(DP(J))
          DO J = 1,NP   ! loop over paths
             DP(J) = DP(J) * EXP(((-1)*M(J,I)) * LOG(T(I)))
          END DO
       END DO

!HPF$ INDEPENDENT, ON HOME(HELP(I))
       DO I = 1,NP       ! loop over paths
          HELP(I) = DP(I)*PH(I)
       END DO
```

Again, all distributed data is perfectly aligned and both loops can be fully parallelized by the compiler without any communication overhead.

The third loop of part 2 requires, however, collective communication in that information from all distributed paths is needed to compute D. HPF provides the REDUCTION attribute such that the compiler is able to use optimized functions for accomplishing this task. However, the loop over all paths is in this part nested within the loop over the edges (remember that this loop layout is advantageous wrt. cache usage). With such a loop layout an expensive reduction operation would be required for each iteration of loop I. By interchanging these two loops we can reduce this overhead to one reduction operation. Since communication overhead is several orders of magnitude higher than the overhead introduced by cache misses it is profitable to apply this transformation which results in the following code:

```
      D = 0.D0
!HPF$ INDEPENDENT, ON HOME(HELP(J)), REDUCTION(D)
      DO J = 1,NP   ! loop over paths
         DO I = 1,A       ! loop over edges
            D(I) = D(I) - HELP(J)*M(J,I)/T(I)
         END DO
      END DO
```

With this coding, all partial results for D stored on any processor can be accumulated by means of a single collective communication operation.

Part 3 Since we do not exploit any parallelism in the third part the code remains unchanged wrt. the sequential implementation discussed above.

5 Experiments

In this section we validate the efficiency of our approach by means of two examples: a small synthetic online train informations system and a medium-size real world system representing an outbound call center which is used by the Austrian Red Cross. We give a brief overview of these systems, present the numeric results of our analysis, and discuss the effectiveness of our parallel implementation.

Both examples were ran on a state-of-the art supercomputer, the NEC Cenju-4 consisting of 64 VR10000 processors connected via a fast multistage interconnection network, and on a medium-sized Beowulf cluster comprised of 8 dual-board PCs equipped with Pentium II processors and connected via Fast-Ethernet. On the Cenju-4 we used the research HPF compiler VFC [1] developed at the University of Vienna. On the Beowulf cluster we used VFC as well as the commercial PGHPF compiler from the Portland Group [16]. For analyzing the effectiveness of our parallel algorithm we measured the speedup as well as the efficiency obtained on varying numbers of processors. Speedup is defined as

$$S_p = T_{seq}/T_p \tag{7}$$

where S_p is the speedup obtained on p processors, T_{seq} is the measured sequential execution time and T_p is the parallel execution time on p processors. Efficiency is defined as

$$E_p = S_p/p \tag{8}$$

For measuring the sequential execution time we compiled our HPF program with native Fortran 90 compilers available on any architecture. Note that HPF directives are only comments to Fortran and thus ignored by a sequential compiler.

5.1 Online Train Information Applet

Description. Figure 5 shows a small synthetic example of an online train informations system that was first introduced in [3]. When a user visits the train information system the applet is initialized by the browser and the *initial screen* (node 2) is shown to the user. After logging in, the *main menu* (node 3) is presented. Starting from this window it is possible to call the submenu *information request* (node 4) or the submenu *booking places* (node 8). In the submenu *information request* the user can select among different informations —*departure information* (node 5), *arrival information* (node 6), or *board service information* (node 7) — from the information database. From this node as well as from the *main menu* and the *initial screen*, the user can exit the system. When the user proceeds to the submenu *booking places* (node 8) he has the possibility to select the menus *preferences* (node 9), *place reservation* (node 10), or *acknowledgment* (node 11).

