University of Helsinki
Department of Computer Science
Series of Publications C, No. C-1994-55

Experimental Evaluation of A Parallel Spectral Method for Run Length Control in Steady-State Simulation

Kimmo E. E. Raatikainen

Helsinki, November 1994

Report C-1994-55

University of Helsinki
Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

Experimental Evaluation of A Parallel Spectral Method for Run Length Control in Steady-State Simulation

Kimmo E. E. Raatikainen

Report C-1994-55
27 pages
Helsinki, November 1994

Abstract. Distributed and parallel simulation has been a popular research topic in recent years. The research has primarily concentrated on correctness and speedup of distributed simulation. The statistical output analysis that is an essential part of simulating stochastic systems has attained only small attention.

Parallel simulation is often mentioned as an attractive alternative for steady-state simulations. However, empirical results are not widely reported. We describe an implementation of an method that combines independent simultaneous replications and the spectral method. We also present experimental results are based on 12 simulation models executed in a network of Sun SPARCstations.

CR Categories and Subject Descriptors: I.6.6 [**Simulation and Modeling**]: Simulation Output Analysis — sequential estimation. I.6.8 [**Simulation and Modeling**]: Types of Simulation — parallel.

General Terms: Algorithms, Experimentation, Performance.

Additional Key Words and Phrases: Parallel simulation, Run length control.

Contents

1 Introduction

Since the fundamental contribution in Chandy and Misra [1981], a number of articles has been published covering various aspects of distributed and parallel simulation. The research has primarily concentrated on distributed simulation in which several processors cooperate on a single realization of the stochastic process simulated. The empirical studies [Baik and Zeigler 1985; Comfort 1984; Duda 1989; Fujimoto 1988; Nicol 1988a, 1988b; Reed 1985; Reed and Malony 1988; Reed *et al.* 1988; Reynolds and Kuhn 1987; Wagner and Lazowska 1989] have reported that typical speedups are only modest.

Parallel simulation in which each processor simulates an independent realization of the stochastic process is an attractive alternative for steady-state simulations [see e.g. Rego and Sunderam 1992; Lin 1994]. The attraction arises from the following two facts. Firstly, modern workstations connected through a local area network often have free processor capacity that could be utilized. Secondly, only minor modifications to uniprocessor simulation software are needed. The most notable drawback is that the size of the simulation model is restricted: Each processor must be able to simulate the whole model.

The research papers have considered primarily the correctness and speedup of the simulation. On the the other hand, statistical aspects of simulation have only attained minor attention. However, even in parallel simulation the study is only a programming exercise if we simulate a system having random input processes and if we do not properly carry out the analysis of output processes; see e.g. Pawlikowski [1990] or Schruben [1987]. The most common way of doing the output analysis is sequential run length control. The objective of run length control is to terminate the simulation as soon as the result is estimated to meet the given precision requirement.

In this paper we empirically examine one possible scheme for run length control in parallel simulation. The method examined combines the spectral method introduced in Heidelberger and Welch [1981] and the method of independent replications. We have two primary reasons for using the spectral method. Firstly, the spectral method is widely recognized as one of the most reliable and robust methods for run length control in uniprocessor simulation; see Pawlikowski [1990]. Secondly, we have learned that the parallelization of independent replications and the parallelization of regenerative method may lead to unacceptable precision. In addition, our preliminary results indicate that a parallel spectral method might be both reliable and efficient.

Our objective is to find out whether a bounded number of independent replications executed in parallel and the spectral method can provide estimates that are accurate enough. Our second objective is to examine practical speedups gained through the parallel execution.

In Section 2 we describe the parallel spectral method and give the outline of the implementation. In Section 3 we discuss our design principles and describe our implementation. In Section 4 we report empirical results based on 12 simulation models executed in a network of workstations.

2 Description of Parallel Spectral Method

2.1 Serial Spectral Method

In uniprocessor simulation the (serial) spectral method introduced in Heidelberger and Welch [1981] maintains non-overlapped batch means $\{\bar{X}_j\}_{j=1}^m$ of the output sequence $\{X_i\}_{i=1}^n$:

$$\bar{X}_j = \sum_{k=1}^{b} X_{(j-1)b+k} \big/ b\,,\ n = mb\,. \tag{1}$$

The number of batches (m) varies from M to $2M-1$. When the number of batches with the current batch size (b) increases to $2M$, the current batch size is doubled and the number of batches is reduced to M by averaging two adjacent batches.

The precision of $\hat{\mu}(n) = \sum_{j=1}^m \bar{X}_j \big/ m$ as the estimate of $\mu = \mathrm{E}\{X\}$ is examined at prespecified checkpoints $n_1, n_2, \ldots\,;\ n_{i+1} = I\,n_i$, where I is a prespecified constant, $1 < I \leq 2$. Most often the precision requirement is specified as the desired relative half-width of the estimated confidence interval. The estimated confidence interval is based on assumption that the distribution of $(\hat{\mu}(n) - \mu)\big/ s_{\hat{\mu}(n)}$ is the Student t-distribution with $\nu(n)$ degrees of freedom. The estimated relative half-width of the confidence interval for μ based on the sequence $\{\bar{X}_j\}_{j=1}^m$ is

$$\hat{\varepsilon}(n) = t_{\nu(n)}(1-\alpha/2)\, s_{\hat{\mu}(n)} \big/ |\hat{\mu}(n)|\,, \tag{2}$$

where $s^2_{\hat{\mu}(n)}$ is the estimated variance of $\hat{\mu}(n)$ with $\nu(n)$ degrees of freedom, $1-\alpha$ is the prespecified confidence level, and $t_{\nu(n)}(x)$ is the $100x^{\text{th}}$ percentile of the Student t-distribution with $\nu(n)$ degrees of freedom.

```
/* input parameters:
 *   desired_relhw:   desired relative half-width of the confidence interval
 *   conf_level:      confidence level
 */

    get_initial_seed(&seed);
    set_seed(seed);
    run_length = 0;
    do
    {        /* simulate to the next checkpoint */
        generate(&run_length, &nbatches, batch_means);
             /* estimate the mean and the relative half-width of the confidence interval */
        est_relhw = estimate(nbatches, batch_means, conf_level, &mean);
    } while (est_relhw > desired_relhw);
```

