Statistical Simulations on Parallel Computers

Hana ŠEVČÍKOVÁ

The potential benefits of parallel computing for time-consuming statistical applications are well known, but have not been widely realized in practice, perhaps in part due to associated technical obstacles. This article develops a simple framework for programming statistical simulations using parallel processing, which does not require changing programming language or forgoing the use of standard statistical libraries. The basic idea of using parallel computing for statistical simulation studies is straightforward in principle, and is based on the standard master-slave model. However, there are several technical obstacles that can make it difficult to implement in practice. These include: nonreproducibility of results due to variations in the distribution of random numbers among processes, creation of excessive numbers of slaves, proliferation of slaves with very short lifetimes, and slaves destroyed due to hardware failures. This article proposes solutions for each of these difficulties, and together these solutions constitute an overall parallel computing framework for statistical simulation studies.

In an experiment with 15 processors, the methods detailed here led to increases in speed by factors that can actually exceed the maximum expected factor of 15, due to the efficiencies of the proposed problem decomposition methods. Different gains may be achieved with different strategies, depending on the problem decomposition used and heterogeneity of the processors. Fault tolerance is an important feature of the framework. In an experiment with faults, a non-fault-tolerant version of our method took almost twice as long, and did not produce any results, while the fault-tolerant method dealt efficiently with the faults.

We conclude that parallel computing can greatly improve the efficiency of statistical computation without greatly increasing programming complexity, and that it deserves wider investigation for such applications. Software to implement the proposed framework in R is available from http://www.stat.washington.edu/hana.

Key Words: Distributed systems; Fault tolerance; Master-slave model; Mixture models; Parallel processing; Random number generation; Statistical applications.

1. INTRODUCTION

Rapid technological development, particularly in the areas of processors and networking, now allows computationally expensive applications to run in times that would not have

Hana Ševčíková is Research Associate, Department of Statistics, University of Washington, Box 354322, Seattle, WA 98195-4322 (E-mail: hana@stat.washington.edu).

Journal of Computational and Graphical Statistics, Volume 13, Number 4, Pages 1–21
DOI: 10.1198/106186004X12605

been conceivable some years ago. This is due in part to more powerful processors, but the sharing of resources has also played a major role in saving processor time. Adapting programming to the available capacity of resources can increase this effect even further.

The benefits of parallel computing are well known and have been documented in the literature for years. In mathematical and statistical research, parallel processing allows scientists to address more complex problems. There are many publications about parallel processing for statistical computing mostly concerning specific methods and applications; see Adams, Kirby, Harris, and Clegg (1996) for a review. Nevertheless, programs for parallel computers in statistics have not been as widely developed as would seem worthwhile given the huge time cost of many statistical applications. Adams et al. (1996) gave some possible reasons for the limited use of parallel computers in statistics, such as belief in the difficulty of programming and hence a need for extra programming skills, the lack of libraries of parallel routines, the wide variety of parallel architectures, and the lack of wide availability of parallel computers.

This article presents a simple framework for programming statistical simulations for parallel processing. The focus is on hardware architecture available to most scientists: a cluster of workstations or personal computers. Moreover, the suggested approach requires neither changing the accustomed programming language, nor forgoing the use of standard libraries such as NAG.

We focus specifically on the problem of simulation studies for assessing the properties of statistical tests and estimators, because this is an area where sequential studies can be very time-consuming and where there is a clear potential for large gains in computation time from parallel computing. However, we believe that the ideas developed here may well be more generally applicable.

The basic idea of using parallel computing for statistical simulation studies is straightforward in principle, and is based on the standard master-slave model. However, there are several technical obstacles that can make it difficult to implement in practice. These include: nonreproducibility of results due to variations in the distribution of random numbers among processes, creation of excessive numbers of slaves, proliferation of slaves with very short lifetimes, and slaves destroyed due to hardware failures. We propose solutions for each of these difficulties, and together these solutions constitute an overall parallel computing framework for statistical simulation studies. Fault tolerance is an important feature of our framework. In an experiment with faults, we found that a non-fault-tolerant version of our method took almost twice as long, and did not produce any results, while our method dealt efficiently with the faults.

Section 2 gives a brief overview of parallel processing, including a short description of hardware and software aspects. Section 3 focuses on statistical simulations, their common structure, and ways to decompose it for parallel processing. The issue of random numbers—as well as making applications tolerant of system faults—is also discussed. Section 4 applies our framework to a concrete application, namely the repeated execution of a chain of statistical tests with bootstrap simulations.

2. SETTINGS FOR PARALLEL PROGRAMMING

2.1 HARDWARE

The possibility of processing programs in parallel depends most of all on the available hardware. For example, splitting a program up into a bunch of independent parts would not increase its speed on a PC with a single CPU.

For a rough theoretical description of hardware architecture, Flynn's classification (Flynn 1972) has been widely used in the literature. It distinguishes between four broad categories: SISD (single instruction single data), MISD (multiple instruction single data), SIMD (single instruction multiple data), and MIMD (multiple instruction multiple data). Most of today's parallel computers work according to the MIMD principle. It describes a computer with several processing units capable of operating on several data streams which could be instructions or data.

A further classification of the MIMD model according to the memory organization is useful, especially from the programmer's point of view:

- Systems with *shared memory*: These consist of a set of processors that share access to a common memory area. An example could be a personal computer with only few CPUs (usually two or four), up to supercomputers with several hundred processors. Process communication on these systems is usually done via shared variables.
- Systems with *distributed memory*: These consist of a set of processors each of which has its own local memory. An example of this category is a cluster of workstations or personal computers connected via a local area network (LAN), which are able to communicate with each other with the help of appropriate software. Processes exchange data via message passing.

Because the cost of parallel supercomputers is still enormous, a usual case of a hardware platform for scientists is a system of the latter type, in most cases a cluster of linked independent computers.

2.2 SOFTWARE

Having a parallel hardware architecture available, the programmer must be aware of the supporting software, such as the operating system, available compilers and libraries for parallel processing. This knowledge will determine his or her view of the system and will affect his or her approach to parallel programming.

Concerning the operating system, computers with shared memory are normally equipped with system software that manages cooperation between all processors and takes care of load balancing. There are also usually software constructs available that help programmers to manage concurrent access to the shared memory, for example, semaphore and lock-unlock mechanisms.