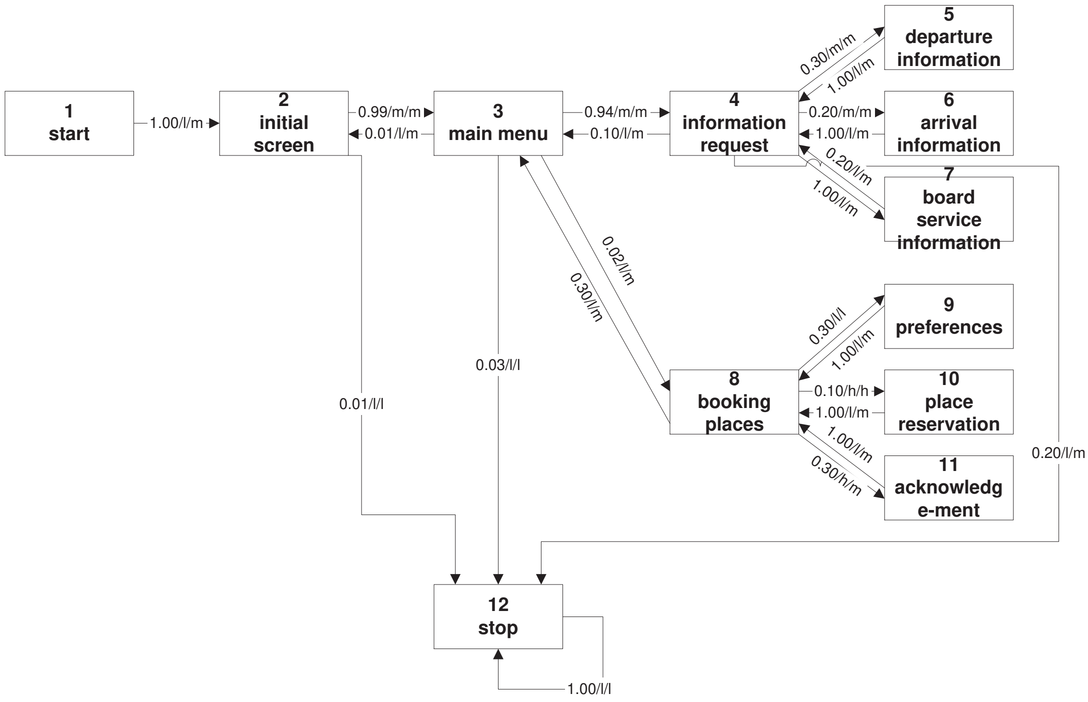


Figure 5: Train Info System

Numerical Analysis. We have applied our program to this example with the following parameters: `NP=1,000`, `max_s=100`, and `max_k=50`. Furthermore, we used the following constant values for indicating the failure rates and expected losses:

Function	l	m	h
f(i,j)	0.001	0.01	0.1
l(i,j)	1	10	100

The results of our analysis are presented in Table 1. It can be observed that the intended effect of favoring the test of critical operations has been achieved. The generation of test sequences according the operational profile of $\mathcal{P}$ gives the critical function (8,10) only a small probability of being tested (note that already (3,8) has a small probability). Within the optimized test profile the transition (8,10) (and accordingly transition (3,8)) obtain a fair chance.

Runtime Analysis. This small problem illustrates the classical tradeoff between performance gains through parallel execution and communication. Since the system consist only of 12 nodes with 23 arcs and only 1,000 paths have been sampled, the computation time spent in each processor is quite low (in fact, the sequential execution time is only about 1.5 minutes) and the communication overhead due to the reduction operation in part 2 becomes quickly dominant.

Table 2 shows the speedup and Table 3 the respective efficiency. While on the Cenju the fast communication network allows to use up to 16 processors efficiently the slow communication network in the Beowulf system degrades performance already on 4 processors (2 PCs). Using all available 16 processors on the Beowulf yields only an efficiency of 26% while on the Cenju 64 processors can at least be used with an efficiency of 62%.