Figure 1: Generic code of uniprocessor simulation

The run length control terminates the simulation at the first checkpoint in which the estimated confidence interval is narrow enough. The termination point N is formally defined as

$$N = \text{smallest } n \in \{n_1, n_2, \ldots\} \text{ such that } \hat{\varepsilon}(n) \leq \varepsilon \ , \qquad (3)$$

where ε is the prespecified desired relative half-width of the confidence interval.

In the spectral method the estimated variance of $\hat{\mu}(n)$ is obtained through the periodogram of $\bar{X}_1, \ldots, \bar{X}_m$. A polynomial of order d is fitted to the first K ($K < M/4$) ordinates of the bias corrected logarithms of the smoothed periodogram. Periodogram ordinates are evaluated using Fast Fourier Transforms and smoothing is done by averaging two adjacent ordinates; for details, see Heidelberger and Welch [1981] or Pawlikowski [1990].

Figure 1 illustrates a typical code of uniprocessor simulation. The essential parts of the code are the model-dependent function `generate` and the model-independent function `estimate`. The function `generate` generates the observations from the current checkpoint (input value of the argument `run_length`, i.e. n_{i-1}) to the next checkpoint (output value of `run_length`, i.e. n_i). The number of batch means ($m_i = \lfloor n_i/b_i \rfloor$) is returned in the argument `nbatches` and the batch means (the $\bar{X}_j$'s, see Eq. (1), for $j = 1, \ldots, m_i$) in the array `batch_means`. The function `estimate` returns the estimated relative half-width of the confidence interval, that is $\hat{\varepsilon}(n_i)$ in Eq. (2), and the estimated mean, $\hat{\mu}(n)$, in the argument `mean`.

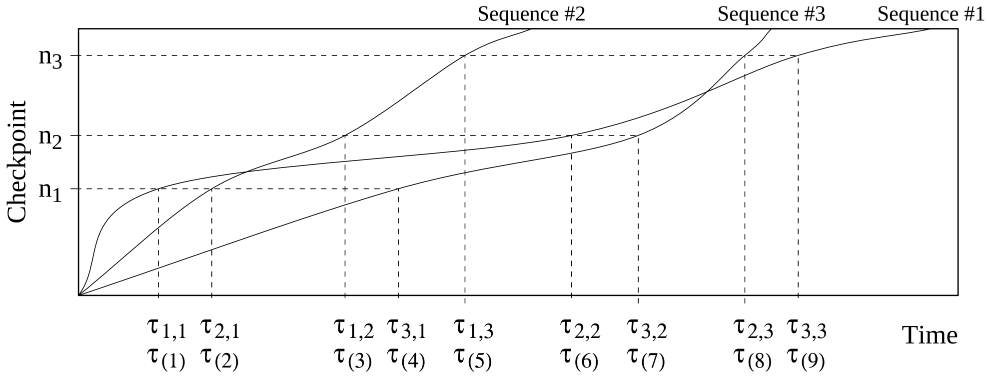


Figure 2: Example of time instances of checkpoints

2.2 Mathematical Description of Parallel Spectral Method

Suppose that we can generate multiple output sequences $\{X_{p,i}\}_{i=1}^{n(p)}$, $p = 1, \ldots, P$, in parallel. When the sequences are independently generated in different processors, the sequences are usually asynchronous in the sense that at time τ the lengths of the output sequences available may be different.

Let $\tau_{p,i}$ denote the time when the p^{th} of the P sequences reaches the checkpoint n_i. Each time a sequence reaches a checkpoint, a new set of batch means becomes available: At the time instance $\tau_{p,i}$ we obtain the batch means $\{\bar{X}_{p,j,i}\}_{j=1}^{m_i}$, $m_i = \lfloor n_i/b_i \rfloor$. Let $\{\overline{Y}_{p,j,i}\}_{j=1}^{m_i}$ denote the averages of batch means at checkpoint n_i available at time $\tau_{p,i}$:

$$\overline{Y}_{p,j,i} = \frac{1}{p} \sum_{k=1}^{p} \bar{X}_{k,j,i} \ . \tag{4}$$

The estimated relative half-width of the confidence interval for $\mu = \mathrm{E}\{X\}$ based on the sequence $\{\overline{Y}_{p,j,i}\}_{j=1}^{m_i}$ is:

$$\hat{\varepsilon}_{p,i} = t_{\nu_{p,n_i}}(1-\alpha/2)\, s_{\hat{\mu}_{p,i}} \big/ |\hat{\mu}_{p,i}| \ , \tag{5}$$

where $\hat{\mu}_{p,i} = \sum_{j=1}^{m_i} \overline{Y}_{p,j,i}/m_i$, $s^2_{\hat{\mu}_{p,i}}$ is the estimated variance of $\hat{\mu}_{p,i}$ with ν_{p,n_i} degrees of freedom, and $t_{\nu_{p,n_i}}(x)$ is the $100x^{\mathrm{th}}$ percentile of the Student t-distribution with ν_{p,n_i} degrees of freedom. The spectral method applied to sequence $\{\overline{Y}_{p,j,i}\}_{j=1}^{m_i}$ provides the estimated variance of $\hat{\mu}_{p,i}$.

```
    /* get server_key and initial seed from the coordinator */
receive_from_simulation_coordinator(&server_key, &seed);
set_seed(seed);
run_length = 0;
for (;;)    /* infinite loop terminated by coordinator */
{       /* simulate to the next checkpoint */
    generate(&run_length, &nbatches, batch_means);
        /* send checkpoint information to coordinator */
    send_to_simulation_coordinator(server_key, run_length, nbatches, batch_means);
}
```

Figure 3: Generic code executed by the servers

The run length control terminates the simulation as soon as the estimated confidence interval is narrow enough. Let the sequence $\tau_{(1)}, \tau_{(2)}, \ldots$ denote the ordered sequence of the $\tau_{p,i}$'s and let $\hat{\varepsilon}(\tau_{p,i})$ denote $\hat{\varepsilon}_{p,i}$. Figure 2 illustrates the time instances of checkpoints.