Such support from the operating system is usually not available in systems with dis-

tributed memory. A network of personal computers will need additional software to build a cluster of processors that can communicate with each other and share the network load. An example of such software is MOSIX (Barak, Guday, and Wheeler 1993), a freeware package for Unix/Linux platforms. This is able to migrate running processes automatically between its nodes according to the current resource usage and thus to share the network load to get the best performance.

Another important issue for the programmer is choosing the programming language. This goes along with choosing the program decomposition approach. There are different levels of parallelism:

- *Instruction level*—single instructions can be processed concurrently.
- *Data level*—the same instruction is processed on different data at the same time (mostly operations on data arrays).
- *Loop level*—a block of instructions from a loop can be run for each loop iteration in parallel.
- *Procedure level*—larger program parts (mostly program procedures) can be run concurrently.

Each of these levels leads to a different program grain size. Grain size refers to the number of instructions performed in parallel. Choice of grain size involves a tradeoff: a fine-grained program provides considerable flexibility with respect to load balancing, but it also involves a high overhead of processing time due to a larger management need and communication expense. Thus, the available hardware architecture has to be taken into consideration. For example, a program running in parallel at the instruction level would not benefit much in a system with distributed memory where the cost for data transfer via a network is high relative to the runtime of the instruction itself.

In addition, the following differentiation has to be considered:

- *Implicit parallelism*: The programmer writes a sequential program and uses an advanced compiler to create a parallel executable binary.
- *Explicit parallelism with implicit decomposition*: The programmer marks constructs for parallel processing. Then, the compiler does not need to analyze data dependencies as in the previous case but it saves the programmer from dealing with process communication, and thus it allows the easy creation of programs of fine grain size. Representatives of this class are extensions of standard sequential languages as Fortran D, HPF (High Performance Fortran), C*.
- *Explicit parallelism*: The programmer handles the decomposition of the algorithm as well as the communication between processes. The advantage of this type of parallelism is that one can use a standard sequential programming language extended by an appropriate software library such as PVM—parallel virtual machine (Geist et al. 1994) or MPI (Dongarra, Hempel, Hey, and Walker 1994). This way of writing programs is the most labor intensive and therefore useful on the procedure level of parallelism. Nevertheless, it offers the possibility of creating very efficient code.

The design proposed here focuses on systems with distributed memory. As mentioned above, this is the kind of platform that is available to most statisticians. Moreover, programs written for a computer cluster will work on a multiprocessor computer as well. For this reason, the message-passing paradigm will be used for process communication.

2.3 PROGRAM STRUCTURE

Parallel programs consist of a set of tasks that are processed on a set of processors. To get the correct algorithm the tasks have to be coordinated properly. There are different organizational forms of tasks. We will focus on models that are common in systems with distributed memory. In Geist et al. (1994), the following models are distinguished:

- *Crowd computation*—a collection of closely related processes, typically executing the same code, perform computations on different portions of the workload. This can be further subdivided into two categories:
  - *Master-slave model*—one process has the "controlling" function. It is responsible for program input, initialization, spawning slave processes, collection and display of results. Slaves perform the actual computation.
  - *Node-only model*—there is no master, only autonomous processes. One process takes over the noncomputational responsibilities in addition to contributing to the computation itself.
- *Tree model*—processes are created in a tree-like manner, thereby establishing a tree-like, parent-child relationship. It can be useful for sorting algorithms, for example.
- *Hybrid model*—this is a combination of the crowd and tree models.

Choosing one of these models is strongly application dependent. It should best match the natural structure of the parallelized program.

3. STATISTICAL APPLICATIONS

As mentioned earlier, among statisticians parallel computing has not become widely used, even though the availability of hardware and software platforms for parallel processing has increased rapidly in recent years. Moreover, there is a large need for parallel computing: in modern statistics, there are often time-consuming methods (e.g. analysis of mixture models) with theoretical properties that are not fully known. These properties can be analyzed by simulation. However, an adequate number of replications in such a simulation experiment often requires an extremely long runtime if it is done sequentially. Thus, parallel processing of simulations is often the only way to get reasonable application runtime.

Our experiences with parallel simulation computing show that only a little effort is needed to get large returns in terms of saving computation time. This effort does not require changing the accustomed programming language; also, we keep using standard libraries (such as NAG) as well as sequential generation of random numbers. The only issue is

how to decompose the problem into parallel tasks and handle their communication and management. Because many statistical simulations have the same or very similar program structure, this work can be done only once and may be used for a wide variety of other simulation studies.

3.1 SIMPLE PARALLEL STRUCTURE OF STATISTICAL SIMULATIONS

Many simulation programs in statistics are of the following rough structure:

```
begin program
   read input; initialization
   for (i=1,...,r) do
       generate random sample
       compute results(i)
   end for
   evaluate results
   print output
end program
```

The line `compute results(i)` represents computation of any complexity—from simple expressions up to time-intensive branching simulations. In general, there is a set of random numbers and the same computation is made on their partitions which are of the same size (samples). Consequently, one can decompose the program along the data partitions and thus, we get one task per replication. Typically the number of replications `r` is large, and so the suggested decomposition will usually provide sufficient flexibility in terms of load balancing. Furthermore, the number of tasks will scale with `r`, although not with the sample size. We will show a possibility of further decomposition (e.g., for bootstrap simulations) in the example of Section 4.

The structure of the program and its decomposition is a typical master-slave model. The master process will handle input, initialization, creating and managing slave processes, collecting and evaluating results, and printing the output. A slave receives parameters from the master, performs the actual computation and sends the result back to the master. The issue of generating random numbers will be discussed in Section 3.2. For now, we assume that each slave is able to generate its own random sample. Thus, the sequential program will be split into two parts:

Master:

```
begin program
   read input; initialization
   for (i=1,...,r) do
       spawn slave s_i
       send parameters to s_i
   end for
   receive results(1,...,r)
```

```
   evaluate results
   print output
end program
```

Slave:

```
begin program
   receive parameters from master
   generate random sample
   compute result
   send result to master
end program
```

This design would not be very efficient in practical implementation. It has the following drawbacks:

- In the normal case, many more slaves will be created than there are processors available. The time overhead for the frequent process switching of the CPUs involved will degrade the system performance. An optimal mapping of processes to processors has to be achieved (see Section 3.3).
- For short computations in the replication loop, slaves will have very short lifetimes. Then, their creation and destruction could take even more time than the computation itself. An alternative solution minimizing the number of slaves being created is discussed in Section 3.3.
- The master has no knowledge about slaves being destroyed due to hardware failures. Some basic administration functions should be implemented in the master routine to be able to dynamically adopt to system faults. This will be discussed in Section 3.4.