5.2 Outbound Call Center

Description. In this example we model the marketing database of the outbound call center used by the Austrian Red Cross. To increase transparency, we have partitioned the entire MUM

	1	2	3	4	5	6	7	8	9	10	11	12
1	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00	.00	.00
2	.00	.00	.99	.00	.00	.00	.00	.00	.00	.00	.00	.01
3	.00	.01	.00	.83	.00	.00	.00	.11	.00	.00	.00	.05
4	.00	.00	.11	.00	.31	.20	.20	.00	.00	.00	.00	.18
5	.00	.00	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00
6	.00	.00	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00
7	.00	.00	.00	1.0	.00	.00	.00	.00	.00	.00	.00	.00
8	.00	.00	.29	.00	.00	.00	.00	.00	.15	.28	.28	.00
9	.00	.00	.00	.00	.00	.00	1.0	.00	.00	.00	.00	.00
10	.00	.00	.00	.00	.00	.00	1.0	.00	.00	.00	.00	.00
11	.00	.00	.00	.00	.00	.00	1.0	.00	.00	.00	.00	0.0
12	.00	.00	.00	.00	.00	.00	.00	.00	.00	.00	.00	1.0

Table 1: Optimal Test Transition Probabilities of the Train Info System

	Cenju	**Beowulf**	
Processors	VFC	VFC	PGHPF
1	1.00	0.99	1.00
2	1.97	1.93	1.94
4	3.95	3.16	3.19
8	7.89	4.23	4.09
16	15.52	4.23	4.48
32	23.65	—	—
64	39.87	—	—

Table 2: Speedup of the Train Info System

	Cenju	**Beowulf**	
Processors	VFC	VFC	PGHPF
1	1.00	0.99	1.00
2	0.99	0.97	0.97
4	0.99	0.79	0.80
8	0.99	0.53	0.51
16	0.97	0.26	0.26
32	0.74	—	—
64	0.62	—	—

Table 3: Efficiency of the Train Info System

description into a set of 9 diagrams. Due to space limitation we restrict our discussion to three diagrams, the *Main Section* (cf. Figure 6), the *Customer Outbound Processing* (cf. Figure 7), and the *Administrator Functions* (cf. Figure 8). Entry node and exit node of a subsection are represented in the Main Section and are repeated in the subsection. These connection nodes are shaded in the figures.

The main functionality is *Customer Outbound Processing* (node 6 and the subsequent part of the graph given in Figure 7). By means of customer processing, the operator can select a potential new or an already registered customer, and the customer's data can be seen on the screen (node 8). There is also further information about the customer available: information on the last phone calls with the customer (node 10), the complete chronology of interactions with him (node 11), the medical history (node 7), and the donation history (node 12).