The termination time $\mathcal{T}$ is formally defined as

$$\mathcal{T} = \text{smallest } \tau \in \{\tau_{(1)}, \tau_{(2)}, \ldots\} \text{ such that } \hat{\varepsilon}(\tau) \leq \varepsilon \ . \tag{6}$$

Mathematically the termination rules (3) and (6) are asymptotically ($\varepsilon \to 0$) equivalent, when P is bounded and $n_{i+1} = n_i + 1$. In practice, the rules may have different properties when $\varepsilon > 0$.

2.3 Implementation Outline of the Parallel Method

A process group of one coordinator and P servers is a natural way to implement the parallel spectral method. The servers generate the P independent sequences of the observations and send the batch means at the prespecified checkpoints to the coordinator. The coordinator initiates and terminates the servers, maintains the averages of the batch means at different checkpoints, and estimates the mean and its relative precision.

The codes executed by the servers and the coordinator are illustrated in Figures 3 and 4, respectively. When these codes are compared to the code of uniprocessor simulation, we see that the generation of observations (function `generate`) and the estimation of relative precision (function `estimate`) are identical. The additional parts found in the parallel version can be classified into three main categories: 1) management of parallelization, 2) communication, and 3) simulation support.

The management of parallelization is implemented by the functions `start_simulation_servers` and `stop_simulation_servers` The communication between the coordinator and the servers is established by the functions `receive_from_simulation_server`, `receive_from_simulation_coordinator`, and `send_to_simulation_coordinator`. The simulation support is implemented in the function `update` which maintains the averages of the batch means, that is the $\overline{Y}_{p,j,i}$'s in Eq. (4). The simulation support also includes the centralized delivery of initial seeds that is necessary for the replications to be independent when the servers are identical.

3 Design Principles and Implementation Details

In this section we describe our software that implements the parallel spectral method. Firstly, we discuss our design principles and present the layered software structure. Secondly, we briefly describe the functionality of our RPC servers. Finally, we present how we have implemented 1) the management of parallelization, 2) the communication, and 3) the recovery.

3.1 Design Principles

When we design an application as a process group, there are several design issues that must be taken into account. Important issues include the hardware and software platforms, distribution of processes, transient communication failures, and permanent hardware or software failures.

Our hardware platform consists of diskless workstations and file server(s) connected via a local area network. The workstations we have are SPARCstation ECLs, IPCs, IPXs, and SCLs. The file server we use is a SPACserver 670MP. Our LAN is a 10 Mb Ethernet. Our software platform is SunOS 4.1.2 with Networking Facility. As the communication mechanism we use Sun RPC with the UDP protocol. [SUN 1990]

The most important design constraint is imposed by the fact that workstations are primarily intended for personal use. Therefore, we speak about *native* and *quest* processes. Our first underlying design principle is that quest processes should not disturb native processes.

```
/* input parameters:
 *  executable:      pathname of code to be executed by the servers
 *  nservers:        number of servers
 *  desired_relhw:   desired relative half-width of the confidence interval
 *  conf_level:      confidence level
 */

        /* fundamental data structures */
    struct message {
        unique_key  server_key;
        int         run_length;
        int         nbatches;
        double      batch_means[MAX_BATCHES];
    } *msg_ptr, *receive_from_simulation_server();

    struct batch_means_info {
        set_of_unique_keys   server_keys;
        int                  nreplies;
        int                  run_length;
        int                  nbatches;
        double               average_batch_means[MAX_BATCHES];
    } *bmi_ptr, *update();

    n_running = start_simulation_servers(nservers, executable);
    do
    {       /* wait for next server to reach a checkpoint
             * receive_from_simulation_server returns a pointer to the message send by a server
             */
        msg_ptr = receive_from_simulation_server(timeout);
            /* update returns a pointer to the data available at the checkpoint of last message */
        bmi_ptr = update(msg_ptr);
        est_relhw = estimate(bmi_ptr->nbatches, bmi_ptr->average_batch_means,
                             conf_level, &mean);
    } while (est_relhw > desired_relhw);
    n_stopped = stop_simulation_servers();
```