3.2 ISSUE OF RANDOM NUMBERS

In the decomposition embodied in the typical master-slave model of Section 3.1, slaves that are carrying out their tasks at the same time require access to random numbers. One of the main desired properties of a random number generator (RNG) is the reproducibility of random numbers. This is guaranteed using a sequential generator. In a parallel application, this issue demands extra attention.

Foster (1995) distinguished three general approaches to the generation of random numbers on parallel computers:

- *Centralized* approach—A sequential generator is encapsulated in a task from which other tasks request random numbers. Adopted to our algorithm, the master would create a sample and send it (together with other parameters) to the spawned slave. This solution guarantees reproducibility and is very easy to implement. On the other hand, increasing sample size causes increased network traffic and could lead

to overflow of communication buffers.

- *Replicated* approach—Each task has an instance of the same RNG. To guarantee reproducibility and independence, each slave has to get exact instructions from the master about the generating, for example, the replication number. This approach would not be very efficient in the parallel design from Section 3.1 having a large number of replications. Each slave would have to start generating samples from replication 1 and continue to the replication number received from the master. It leads to time cost of complexity $O(nr^2)$ with sample size n and number of replications r. Nevertheless, it could be used in applications with time-consuming single computations where, by comparison, the generation time itself is negligible.
- *Distributed* approach—From a single generator many generators are derived that are all responsible for generating a single sequence. There exist several techniques for implementing distributed generators. In Srinivasan and Mascagni (2000), a freeware library for parallel random number generation SPRNG is introduced.

In the first two approaches, standard sequential RNGs may be used. In our implementations, we do not want to abandon a usage of the large amount of RNGs and diverse transformations implemented in the NAG library. Section 3.3 introduces an improved program design with a relatively small number of slaves g, properly mapped into the number of available CPUs. In this case, we can use the replicated approach for generating random numbers, because by keeping g constant, the time complexity can be reduced to $O(nr)$.

For using sequential RNGs in branching simulations, we introduce a combination of the centralized and replicated approach:

- *Quasi-replicated* approach—There is an instance of the same RNG in each task. Only one task (mostly the master process) generates the whole set of random numbers for the application. The set is decomposed into subsets, each for one task. Instead of sending the actual random numbers, only a state of the RNG corresponding to the first number of the appropriate subset is sent to a task. Thus, each task is able to generate its random numbers starting from the received state.

The last approach avoids the large communication cost of the centralized approach as well as it reduces the high time cost of the pure replicated approach. Although the size of the subsets must be known, it does not appear as a disadvantage in usual simulations with finite replication loops. We will use this method in the example of Section 4. Nevertheless, if the number of random numbers required on the slaves is unknown and could be very large, the use of a distributed generator such as SPRNG might give better performance.

3.3 IMPROVED IMPLEMENTATION DESIGN

Here, the simple parallel decomposition suggested in Section 3.1 will be improved. The goals of the improvement are:

- To control the number of running processes, so that the system will not get overloaded. For this, we introduce a variable `max_degree` for the maximum number

of processes being executed in parallel and a variable `cur_degree` for the current number of such processes. We allow `max_degree` to be changed during the execution, so that the user can adopt the parallelism degree of the simulation according to the current system load (e.g., increase the degree for running during the night and decrease it back in a time of high load). The master reads this value during its waiting time from a file (which can be changed at any time). Extensions, such as a monitoring routine that creates this file automatically depending on the system load, are conceivable.

- To minimize the total number of spawned slaves by keeping the current degree of parallelism as high as possible. This is based on the idea of spawning `max_degree` slaves that will share the work on the whole set of tasks. Each time a computation is finished, the slave does not exit after sending its results to the master but waits for the next message from the master. This is either a request to destroy itself or parameters for a new computation.
- To solve the issue of generating random numbers as discussed in Section 3.2. Each slave keeps the state of its RNG in the local variable `local_repl` in terms of the replication number that its last computation corresponds to. With every other computation request, it will continue generating samples from `local_repl` to the current replication i. Each slave will receive the value of i from the master (together with other computation parameters).

Master:

```
begin program
   read input; initialization
   read max_degree; cur_degree:=0
   finished:=0     ! counts finished replications
   for (i=1,...,r) do
10     if (cur_degree ≥ max_degree) read max_degree
       if (cur_degree ≥ max_degree) then
15           receive results(j) from s_k ! wait for a finished slave
             finished:=finished+1
             if (cur_degree > max_degree) then
                   send exit request to s_k
                   cur_degree:=cur_degree−1; go to 10
             end if
       else     ! there is free capacity for additional slave
             spawn slave s_k
             cur_degree:=cur_degree+1
       end if
             send param(i,...) to s_k
       end for
       for (i=finished+1,...,r) do     ! collect remaining results
20           receive results(j) from s_k
```

```
            send exit request to s_k
        end for
        evaluate results
        print output
    end program
```

Slave:

```
    begin program
       initialization of RNG
       local_repl:=0
    30 receive message from master
       if (message ≠ param(i,...)) exit program
       generate (i−local_repl) random samples
       compute result with the last sample
       send result to master
       local_repl:=i
       go to 30
    end program
```

All receive-calls are implemented as blocking receive routines, that is, the program waits until a message arrives. In line 15 and 20 of the master program, the master must be able to identify the slave the message came from. Usually, standard message-passing libraries offer functions to get information such as process ID. Additionally, the master must receive the value of j which denotes the replication number the computation belongs to. It can either be sent by the slave (together with results), or alternatively, the master can hold a table which maps process IDs to replication numbers. We will show the latter solution in the next section.