As soon as the operator knows the important facts about the potential customer he will, with a certain probability, proceed to perform a phone call (node 14). After the call the operator can modify customer data, e.g. register an telephone number change. If there are no changes the operator can go directly to a function where he can feed the call results into the computer (node 9).

In subsection *Administrator Functions* (node 123 in Figure 8), the three main functionalities *medical expert* (node 131), *distribute campaigns* (node 124), and *maintain campaigns* (node 128) are implemented. One task of the supervisor is to distribute the work to the available operators (node 126). Certain customers are grouped to so-called campaigns, and each campaign is assigned to an operator. In the functionalities of *maintain campaigns* (node 128), new campaigns are created (node 129) and old campaigns are deleted (node 130). The customer data of unsuccessful calls can also be grouped to new campaigns (node 137). In the subsection *medical expert* all the

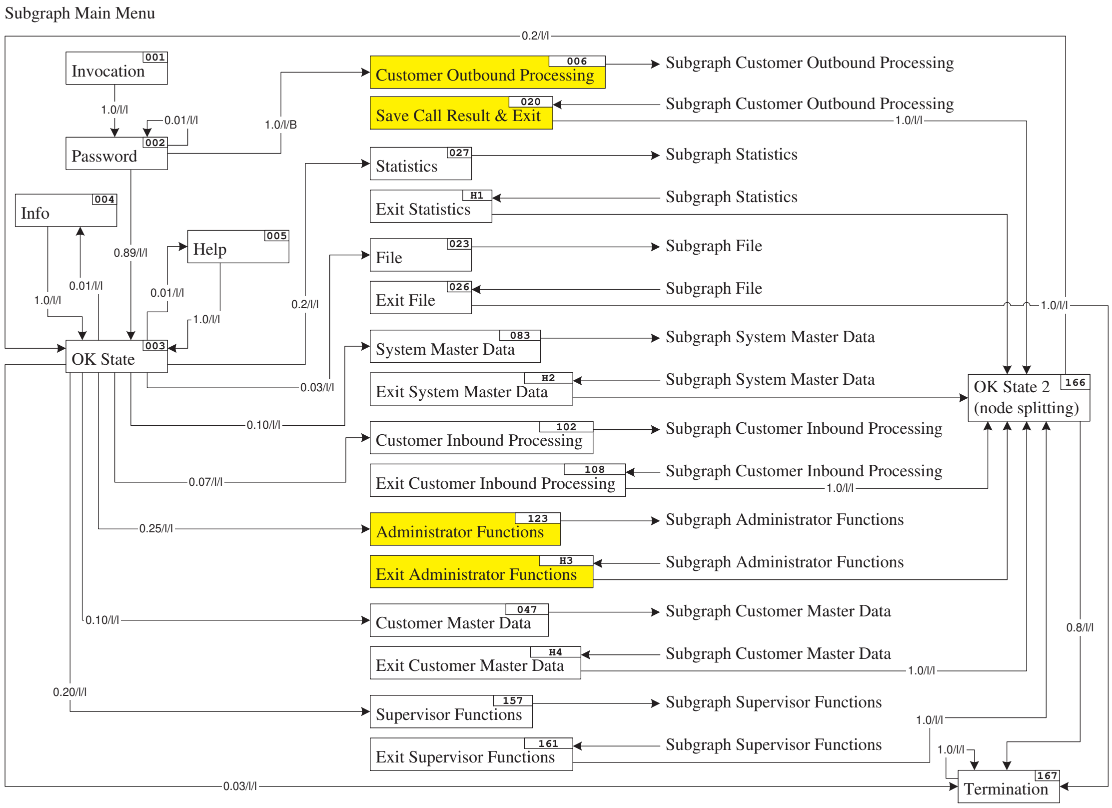


Figure 6: A MUM of an Outbound Call Center - Main Section

customer data of donors which have a medical problem are stored (node 139). These customers will be called by a medical expert. Therefore, the same functionality like in outbound processing to handle a call with a customer is available.

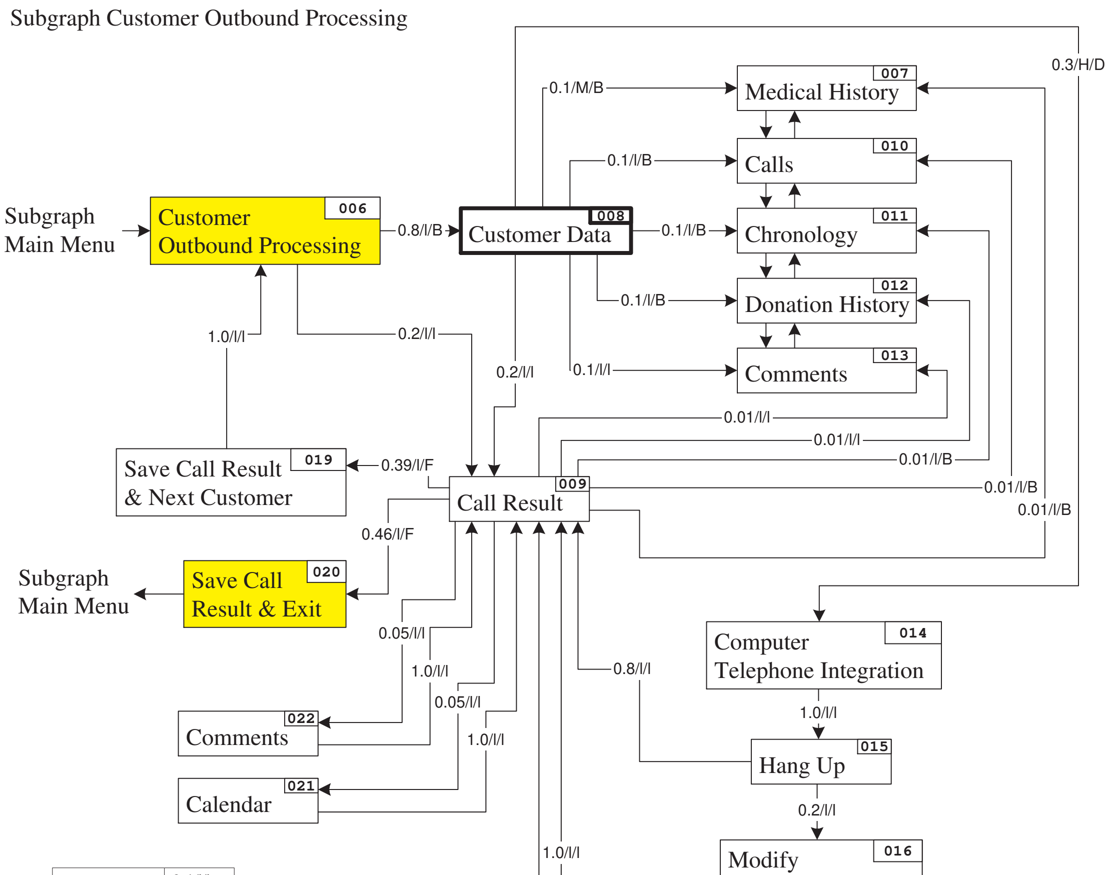
Numerical Analysis. We have applied our program to this example with the following parameters: `NP=10,000`, `max_s=100`, and `max_k=100`. Furthermore, we used the following constant values for indicating the failure rates and expected losses:

Failure rate: l=0.001; M=0.01; H=0.1;
Loss: l=1; A=2; B=500; C=800; D=1000; E=5,000; F=10,000.