Figure 4: Generic code executed by the coordinator

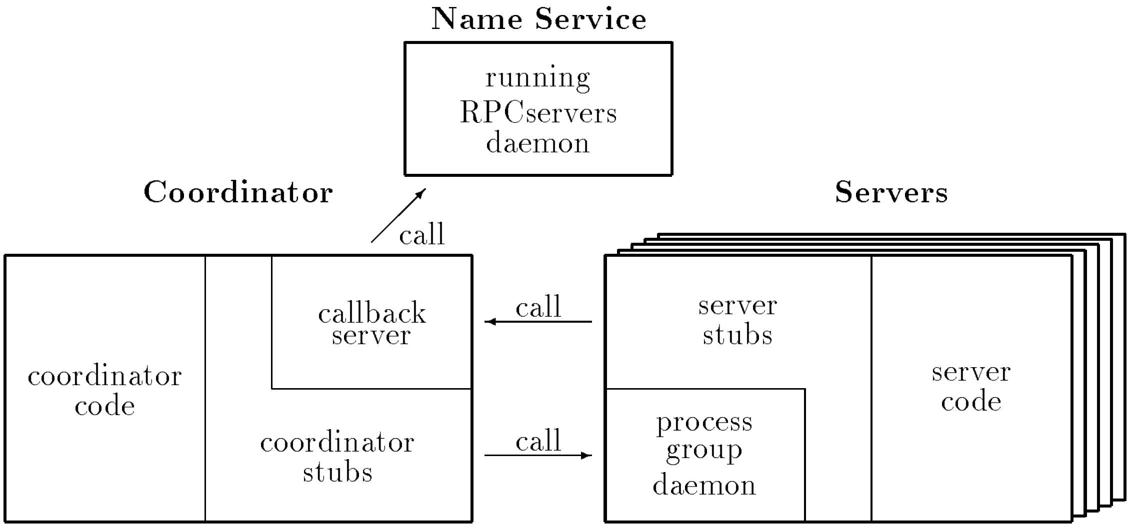


Figure 5: Software organization of the parallel spectral method

Our software is designed so that the processes group coordinator is a native process and the other processes in the group are quest processes.

Our second underlying design principle is the separation of policies from mechanisms. This principle leads to a layered software structure. The lower layers provide the basic application independent mechanisms. The upper layers implement the policies tailored for the specific needs of the application.

Our software is diveded into three layers. The bottom layer consists of RPCservers that provide the basic mechanisms for management of parallelization, communication, and recovery from persistent hardware and software failures. The middle layer consists of stubs which are the corresponding RPCclients and provide interfaces to the RPCservers. The RPCservers and RPCclients are generic in the sense that they do not depend on the application. The top level consists of functions that implement the policies suitable for parallel simulation.

Figure 5 illustrates the organization of the parallel spectral method. On the coordinator side we have the *callback server*, the *coordinator stubs*, and the *coordinator code*. On the server side we have the *process group daemon*, the *server stubs*, and the *server code*. The name service provides *running RPCservers daemon* that returns a list of workstations on which a specified RPCserver is running.

The callback server is an RPCserver that provides two RPC functions which the server stubs call to send or receive data to or from the coordinator. The coordinator stubs include functions to initiate and terminate the servers and to receive and deliver data. The coordinator code contains the code outlined in Figure 4 and the functions `start_simulation_servers`, `receive_from_simulation_server`, `update`, `estimate`, and `stop_simulation_servers`. The coordinator code also contains the function `deliver_to_simulation_server`, which is called by the RPC function when a server wants to receive data from the coordinator, and the function `simulation_servers_alive`, which is called by the coordinator to obtain the number of servers that can be reached.

The process group daemon is an RPCserver that provides six RPC functions which are called by the server and coordinator stubs. The server stubs include functions to communicate with the coordinator and to select a new coordinator in the case of a permanent coordinator failure. The server code contains the code outlined in Figure 3 and the functions `receive_from_simulation_coordinator`, `generate`, and `send_to_simulation_coordinator`. The server code also contains function `recover` which is used in the recovery.

3.2 RPC Servers

The coordinator creates and registers the callback server at the beginning of the process group application. The callback server is an RPCserver that provides two functions: `RECEIVEFROMSERVER` and `DELIVERTOSERVER`. These functions are called by the server stubs when the server wants to deliver or receive data to or from the coordinator. The callback server maintains send and receive queues where messages wait to be sent to a server or to be delivered to the coordinator. Therefore, the communication from the servers to the coordinator is asynchronous as long as buffer space allows.

The process group daemon is continuously running in each workstation that can execute servers. It is an RPCserver providing six functions: `CREATESERVER`, `KILLSERVER`, `SERVERSTATUS`, `STARTRECOVERY`, `COMPLETERECOVERY`, and `SHAREDMEMORYKEY`.

The function `CREATESERVER` is called by the coordinator to start a new server. The function `KILLSERVER` is called by the coordinator to deliver a signal to the server. Function `SERVERSTATUS` is called by the coordinator to obtain the status of the given server. Function `SHAREDMEMORYKEY` is called by the server to obtain the key for the data shared by the daemon and the caller. Functions `STARTRECOVERY` and `COMPLETERECOVERY` provide a simple method to choose one of the servers as a new coordinator.

Our software needs the service that can provide the list of workstations on which the process group daemon is running. Currently the list of workstations, on which a specific RPC program is registered, is obtained through the RPC call `GETHOSTLIST` at the host executing the running RPCservers daemon. The argument of the RPC procedure is the triple of program number, version number, and transport protocol code. The result is a list of workstation names on which the `portmapper`s know the given triple. Our current daemon polls each known workstation once in every ten minutes to obtain the list of registered RPC programs. Therefore, the list is not necessarily up-to-date.

3.3 Management of Parallelization

In the coordinator code (Figure 4) the parallelization is managed using the coordinator functions `start_simulation_servers` and `stop_simulation_servers`. These functions use the coordinator stubs `start_servers`, and `stop_servers`, respectively. The coordinator function `simulation_servers_alive`, which calls the coordinator stub `server_status`, is also used in the management of parallelization.

The coordinator stubs `start_servers` has four arguments: the number of servers to be started (`nservers`), the command line to be executed by the servers (`command_line`), size of the buffer space (`buffer_size`) for messages waiting to be delivered to a server or to be received by the coordinator, and the function (`get_delivery_data_func`) that will be called by the callback server to obtain the message to be delivered to the server. The return value of `start_servers` is the number of servers created.

The first two arguments of `start_servers` are the arguments of `start_simulation_servers`. In our implementation the buffer space is `nservers` $\times$ 16 KB (messages sent by the servers are 4108 bytes). The function `get_delivery_data_func` returns the message containing the next `server_key`, next initial seed, and the precision requirement to be sent to the server. The precision requirement is need for the recovery from permanent coordinator failures.