3.4 CONSIDERATION OF HARDWARE FAILURES

Executing parallel applications across distributed clusters introduces the problem of fault tolerance. The more processors are involved in the computation, the smaller the mean time to failure. With applications that are fault tolerant against system failures (one can just think of rebooting computers), one avoids an annoying need for multiple executions of the same simulations. Especially in statistical simulations where a computation is made up to several hundred thousand times, results will not suffer from a small number of them being discarded.

A great deal of work has been done in the area of fault tolerant programming. Most of it is based on checkpointing the entire state of the program to disk. Hough, Kolda, and Torczon (2001) introduced an interesting parallel algorithm that incorporates recovering from failures without checkpointing. Nevertheless, it is constrained to a specific class of methods, namely pattern search for nonlinear optimization.

Because we are interested in developing a framework that is independent of the particular application, and thus of the slave program, we suggest letting the master take care of indicating failures and recovering from them. In the program design from Section 3.3, the master does not have any knowledge about slaves being alive or not. For example, if just one slave were to crash during the execution, the program would get stuck in line 20 and never come to an end. We introduce simple management functions for the master to be able to react to system faults without losing other results.

For this, we suggest that the master hold two tables—one for running processes (called `slaves_alive`) and one for processes found to be aborted (`slaves_dead`). Both tables are of the same data structure: one row per slave, one column for the slave's task identifier (TID), and one column for the corresponding replication number. Depending on the application, there could be additional columns; for example, for information used for time measurement or, in branching simulations, for the number of a slave's children. In the following, because of their simplicity, we will forgo detailing step-by-step functions for manipulating the management tables. Instead, we give an informal brief overview of the tables. We present a concrete use of the tables in the application of Section 4 and in more detail in the Appendix available at http://www.amstat.org/publications/jcgs/ftp.html.

There should be functions implemented that handle:

- adding new slaves to the table (`Add_slave`)
- removing slaves from the table (`Remove_slave`)
- modifying values in a certain row (`Write_slave`)
- searching for replication number of a certain TID (`Find_replication`)
- searching for the TID of a certain replication number (`Find_TID`)

For the table `slaves_alive`, a call of the routine `Add_slave` will be joined with each spawn-call, whereas a call of the routine `Remove_slave` will be associated with sending an exit request to a slave. Because the value of `cur_degree` corresponds to the number of entries in `slaves_alive`, `Add_slave` increases and `Remove_slave` decreases this value. `Write_slave` should be called when parameters for a new computation are sent to a slave without new spawning. After the master receives a result-message from a slave of a certain TID, it will use the `Find_replication` routine to find the corresponding replication number.

During its waiting time, in addition to reading the value of `max_degree` from a file, the master can search for slaves being aborted (we call this function `Find_dead_slaves`). To identify such processes, the routine may use a standard library function, such as `pvm_task` from the PVM-library, giving information about running tasks. Alternatively, it can determine such processes from the data in `slaves_alive`; for example, from the replication number or from the start time of slaves if this information is kept in the table. A slave that has been determined as aborted is removed from `slaves_alive` (with `Remove_slave`) and added to the table `slaves_dead` (with `Add_slave`). This could be useful in situations where a determination was incorrect and a slave moved to `slaves_dead` was still working. The remaining number of entries in `slaves_alive` is the new value of `cur_degree`. Then, if the program is not successful in the routine `Find_replication` called for `slaves_alive`,

the same routine should be called for `slaves_dead`. If `d` is the number of entries in `slaves_dead`, the loop for collecting remaining results in the master program will be done for `(r-finished-d)` iterations. Alternatively, `d` new computations can be started. If new slaves are spawned in such case and they receive the failed replication numbers, with our solution of generating random numbers, the reproducibility of the results will be guaranteed.

Figure 1 presents a flow chart for the fault-tolerant master program. The additional management functions discussed in this section are shaded. From Figure 1 it is obvious that the program structure consists of two main parts: the main replication loop and a loop for collecting the remaining results. It should be noted that this program structure is independent of the actual computation and thus may be used for any other simulation of sequential structure discussed in Section 3.1. Furthermore, a usual programming language can be used for implementation. There are only a few message-passing calls (shown in Figure 1 as bold framed boxes) for which a message-passing library is needed. For example, one might use PVM coupled with C, C++, or Fortran. PVM can also be used with higher-level programming languages. For instance, an interface to PVM is available for R, called RPVM (Na Li and Rossini 2001).

Detecting an aborted slave as soon as possible will significantly influence the runtime of the simulation. To demonstrate this on an example, we carried out the following experiment. Our simulation was based on repeating 100,000 times a computation that takes three seconds on average. We had 10 (in this case homogeneous) CPUs available at the time of starting the process. After four hours processing we reduced the number of participating processors to five; this represents the likely effect of an increase in system load due to other users' processes. We considered two situations, one with no system faults during the whole processing time, and another with system faults during the first hour of processing that led to discarding two slaves. The simulation was processed in each of these situations in two versions: the non-fault-tolerant version from Section 3.3 and the fault-tolerant version from this section.

The results are summarized in Table 1. As can be seen, there is a significant difference in the runtime of the non-fault-tolerant version in the two fault situations. Because the master is not aware of any lost slave, if there are system faults, it will make a usage only of eight and later of three CPUs. Moreover, it delivers no results, because at the end (here after 22 hours, 45 minutes) it will put itself in the waiting position for the nonexisting slaves. The large runtime difference between the two program versions in the case of failures shows that even if the programmer implements the last receive routine of the non-fault-tolerant version with finite timeout, there is a huge inefficiency in that solution.

In a situation without any failures, there is almost no difference between the two versions. Thus, the time overhead for managing the fault tolerance is negligible. Apparently, the delay in the case of failures in the fault-tolerant version is caused by the time during which the aborted slaves have not yet been identified. Therefore, if aborted slaves are guessed in the routine `Find_dead_slaves` from their runtime or replication number, a more rapid identification will improve the simulation runtime. On the other hand, the more slaves are guessed falsely, the more overload will be caused.