As in the previous example, failure rates and losses in case of failure have been estimated. In the following we will focus our discussion of the numerical results to three specific nodes: node 8 *Costumer Data* (cf. Figure 7), node 128 *Maintain Campaigns*, and node 133 *Inbound Costumer Selection* (both within Figure 8). These nodes are marked with a bold box in the figures.

A high failure rate is for instance expected for function (8,14) which builds the connection to the telephone system. Also the loss in case of failure is high for this function, since it is not possible to perform a phone call if the computer telephone integration does not work properly. We also expect a high failure rate as well as a high loss in case of failure for function (133, 136) since it uses additional information from an error-prone legacy system. *Inbound Customer Selection* (node 133) is reached from *Maintain Campaigns* (node 128) therefore we report also the computed test transition probabilities for this node.

Table 4 reports on the optimal transition probabilities for the selected nodes 8, 128, and 133; the original probabilities are given embraced. It can be immediately observed that the probability



007 => 008	0.1/I/I	
007 => 009	0.3/I/I	
007 => 010	0.15/I/B	
007 => 011	0.15/I/B	
007 => 012	0.15/I/B	
007 => 013	0.15/I/I	

010 => 008	0.1/I/I
010 => 009	0.3/I/I
010 => 007	0.15/I/B
010 => 011	0.15/I/B
010 => 012	0.15/I/B
010 => 013	0.15/I/I

011 => 008	0.1/I/I
011 => 009	0.3/I/I
011 => 007	0.15/I/B
011 => 010	0.15/I/B
011 => 012	0.15/I/B
011 => 013	0.15/I/I

012 => 008	0.1/I/I
012 => 009	0.3/I/I
012 => 007	0.15/I/B
012 => 010	0.15/I/B
012 => 011	0.15/I/B
012 => 013	0.15/I/I

013 => 008	0.1/I/I
013 => 009	0.3/I/I
013 => 007	0.15/I/B
013 => 010	0.15/I/B
013 => 011	0.15/I/B
013 => 012	0.15/I/B

Figure 7: A MUM of an Outbound Call Center - Subsection Customer Outbound Processing

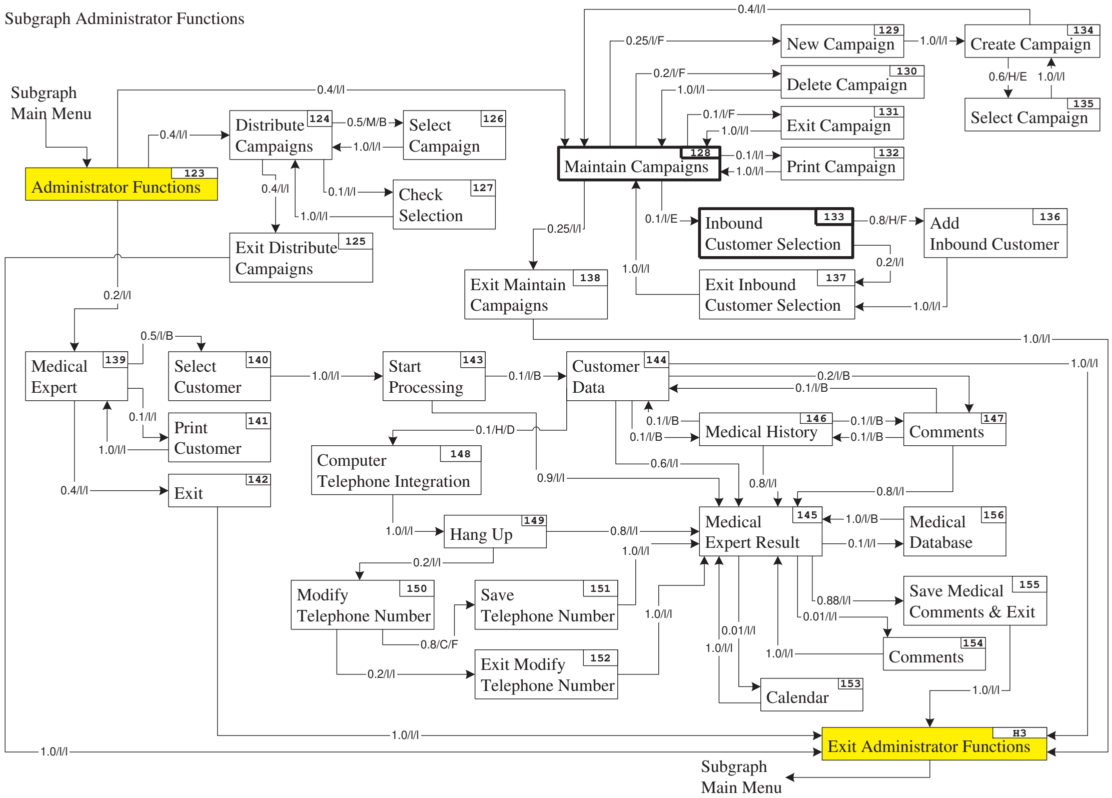


Figure 8: A MUM of an Outbound Call Center - Administrator Functions

of the critical functions (8, 14), and (133, 136) has been increased. Due to the fact that transition (133, 136) is critical the probability of transition (128, 133) is increased, as well.