The coordinator stub `start_servers` creates the shared memory block containing data accessed both by the callback server and the coordinator. The stub also creates the semaphores that will be used to synchronize the coordinator and the callback server. The next step of the stub is to create a child process. The child will continue the execution of coordinator, while the parent will become the callback server.

The callback server does an RPC call to the name service to obtain a list of workstations on which the *process group daemon* is running. After that the callback server registers itself as an RPC server. When the callback server is registered, the coordinator does RPC calls of `CREATESERVER` at the workstations that are running the process group daemon. The call is repeated until `nservers` daemons have replied with a non-zero `server_id` or until each workstation on the list is called.

If the workstation is free when `CREATESERVER` is called, then the deamon allocates a new `server_id` and sends the `server_id` and `host_id` to the caller. The `server_id`'s are unique in a workstation and the `host_id`'s are unique among the workstations. If the workstation is occupied, then 0 as the `server_id` is send to the coordinator. In our implementation the daemon regards the workstation as occupied if the workstation has interactive users or if a server started by the daemon is running.

After sending the results the deamon creates a process that starts the execution of `command_line` given as an argument in the RPC call. The design decision of sending the result before the server process is created was based on performance reasons. In typical applications of process groups the coordinator starts several servers. Therefore, we decided to minimize the serialization delay in creating a new server. However, the consequence is that the server is not necessarily running even if the coordinator receives a non-zero `server_id`. We also considered an alternative way of creating the servers based on broadcasting the RPC call. However, the broadcasts created undesirable side-effects in our network.

The startup phase of process group creates a lot of network traffic when several servers load their code to be executed from the file server. The best solution to overcome the "traffic jam" we found was to include a delay parameter (`delay`) in the call of `CREATESERVER`. After the daemon creates the child using the system call `fork()`, the child executes the system call `sleep(delay)` before executing the system call `exec(command_line)`.

When a server is running, the daemon checks once in a minute that no interactive process is present. when the daemon detects that an interactive process is present, then the daemon suspends the server. When the daemon detects that no interactive process is any more present, the suspended server is continued.

When the coordinator wants to terminate the servers, it calls the coordinator function `stop_simulation_servers` which calls the coordinator stub `stop_servers`. The stub does the RPC call of `KILLSERVER` with `SIGTERM` as the value of argument `signal_no` at the workstations that replied by a non-zero `server_id` to the call of `CREATESERVER`. The coordinator function `stop_simulation_servers` returns the number of workstations that accepted the delivery of the signal.

The coordinator function `simulation_servers_alive` uses the coordinator stub `server_status` to obtain the number of servers the state of which is either `RUNNING` or `SUSPENDED`. The stub does RPC calls of `SERVERSTATUS` at the workstations that replied by a non-zero `server_id` to the call of `CREATESERVER`. The stub returns a list of triples (`host_id`, `server_id`, *reply*).

3.4 Communication

In the coordinator code the function `receive_from_simulation_server` is used to obtain the messages from the servers. The argument `timeout` specifies how long the coordinator waits for a message. The function uses the the coordinator stub `receive_from_server` to copy the first message from the receive queue maintained by the callback server in the shared memory.

The coordinator function `receive_from_simulation_server` returns a pointer to the message or `NULL` if the timeout expires. When the `NULL` is returned, then the coordinator function `simulation_servers_alive` is called to obtain the number of servers the state of which is either `RUNNING` or `SUSPENDED`. If the number of servers is zero, the the coordinator reports the current estimates and terminates. Otherwise the timeout is doubled. In our implementation the initial timeout is 15 seconds.

In the server code the communication functions are `receive_from_simulation_coordinator` and `send_to_simulation_coordinator`. The function `receive_from_simulation_coordinator` is used to receive the server key, the initial seed, and the precision requirement. The precision requirement is needed in the case of permanent coordinator failure, when one of the servers is selected to become a new coordinator. When a server reaches a checkpoint, it calls the function `send_to_simulation_coordinator` to send the data to the coordinator.

The server function `receive_from_simulation_coordinator` uses the server stub `receive_from_coordinator`. The stub returns `COMMUNICATION_SUCCESS`, `COMMUNICATION_ERROR`, or `APPLICATION_ERROR`. `COMMUNICATION_ERROR` indicates an RPC failure while `APPLICATION_ERROR` indicates that the RPC was successful but the callback server was unable to receive the data.

The server stub `receive_from_coordinator` calls the RPC function `DELIVERTOSERVER` in the callback server at the coordinator host. The callback server can obtain the data to be delivered to the servers in two different ways. In the first way the coordinator calls the coordinator stub `send_to_server` which puts the message into the `send_queue`. The second way is used, when the callback server obtains an RPC call of `DELIVERTOSERVER` with the argument `message_id` that is not found in the `send_queue`. In this case the callback server calls the function that was given as the argument `get_delivery_data_func` when the coordinator stub `start_servers` was called.

If the server stub `receive_from_coordinator` does not return `COMMUNICATION_SUCCESS`, then the server function `receive_from_simulation_coordinator` implements the following recovery policy. If the return value is `APPLICATION_ERROR`, then the server is terminated, since the callback server could not receive the data from the coordinator. If the return value is `COMMUNICATION_ERROR`, then the action depends on the RPC failure. If the RPC failure is `RPC_TIMEOUT`, then an immediate recall of `receive_from_coordinator` is done, otherwise the server is terminated. If the call of `receive_from_coordinator` leads to three `RPC_TIMEOUT` failures, then the server is terminated. In our implementation the timeout value in all RPC calls is ten seconds.