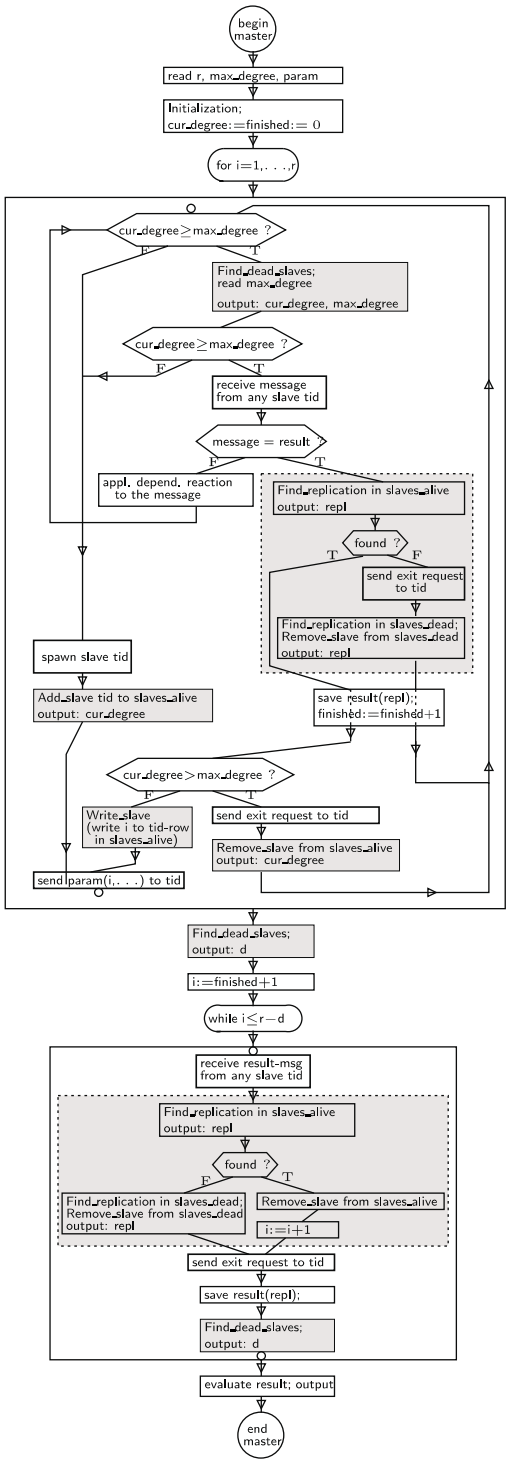

Figure 1. Flow chart for a fault-tolerant master program.

Table 1. Comparison of the Two Program Versions

Program version	*Run without faults*	*Run with two faults*	
	time (h:min)	*time*	*getting results*
non-fault-tolerant	13:23	22:45	no
fault-tolerant	13:26	13:47	yes

Despite the obvious effectiveness of the fault-tolerant strategy, the method is not robust against failures of the master node. This is a deficiency of current implementations of the message passing framework rather than of the strategy adopted here. Therefore, this drawback may be remedied by the next generation of message passing methods.

4. APPLICATION: EVALUATING THE STATISTICAL PROPERTIES OF A CHAIN OF NESTED TESTS

This section applies the discussed parallel program structure to a statistical application that is extremely time-consuming. We are dealing with performing a sequence of statistical tests each of which requires an inner bootstrap simulation and with analyzing the properties of such a test by simulation.

4.1 STATISTICAL PROBLEM

Let $\mathcal{F}$ be a family of probability densities and let $\mathcal{F}_1 \subset \mathcal{F}_2 \subset \mathcal{F}_3 \subset \cdots \subset \mathcal{F}$ be a sequence of hypotheses about $f \in \mathcal{F}$ such that $\mathcal{F} = \bigcup_{k=1}^{\infty} \mathcal{F}_k$ holds. Given a sample $x_1, \ldots, x_n$ from a particular (unknown) $f \in \mathcal{F}$, we would like to identify the number $k(f)$ such that $f \in (\mathcal{F}_{k(f)} \setminus \mathcal{F}_{k(f)-1})$ (here we set $\mathcal{F}_0 = \emptyset$).

For example, $\mathcal{F}$ may represent the family of all finite mixtures of distributions of a particular parametric model. If $\mathcal{F}_k$ stands for the family of all mixtures with no more than k components, then $k(f)$ is the number of components of a particular mixture f.

Let us suppose that $k(f)$ is estimated on the basis of a chain of tests of the hypothesis $\mathcal{F}_k$ against the alternative $\mathcal{F}$ (shortly "$\mathcal{F}_k \leftrightarrow \mathcal{F}$"), starting with $k = 1$.

For each k, the test is based on a test statistic T_k. A level α-test would reject $\mathcal{F}_k$ if the particular realization t_k of T_k exceeds the $(1-\alpha)$-quantile $c_{k,1-\alpha}$ of the distribution of T_k. The distribution of T_k and therefore also $c_{k,1-\alpha}$ depends on $f \in \mathcal{F}_k$. Suppose that we can find a likelihood estimator $\hat{f}_k$ of f in the family $\mathcal{F}_k$. Then, $c_{k,1-\alpha}$ could be estimated by simulation on the basis of a number rb of bootstrap samples $y_1^i, \ldots, y_n^i$, $i = 1, \ldots, rb$, from $\hat{f}_k$. Thus, the test consist of the following steps:

1. Estimate $\hat{f}_k \in \mathcal{F}_k$ and calculate t_k on basis of the sample $x_1, \ldots, x_n$.
2. For $i = 1, \ldots, rb$
   - simulate a bootstrap sample $y_1^i, \ldots, y_n^i$ from $\hat{f}_k$
   - calculate t_k^i.

3. Estimate $\hat{c}_{k,1-\alpha}$ from $t_k^1, \ldots, t_k^{rb}$.
4. Reject $\mathcal{F}_k$ if $t_k > \hat{c}_{k,1-\alpha}$.

Each test results in a decision $s \in \{0, 1\}$, where $s = 0$ denotes rejection of $\mathcal{F}_k$, whereas $s = 1$ denotes acceptance of $\mathcal{F}_k$ (in a very wide sense). Now, testing $\mathcal{F}_k$ against $\mathcal{F}$ for $k = 1, 2, \ldots$ on basis of one sample $x_1, \ldots, x_n$ from an unknown f until for some first $\hat{k}$, $\mathcal{F}_{\hat{k}}$ is not rejected, results in a sequence $0, 0, 0, \ldots, 1$; with "0" in the positions $1, \ldots, \hat{k} - 1$ and "1" in the position $\hat{k}$. We will need this representation later. Finally, $k(f)$ is estimated by $\hat{k}(f) = \hat{k}$.