Runtime Analysis. Our previous discussion of the synthetic train info system revealed that small systems cannot be optimized efficiently on a large number of processors because of the dominant communication overhead. The medium range call center system consists, however, of 167 nodes with 337 arcs and the path sample size was 10,000. Hence, the computation time of this system is large enough to exploit all processors available on both machines efficiently, as can be seen in Table 5 and Table 6 reporting on the speedup and the efficiency, respectively. By exploiting the massive parallelism provided by the Cenju we were able to reduce the runtime of this analysis from almost 7 hours (the sequential program) to less than 7 minutes (on 64 processors). This significant reduction allows to optimize test plans even for larger systems.

	7	9	10	11	12	13	14
8	.10 (.10)	.17 (.20)	.10 (.10)	.10 (.10)	.10 (.10)	.09 (.10)	0.34 (0.30)

	129	130	131	132	133	138
128	.26 (.25)	.20 (.20)	.10 (.10)	.09 (.10)	.18 (.10)	.17 (.25)

	136	137
133	.89 (.80)	.11 (.20)

Table 4: Optimal Test Transition Probabilities of the Call Center System

	Cenju	Beowulf	
Processors	VFC	VFC	PGHPF
1	0.99	0.99	1.00
2	1.98	1.88	1.95
4	4.01	3.89	3.94
8	7.95	8.03	8.04
16	15.97	15.95	15.97
32	29.04	—	—
64	62.10	—	—

Table 5: Speedup of the Call Center System

	Cenju	Beowulf	
Processors	VFC	VFC	PGHPF
1	0.99	0.99	1.00
2	0.99	0.94	0.98
4	1.00	0.97	0.99
8	0.99	1.00	1.00
16	1.00	1.00	1.00
32	0.91	—	—
64	0.97	—	—

Table 6: Efficiency of the Call Center System

6 Conclusions

Statistical software testing is increasingly employed in the software development cycle. However, obtaining test plans that reflect reality to a sufficient extent is a challenging task. Methods have been devised that allow to optimize test plans wrt. some goal such as minimizing risk. Although these techniques exist for quite some while, they are currently not applied in commercial systems, mainly because of their computational complexity.

In this paper we showed that these stochastic optimization methods may well be applied to large scale software systems if high performance computing techniques are exploited. Specifically, we explored the parallelism potential provided by these optimization methods and discussed a prototype implementation. For implementing the parallel algorithm we used a high level parallel language (HPF) that allows to easily port the application to a wide range of parallel machines.