The server function `send_to_simulation_coordinator` uses the server stub `send_to_coordinator`. The server stub returns `COMMUNICATION_SUCCESS`, `COMMUNICATION_ERROR`, or `APPLICATION_ERROR`. In the server function `send_to_simulation_coordinator` the following recovery policy is implemented. If the return value of `send_to_coordinator` is `APPLICATION_ERROR`, then the buffer space in the callback server is full. The call of `send_to_coordinator` is repeated after a delay. The duration of the delay is k seconds before the k^{th} retry. The maximum number of retries is six. After the sixth unsuccessful retry the recovery from the permanent coordinator failure is started.

If the return value of `send_to_coordinator` is `COMMUNICATION_ERROR`, then the action depends on the RPC failure. If the RPC failure is `RPC_TIMEOUT`, then an immediate recall of `send_to_coordinator` is done. If the call of `send_to_coordinator` leads to three `RPC_TIMEOUT` failures, then the recovery from the permanent coordinator failure is started. If the RPC failure is `RPC_PROGUNAVAIL`, then the callback server is not running and the action is to start the recovery immediately. Other RPC failures are handled as `APPLICATION_ERROR`. During the recovery the calls of server stubs `send_to_coordinator` and `receive_from_coordinator` will suspend the execution of the server.

3.5 Recovery from Permanent Coordinator Failure

In the server code the permanent coordinator failure is detected in the server function `send_to_simulation_coordinator`. When the policy decision is to start the recovery, then the server function `recover` is called. The function uses the server stubs `collect_votes`, `wait_for_new_coordinator`, and `create_new_coordinator`.

From the server's point of view there are three possible outcomes of the recovery: 1) the recovery is unsuccessful, 2) the recovery is successful and another server is the new coordinator, and 3) the recovery is successful and the server is the new coordinator. In the first two cases the function `recover` returns. Figure 6 outlines the code of the function.

When the recovery fails, then the server is terminated. Before the termination the relative half-width based on the local batch means is estimated and an email containing the results is send to the user. When the recovery is successful and the server is not the new coordinator, then the server function `send_to_simulation_coordinator` sends the data to the new coordinator.

The server stub `collect_votes` returns the number of servers that accepted the server as the new coordinator. If the state of the server is `SERVER_RECOVERING`, the stub returns immediately 0. This indicates that another server has called the RPC function `STARTRECOVERY`. Otherwise the stub takes the timestamp (local time concatenated with the `host_id`) and changes the state to `SERVER_RECOVERING`. The next step is to obtain the list of workstations running the process group daemon. When the list is obtained from the name service, the stub does RPC calls of `STARTRECOVERY` at the workstations in the list. The call is repeated

```
enum {RECOVERY_ERROR,RECOVERY_SUCCESS}
recover( parameters )
{   ...
    n_servers = collect_votes();
    if (n_servers < 0)
        return RECOVERY_ERROR;
    if (n_servers == 0)
        if (wait_for_new_coordinator(timeout))
            return RECOVERY_ERROR;
        else
            return RECOVERY_SUCCESS;

    if (create_new_coordinator( parameters ) > 0)
    {       /* code of new coordinator */
    }

    return RECOVERY_ERROR;
}
```

Figure 6: Outline of code for the function `recover`

until each workstation is called or until the result is `TERMINATE_RECOVERY` indicating that the callee has received an older timestamp. In this case the function returns 0, otherwise the function returns the number of workstations that replied `CONTINUE_RECOVERY`.

The server stub `create_new_coordinator` creates a new callback server. The function is quite similar to the coordinator stub `start_servers`. The most notable differences are that the RPC function `COMPLETERECOVERY` is called instead of `CREATESERVER` and that the list of workstations to be called is created by the function `collect_votes`. The return value of `create_new_coordinator` is the number of daemons that replied to `COMPLETERECOVERY`.

The server stub `create_new_coordinator` creates the shared memory block containing data accessed both by the callback server and the coordinator, and the semaphores that will be used in the synchronization. The next step of the function is to create a child process. The child will continue the execution of `create_new_coordinator`, while the parent will become the callback server. When the callback server is registered, the execution of `create_new_coordinator` is continued. The function does RPC calls of `COMPLETERECOVERY` at the workstations in the list created during the execution of `collect_votes`. Finally, the function change the server state to `SERVER_IS_NOW_COORDINATOR`. This implies that the server is now a native process.

When the process group daemon receives the call of `COMPLETERECOVERY`, then it updates the control information. The execution of servers which is blocked due to the call of `send_to_coordinator` or `receive_from_coordinator` during the recovery or due to the call of `wait_for_new_coordinator` is continued.

4 Experimental Results

We examined the properties of the parallel spectral method using 12 different simulation models and 6 different target precisions. As stated in the Introduction our objective was to get answers to the following two questions: 1) Does the parallel spectral method provide estimates that meet the target precision prespecified by the user? 2) Is it possible, in practice, to obtain remarkable speedup through parallel generation of observations?

4.1 Description of Experiments

The simulation models examined are taken from the survey of sequential procedures for steady-state simulation [Law and Kelton 1982] and from the paper introducing the spectral method [Heidelberger and Welch 1981]. Models 1–9 are exactly the same as in Law and Kelton [1982]. Model 10 is a slight modification of Model 10 in Law and Kelton [1982]. Models 11 and 12 are the queueing network models used in Heidelberger and Welch [1981]. The simulations were initialized using the known (Models 1–3, 5–8, 10–12) or approximated (Model 4) steady-state distribution for the number of customers at different service centers. Model 9 was initialized as in Law and Kelton [1982].

It should be noted that these models were initially chosen as the test models for serial simulation. Therefore, they may be inappropriate when the speedup in parallel simulation is examined. However, the models are appropriate when the obtained precision is examined since they are widely used in many independent simulation studies. Appendix A gives detailed descriptions of the models.