In order to study the statistical properties of the estimator $\hat{k}(f)$ by simulation, the whole sequence of tests has to be repeated r times on the basis of simulated samples $x_1^i, \ldots, x_n^i$, $1 \leq i \leq r$, from a pregiven $f \in \mathcal{F}$. Then, we have an outer loop representing the replications $i = 1, \ldots, r$, and for each replication an inner loop representing the chain of tests based on $x_1^i, \ldots, x_n^i$, including the bootstrap samples $y_1^j, \ldots, y_n^j$, $j = 1, \ldots, rb$, for each test. If calculation of $\hat{f}_k$ and t_k is time-consuming, parallel processing seems worth considering.

In the context of mixtures, McLachlan and Peel (2000) proposed estimating the number of components by a similar sequence of tests "$\mathcal{F}_k \leftrightarrow \mathcal{F}_{k+1}$." Here, t_k is based on the likelihood estimators $\hat{f}_k$ and $\hat{f}_{k+1}$ of f under $\mathcal{F}_k$ and $\mathcal{F}_{k+1}$. For calculating $\hat{f}_k$ and $\hat{f}_{k+1}$ the EM algorithm is proposed which is very time-consuming. We modified this proposal by testing "$\mathcal{F}_k \leftrightarrow \mathcal{F}$" on the basis of $\hat{f}_k$ under $\mathcal{F}_k$ and the nonparametric maximum likelihood estimator $\hat{f}$ of f under $\mathcal{F}$. Here too, calculating t_k requires a considerable amount of time.

Remark: Suppose, there is a rough initial estimate $\hat{k}_0$ of $k(f)$ (e.g., in mixture models, the number of components of the nonparametric maximum likelihood estimator $\hat{f}$ of f in $\mathcal{F}$). Then, we can start the chain of tests from $\hat{k}_0$ instead of $k = 1$ and thus, decrease the searching time. In this case, we proceed as follows:

1. Perform test "$\mathcal{F}_{\hat{k}_0} \leftrightarrow \mathcal{F}$."
   (a) If $\mathcal{F}_{\hat{k}_0}$ is accepted, perform tests "$\mathcal{F}_k \leftrightarrow \mathcal{F}$" for $k = \hat{k}_0 - 1, \hat{k}_0 - 2, \ldots, 1$ until $\mathcal{F}_k$ is rejected. Then, $\hat{k}$ is the last k for which "$\mathcal{F}_k \leftrightarrow \mathcal{F}$" was accepted.
   (b) If $\mathcal{F}_{\hat{k}_0}$ is rejected, perform tests "$\mathcal{F}_k \leftrightarrow \mathcal{F}$" for $k = \hat{k}_0 + 1, \hat{k}_0 + 2, \ldots$ until $\mathcal{F}_k$ is accepted and the last k is $\hat{k}$.

4.2 PARALLEL STRUCTURE

4.2.1 Decomposition

As mentioned in Section 3.1, the number of tasks created by decomposition along the replication loop will scale with the replication number `r`, but not with the sample size or with the number of executed single tests in a chain of tests. Consequently, a single task could be extremely time-consuming so that further decomposition and thus, a finer grain size of the program can be advantageous. In the context of our simulation study, two levels of parallelism are possible:

1. one task per loop iteration (as described in Section 3.3),
2. one task per single test "$\mathcal{F}_k \leftrightarrow \mathcal{F}$".

We obtain a tree-like structure of tasks with three levels: (1) master task as root; (2) its slaves responsible for processing a chain of tests each; they are parents of tasks on the (3) level that handle single tests (we call such process slave-child). The relationship between the master and its slaves will be handled according to Sections 3.3 and 3.4. This means that there is a certain number of spawned slave processes each of which performs computations of chains of tests, one after another, whereby the master supervises the work sharing between the slaves. Additionally, slaves receive from the master a value of the maximum number of children they are allow to create (say `max_child`). Thus, the master can control the total current degree of parallelism.

Between slaves and their children, we choose a dynamic spawn-destroy relationship. This is because of the simplicity of such implementation, because slaves often do not need the whole capacity of `max_child` for the whole computation time. Moreover, we are dealing with time-consuming computations and therefore there is no imbalance between the time needed for the computation and the time overhead due to process creation and destruction.

If a slave has fewer than `max_child` children, it informs the master of the actual value of its children. The master keeps the numbers of children of each slave in an extra column of the management table `slaves_alive` and the total current degree of parallelism `cur_degree` is given as the sum of all the values in this column. Then a new computation can be started only if `cur_degree` $\leq$ `(max_degree-max_child)`. For example, if `max_degree=9` and `max_child =3`, there could be three slaves running, each of which has three children. Or, there could be also five slaves running, one with three, two with two children, and the remaining two with only one child. In the latter situation, if the next finished slave had one or two children, it will be destroyed because there is no capacity for starting a computation with three children.

4.2.2 Random Numbers

It follows from the above discussion that in this solution, slaves can be more often destroyed than in the design of Section 3.3 where it was the user's responsibility. Thus, we will lose the advantage of the reduced time complexity of generating random numbers in comparison to the replicated approach as discussed in Sections 3.2 and 3.3. Therefore, we decide to use the quasi-replicated approach. We assume that there is a maximum number of k to be tested, say $k_{\max}$. Then, with a sample size n and a number of bootstrap replications rb, one chain of tests requires at most $(n + k_{\max} \cdot rb \cdot n)$ random numbers which is thus the size of the data partitions for tasks on the first level of parallelism.