We showed the efficiency of our techniques by optimizing a real world software system. The exploitation of parallel computing resulted in a drastic decrease of runtime spent in the optimization; both on a high end parallel machine as well as on a more cost effective PC-cluster. These results are encouraging and allow the optimization of even larger systems within the narrow time-frames of a testing budget. Furthermore, the techniques described in the paper are general enough such that they may be successfully employed to other application fields such as regression and partition tests. We plan to experiment with these applications in the near future.

Acknowledgments

The authors would like to thank Walter J. Gutjahr for his support throughout the work on this paper and for discussing details of his approach. Furthermore we are grateful to Jesse H. Poore for valuable comments on a draft of this paper.

References

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

[2] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufman, 1999.

[3] W. J. Gutjahr. Failure Risk Estimation via Markov Software Usage Models. In E. Schoitsch, editor, *SAFECOMP 96, Proc. of the 15th International Conference on Computer Safety, Reliability and Security*, pages 183–192. Springer, 1997.

[4] W. J. Gutjahr. Importance Sampling of Test Cases in Markovian Software Usage Models. *Probability in the Engineering and Informational Sciences*, 11:19–26, 1997.

[5] W. J. Gutjahr. Software Dependability Evaluation Based on Markov Usage Models. *Performance Evaluation*, 40:199–222, 2000.

[6] High Performance Fortran Language Specification Version 2.0, January 1997.

[7] ISO. Fortran 90 Standard. ISO/IEC 1539 :1991 (E), 1991.

[8] ISO. Fortran 95 Standard. ISO/IEC 1539 :1997, 1997.

[9] J. C. Laprie. Dependability - Its Attributes Impairments and Means. In B. Randell, J. C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*. Springer, Washington, 1998.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.1*, June 1995.

[11] K. Murty. *Linear Complementary, Linear and Nonlinear Programming*. Heldermann, Berlin, 1988.

[12] J. H. Poore and C. J. Trammell. Application of Statistical Science to Testing and Evaluating Software Intensive Systems. In *Statistics, Testing, and Defense Acquisition*. National Academy Press, Washington, 1998.

[13] S. J. Prowell et al. *Cleanroom Software Engineering: Technology and Process*. The SEI Series in Software Engineering. Addison Wesley Longmann, Reading, Massachusetts, 1998.

[14] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proceedings IEEE Aerospace Conference*, 1997.

[15] K. Sayre and J. H. Poore. Partition testing with usage models. *Information and Software Technology*, 42(12):845–850, 2000.

[16] The Portland Group, Inc. *PGHPF User's Guide*, 2000.

[17] M. Ujaldón, E. Zapata, S. Sharma, and J. Saltz. Parallelization Techniques for Sparse Matrix Applications. *Journal of Parallel and Distributed Computing*, 38(2):256–266, November 1996.

[18] G. H. Walton, J. H. Poore, and J. Trammell. Statistical Testing of Software Based on a Usage Model. *Software – Practice and Experience*, 25(1):97–108, 1995.

[19] J. A. Whittaker and J. H. Poore. Markov Analysis of Software Specification. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106, 1993.

[20] J. A. Whittaker and M. G. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[21] H.P. Zima and B.M. Chapman. Compiling for Distributed-Memory Systems. In *Proc. of the IEEE, Special Section on Languages and Compilers for Parallel Machines*, pages 264–287, February 1993.

A Notation

Symbol	Description	Code equivalent
$\mathcal{G}$	Graph	-
V	Set of nodes in $\mathcal{G}$	-
A	Set of arcs in $\mathcal{G}$	-
n	Number of nodes in $\mathcal{G}$	N
$-$	Number of arcs in $\mathcal{G}$	A
X	Sampled path	-
np	Number of sampled paths	NP
P	Probability of a path	PROB
H	Dependability measure with respect to X	H
M	Arc traversal frequency of the sampled paths	M
$\mathcal{P}$	Usage profile augmented with	-
p	Propabilities	P
f	Failure rates	F
l	Expected loss	L
$\mathcal{T}$	Test profile	T
G	Weighted expected dependability value	-
D	Partial derivative $\partial G(T)/\partial t(i,j)$	D
$-$	First term in Equation 4	PH
$-$	Dot product in Equation 4	DP
j_{min}	Minimum D	J_MIN for a specific node