In the experiments we examined six pairs of (ε, α) specifying the target precision: $\varepsilon =$ 0.20, 0.10, 0.05 and $\alpha = 0.10$, 0.05. The number of server processes (P) examined was 5, 10, 15, and 20. As the baseline reference we simulated the models with the serial spectral method ($P = 1$). The simulation was independently repeated 100 times in each experimental point (model, P, ε, α). The experiments were executed during two weekends in summer time when other workload in the network and workstations was very low.

The spectral method used to estimate the variance of the means has the following parameters: 1) minimum number of batches (M), 2) degree of polynomial fitted to the logarithms of smoothed periodogram ordinates (d), 3) number of points to which the polynomial is fitted (K; $K < M/4$), 4) first checkpoint (n_1), and 5) multiplier for the next checkpoint (I; $n_{i+1} = I\,n_i$). In our implementation we have used the following values: $M = 512$, $d = 2$, $K = 31$, $n_1 = 512$, and $I = 2$. These values imply that the degrees of freedom in the variance estimator ($\nu(n)$) is 9.

4.2 Observed Coverages

The question whether the estimates meet the prespecified target precision is examined through analyzing the observed coverages, which are shown in Table 1. The observed coverage is the fraction of experiments in which the estimated confidence interval, $[\hat{\mu} - s_{\hat{\mu}} t_v(1 - \alpha/2), \hat{\mu} + s_{\hat{\mu}} t_v(1 - \alpha/2)]$, includes μ.

Since the experiments were independent, the observed coverage ($\hat{\pi}$) has binomial distribution with parameters n and π where n is the number of experiments and π is the true unknown coverage. When the requested confidence level is $1 - \alpha$, we can construct the following pair of hypotheses to test whether the true coverage is acceptable:

$$
\begin{aligned}
H_0 &: \quad \pi \geq 1 - \alpha \quad \text{(true coverage is acceptable)} \\
H_A &: \quad \pi < 1 - \alpha
\end{aligned}
$$

In Table 1 we have highlighted the observed coverages that lead to the rejection of the null-hypothesis at significance levels 0.10, 0.05, and 0.01. The overall impression is that quite many observed coverages lead to the rejection of the null-hypothesis that the true coverage is acceptable.

In order to examine the differences between the coverages in serial ($P = 1$) and parallel ($P = 5, 10, 15, 20$) spectral methods, we examined the following pair of hypotheses:

$$
\begin{aligned}
H_0 &: \quad \pi_{\text{serial}} = \pi_{\text{parallel}} \\
H_A &: \quad \pi_{\text{serial}} > \pi_{\text{parallel}}
\end{aligned}
$$

Table 1: Observed coverages in 100 independent replications

α	ε	P	Model											
			1	2	3	4	5	6	7	8	9	10	11	12
0.10	0.20	1	.89	**.76**	.87	.87	.94	.96	<u>*.84*</u>	.98	.93	*.85*	.86	<u>*.84*</u>
		5	**.77**	**.80**	**.69**	**.80**	.93	.96	<u>*.82*</u>	.98	.95	**.71**	.90	<u>*.82*</u>
		10	**.80**	**.74**	**.65**	**.69**	.90	.97	<u>*.82*</u>	.97	.92	**.78**	<u>*.82*</u>	*.85*
		15	**.74**	**.69**	**.66**	**.76**	.96	.94	.86	.98	.90	**.68**	.90	**.78**
		20	**.77**	**.69**	**.69**	**.76**	.86	.99	.87	.98	.91	**.74**	.88	*.85*
	0.10	1	.93	.90	*.85*	<u>*.84*</u>	.94	.94	**.80**	.97	.93	.86	.90	.93
		5	.90	<u>*.83*</u>	**.80**	**.79**	.93	.95	<u>*.83*</u>	.96	.95	**.80**	.89	*.85*
		10	*.85*	.87	**.77**	**.78**	.90	.97	**.75**	.94	.92	<u>*.84*</u>	*.85*	**.76**
		15	<u>*.82*</u>	**.78**	**.72**	**.77**	.96	.91	<u>*.83*</u>	.98	.90	<u>*.83*</u>	.91	**.79**
		20	<u>*.84*</u>	<u>*.82*</u>	**.79**	**.71**	.86	.98	**.80**	.95	.91	<u>*.84*</u>	.93	**.81**
	0.05	1	.91	.90	.90	**.78**	.96	.88	.90	.92	.93	.95	**.77**	.96
		5	.91	**.79**	.86	**.76**	.86	.90	*.85*	.91	.95	<u>*.84*</u>	**.80**	<u>*.84*</u>
		10	<u>*.82*</u>	**.77**	.88	**.72**	.90	.92	**.81**	.90	.92	*.85*	<u>*.83*</u>	.89
		15	.88	*.85*	.89	**.73**	.89	.88	**.80**	.96	.90	.90	<u>*.82*</u>	.88
		20	.91	.87	.89	**.69**	.87	.94	**.81**	.94	.91	.88	.90	.92
0.05	0.20	1	.97	<u>*.90*</u>	.93	.94	.98	.98	*.91*	1.0	.96	*.91*	<u>*.90*</u>	<u>*.90*</u>
		5	**.86**	**.83**	**.82**	**.86**	.96	1.0	.92	.99	.96	**.83**	.95	**.88**
		10	**.87**	**.82**	**.74**	**.82**	.94	.97	*.91*	.99	.94	**.79**	.92	<u>*.90*</u>
		15	**.85**	**.87**	**.75**	**.88**	.98	.98	.94	.98	.93	**.80**	.92	<u>*.89*</u>
		20	**.84**	**.79**	**.77**	**.88**	.96	1.0	.92	.99	.95	<u>*.90*</u>	.96	.93
	0.10	1	.96	.95	.94	<u>*.89*</u>	.97	.94	<u>*.89*</u>	.99	.96	.97	.94	.98
		5	*.91*	.93	.93	<u>*.90*</u>	.98	1.0	<u>*.89*</u>	.97	.96	**.87**	.92	<u>*.90*</u>
		10	*.91*	<u>*.89*</u>	.92	**.84**	.94	.95	**.86**	.99	.94	<u>*.89*</u>	.93	<u>*.90*</u>
		15	.93	**.88**	**.81**	**.81**	.98	.95	<u>*.90*</u>	.97	.93	<u>*.89*</u>	.96	**.88**
		20	.95	**.87**	**.86**	**.82**	.95	.99	<u>*.89*</u>	.98	.95	.95	.96	*.91*
	0.05	1	.97	.94	.94	**.78**	.98	.96	.95	.94	.96	.97	**.86**	.98
		5	.94	<u>*.90*</u>	.96	**.81**	.96	.97	.93	.94	.96	**.88**	**.85**	<u>*.89*</u>
		10	<u>*.89*</u>	**.88**	*.91*	**.82**	.94	.93	**.88**	.95	.94	.95	**.86**	.95
		15	<u>*.90*</u>	*.91*	*.91*	**.83**	.93	.96	<u>*.90*</u>	.98	.93	.93	**.86**	.95
		20	.97	*.91*	.92	**.75**	.92	.97	<u>*.89*</u>	.98	.95	.94	.94	.98