Figure 2 shows the scheme of generating random numbers in the three different instances of the same RNG. The master captures the state of its RNG at the beginning of each loop replication (represented by black dots) and sends it to a slave (together with other computation parameters). Then, it generates $(1 + k_{\max} \cdot rb)$ random samples of size n and continues managing its slaves until the next replication can be started. After a slave receives computation parameters (including the RNG state) from the master, it restores accordingly the state of its RNG and generates one sample of size n which is the testing sample. It spawns its children and sends them the same RNG state received from the master. Each slave-child

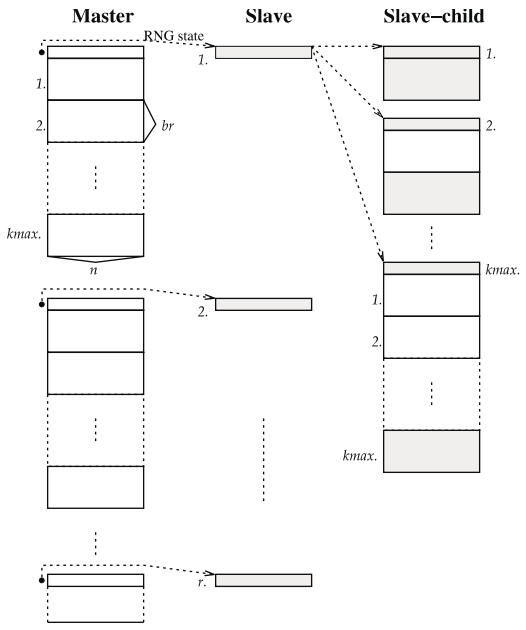

Figure 2. Generating random numbers on different levels of the tree-like structure.

responsible for test "$\mathcal{F}_k \leftrightarrow \mathcal{F}$" restores its RNG accordingly to the received state as well. It also creates a testing sample and before generating its own bootstrap samples, it creates $(k-1)rb$ nonused samples, that is, samples that are used by children responsible for tests with smaller k.

The width of all boxes in Figure 2 represents the unique sample size n; the two different heights represent the number of generated samples (either 1 or rb). Samples that are actually used by the process for a computation are shaded. The enumeration of slaves in the figure corresponds to r replications and not to the number of actually spawned slave processes. Note that even though slaves would not perform the testing to $k = k_{\max}$, and thus would not use all their random numbers, applying this scheme in all instances will guarantee reproducibility of the generated random numbers.

Suppose that there are system faults and that at the end of the replication loop, the master identifies d aborted slaves. It can either start new computations for $r+1, \ldots, r+d$, or it can repeat computations for the failed replications. In the latter case, it has to run the RNG once again (starting from the initialization) to achieve the corresponding RNG state. The first solution is easier to implement but it does not guarantee reproducibility of the results. The same holds if no action is taken and $d > 0$.

4.2.3 Parallel Searching Algorithm

Extending the sequence of tests defined in Section 4.1 to $k = 1, \ldots, k_{\max}$ should result in regular cases in a sequence s_k of decisions with $s_k = 0$ for $k < \hat{k}$ and $s_k = 1$ for $k \geq \hat{k}$. Then $\hat{k}$ is uniquely characterized by $s_{\hat{k}-1} = 0$ and $s_{\hat{k}} = 1$, and it is not really

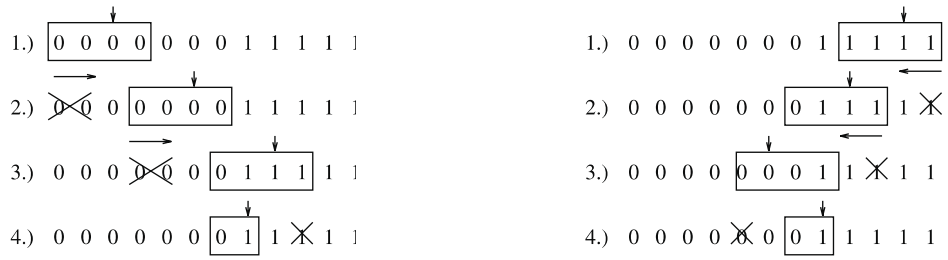

Figure 3. Examples of searching for $\hat{k}$ with a window of parallel units by moving to the right (right panel) and left (left panel).

necessary to perform all tests for $k < \hat{k}$. However, "irregular" sequences such as, for example, 000011010111 are also possible. In this case, we have a free choice between several possible estimators of $k(f)$. We decide to estimate $k(f)$ by any value $\hat{k}$ such that $s_{\hat{k}-1} = 0$ and $s_{\hat{k}} = 1$. This $\hat{k}$ is found by an appropriate windowing technique which supports parallel programming.

Starting from $k = 1$, we put a searching window of size `max_child` on the beginning of the sequence (on the positions 1,. . . ,`max_child`) and start the computation of s_i on these positions in parallel. The left panel of Figure 3 shows an example of such a scenario. Here, the vertical arrows mark the first finished s_i. In the first step, a zero is found and therefore the window can be moved to the right, up to s_{i+1} and all running computations on the left side can be discarded (step 2). As long as further zeros will be found, the window is moved further to the right in the same way (step 3). In the third step of the figure, $s_i = 1$ is found first. Thus, we can shrink the window by destroying computations on the right of s_i, since $\hat{k}$ must lie between the left side of the window and i. Although $\hat{k}$ is found in the fourth step, we need to wait for the remaining results in the window to ensure that a zero is a predecessor of $s_{\hat{k}}$.

Now, assume we have $\hat{k}_0$ and the knowledge that $\hat{k}$ is somewhere close to this value. Then we set an appropriate neighborhood $[k_1, k_2]$ so that $1 \leq k_1 \leq \hat{k}_0 \leq k_2 \leq k_{\max}$ and $k_2 - k_1 \leq$ `max_child`. That is, a searching window lies at the computation beginning between k_1 and k_2. If the first finished s_i is 0, the window moves to the right and the procedure continues as in the previous case. In the right panel of Figure 3, there is a situation in which $s_i = 1$ is returned as first value in the first step. Then the window will be moved to the left, discarding running computations on the right from s_i. This will be done as long as $s_i = 1$ is found (step 2). In the window of step 3, a zero is found, which means that our window contains $\hat{k}$ and it can be shrunk to locate the value. Analogously to the left panel, after getting $\hat{k}$ in step 4, computation of the remaining values has to be finished.

4.3 IMPLEMENTATION AND RESULTS

In the final implementation of a repeated execution of a chain of tests, the master program corresponds to the fault tolerant solution from Section 3.4. In the slave program, the searching algorithm from Section 4.2.3 is implemented. Moreover, it also administers

its local management table for its own spawned children. If an aborted child is detected, it executes the corresponding test again. The program of the slave-child performs the actual computation. See Appendix at http://www.amstat.org/publications/jcgs/ftp.html for detailed program listings.

Having a certain number of processors available and thus, a fixed value of `max_degree`, the question arises of how to choose the value of `max_child` for the degree of parallelism at the second level. Our experiences show that the choice depends strongly on the nature of the problem we are dealing with. There are two important criteria that must be taken into account:

1. Is the average computing time for a single test independent of k?

   The main speedup of the parallelism on the second level is based on aborting non-finished tests as described in Section 4.2.3. With a searching algorithm starting from $k = 1$ and computation time that increases with increasing k, the speedup benefit would be small, since there will be only rare abortions. With such application, a return will be made using the alternative searching algorithm starting from an estimate $\hat{k}_0$ and having some probability that $\hat{k} < \hat{k}_0$.

2. How many single tests are expected to be performed?

   Because there is some risk of carrying out tests that are not needed for the result and thus would not be performed by a sequential algorithm, the mean distance from the starting k to the expected $\hat{k}$ has to be considered. For example, if we start from $k = 1$ and expect $\hat{k} \in [2, 4]$, it would not make sense to set `max_child` much larger than 4, since we would thus increase the mentioned risk.

The following comparisons are based on a program of a repeatedly executed chain of tests where numbers of mixture components are estimated. The calculation of $\hat{f}_k$ is done via the EM algorithm and thus, the time increases with increasing k. We perform a sequential as well as a parallel alternative of this test. Moreover, we compare a parallel version with parallelizing only at the first level to versions with both levels running in parallel of different degrees. To guarantee an adequate comparison of the different program versions, we ensure that they are not influenced by irregular sequences as mentioned in Section 4.2.3 and therefore provide exactly the same result, that is, in all corresponding chains the same number of tests are executed. For this, we assume hypothetical $\hat{k} = 9$ in all replications (the program rejects $\mathcal{F}_k$ for all $k \leq 9$ and accepts $\mathcal{F}_k$ for all $k > 9$). Then, starting from $k = 1$, the mean number of started tests (but not necessary finished) per replication is 9. The number of replications is set to 50.

We also show how the dependence of computation time on k can influence the runtime of the whole application. We simulate the opposite effect of the time-dependence of k of our application by starting from $k = 9$ and executing tests until $\hat{k} = 1$ is found (the program accepts $\mathcal{F}_k$ for all k). Thus, the searching algorithm will yield to more test abortions than in the version that starts from $k = 1$.

In Table 2, the performance of the different program versions is compared. For the

Table 2. Performance Comparison of Different Program Versions with 15 Processors

Program version	*Time (h:min)*	*Speedup*	*Efficiency = speedup/15*
sequential	48:23	–	–
parallel 1 level	5:10	9.38	.63
↑ `max_child`=5	4:37	10.49	.70
↓ `max_child`=3	2:48	17.32	1.16
↓ `max_child`=5	2:06	23.11	1.54

four parallel programs, 15 processors of the same speed were used. The sequential version was run on one of these CPUs. We computed the absolute speedup as T_s/T_p, where T_s is the execution time of the sequential program and T_p the time needed for performing the parallel version on p processors. The absolute efficiency is then computed as speedup$/p$, where $p = 15$. The performance loss of the program with only one level running in parallel in comparison to the ideal efficiency ($= 1$) is due to communication and management cost that sequential program is spared. The third row in the table shows that even with additional communication and management overhead on the second level, further problem decomposition leads to increasing efficiency, having the same degree of parallelism. This program version (marked with ↑), starts from $k = 1$. Versions going in the opposite direction (marked with ↓) show even superlinear efficiency which is due to frequent test abortions. It is obvious from the table that choosing `max_child` closer to the expected number of tests to be executed, the efficiency increases.

It should be mentioned that our project of analyzing properties of a chain of tests in mixture models would hardly be possible without parallel computing.

The basic (application independent) framework introduced in this article has been adapted to R, using RPVM (Na Li and Rossini 2001) and rsprng (Na Li 2002). The resulting code (FPSS—framework for parallel statistical simulations) is available from the author's Web site http://www.stat.washington.edu/hana.

ACKNOWLEDGMENTS

This research has been supported by DFG grant UT340 and NSF grant 0134264. The author would like to thank the associate editor, the anonymous reviewer, Wilfried Seidel, and Adrian Raftery for helpful comments on this article.

[Received September 2002. Revised November 2003.]

REFERENCES

Adams, N., Kirby, S., Harris, P., and Clegg, D., (1996), "A Review of Parallel Processing for Statistical Computation," *Statistics and Computing*, 6, 37–49.

Barak, A., Guday, S., and Wheeler, R. (1993), *The MOSIX Distributed Operating System, Load Balancing for Unix*, volume 672 of *Lecture Notes in Computer Science*, New York: Springer Verlag.

Dongarra, J., Hempel, R., Hey, A., and Walker, D. (1994), "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications*, 8, 159–416.

Flynn, M. (1972), "Some Computer Organizations and their Effectiveness," *IEEE Transactions on Computers*, 21, 948–960.

Foster, I. (1995), *Designing and Building Parallel Programs*, Reading, MA: Addison-Wesley.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994), *PVM: Parallel Virtual Machine*, Cambridge, MA: The MIT Press.

Hough, P. D., Kolda, T. G., and Torczon, V. J. (2001), "Asynchronous Parallel Pattern Search for Nonlinear Optimization," *SIAM Journal of Scientific Computing*, 23, 134–156.

McLachlan, G., and Peel, D. (2000), *Finite Mixture Models*, New York: Wiley.

Na Li, M. (2002), rsprng [on-line]. Available at http://cran.r-project.org.

Na Li, M., and Rossini, A. (2001), "RPVM: Cluster Statistical Computing in R," technical report, University of Washington [on-line]. Available at http://www.analytics.washington.edu/Zope/projects/rpvm/.

Srinivasan, A., and Mascagni, M. (2000), "SPRNG: A Scalable Library for Pseudorandom Number Generation," *ACM Transactions and Mathematical Software*, 26, 436–461.