bold: hypothesis of *true coverage* $\geq 1 - \alpha$ rejected at significance level 1%
<u>*italics:*</u> hypothesis of *true coverage* $\geq 1 - \alpha$ rejected at significance level 5%
italics: hypothesis of *true coverage* $\geq 1 - \alpha$ rejected at significance level 10%

The hypotheses above can statistically be tested applying the comparison of two proportions given e.g. in Hald [1952, Ch. 21.13]. Table 2 gives the classification of models based on the significance of the test statistics Z:

$$Z = \sqrt{\frac{n}{2}} \left((\hat{y}_{\text{serial}} - \hat{y}_{\text{parallel}}) + \frac{\hat{\pi}_{\text{serial}} - \hat{\pi}_{\text{parallel}}}{\sqrt{\hat{\pi}_{\text{common}}(1 - \hat{\pi}_{\text{common}})}} \right) ,$$

where n is the number of replications, $\hat{y}_{\text{x}} = 2\sin^{-1}(\sqrt{\hat{\pi}_{\text{x}}})$ and $\hat{\pi}_{\text{common}} = (\hat{\pi}_{\text{serial}} + \hat{\pi}_{\text{parallel}})/2$.

Table 2 indicates that the serial spectral method provides better coverage than the parallel spectral method. This is especially true for $\varepsilon = 0.20$. Therefore, our overall conclusion is that the parallel spectral method should only be used if the number of server processes is low or if the target precision is stringent. When the number of server processes is high and the target precision is not stringent, it is possible that observed coverage is low. In the light of results reported for the method of independent replications this is not surprising. If the number of server processes is not small and the target precision does not force the simulation to be continued until the Student t-approximation applied in the spectral method is accurate enough, the constructed confidence intervals will usually have a low coverage.

4.3 Execution Times

The practical speedups obtained by the parallel generation of observations were examined through measuring the execution times. In the serial method (Figure 1) the measurement points were before the line '`get_initial_seed(&seed);`' and after the line '`} while (est_relhw > desired_relhw);`'. In the parallel method (Figure 4) the measurement points were before the line '`n_running = ...`' and after the line '`n_stopped = ...`'.

The averages of the execution times are given in Table 3. We can find three groups of models such that the averages in the group have similar patterns for the triples (P, ε, α). The groups are: 1) Models 1, 2, 3, 10; 2) Models 4, 11, 12; and 3) Models 5, 6, 8. Figure 7 shows the average execution times of one model in each group. Model 7 is between the groups 2 and 3. Model 9 is exceptional. It is stochastically so simple that the simulation is almost always terminated at the first checkpoint.

The first impression is that the parallel spectral method does not gain any practical speedup ($S = T_{\text{parallel}}/T_{\text{serial}}$). However, this is primarily due to the high management costs

Table 2: Classification of models based on the significance in the statistical test of $H_0 : \pi_{\text{serial}} = \pi_{\text{parallel}}$ vs. $H_A : \pi_{\text{serial}} > \pi_{\text{parallel}}$

			Significance				
α	ε	P	0–.01	.01–.05	.05–.10	.10–.20	.20–1
0.10	0.20	5	3	1 10		4	2 5 6 7 8 9 11 12
		10	3 4		1	10	2 5 6 7 8 9 11 12
		15	1 3 10	4		2 12	5 6 7 8 9 11
		20	3	1 4 5 10		2	6 7 8 9 11 12
	0.10	5			12	2 10	1 3 4 5 6 7 8 9 11
		10	12		1	3 4 11	2 5 6 7 8 9 10
		15	12	1 2 3		4	5 6 7 8 9 10 11
		20	12	1 4 5	2	3	6 7 8 9 10 11
	0.05	5	10 12	2 5		7	1 3 4 6 8 9 11
		10		1 2 10	5 7 12		3 4 6 8 9 11
		15		7 12	5	2 10	1 3 4 6 8 9 11
		20		5	4 7 10	12	1 2 3 6 8 9 11
0.05	0.20	5	1	3 4	10	2	5 6 7 8 9 11 12
		10	1 3 4	10	2	5	6 7 8 9 11 12
		15	1 3	10		4	2 5 6 7 8 9 11 12
		20	1 3	2		4	5 6 7 8 9 10 11 12
	0.10	5	10	12		1	2 3 4 5 6 7 8 9 11
		10		10 12	2	1	3 4 5 6 7 8 9 11
		15	3 12	10	2 4		1 5 6 7 8 9 11
		20		2 3 12		4	1 5 6 7 8 9 10 11
	0.05	5	12	10			1 2 3 4 5 6 7 8 9 11
		10		1	7	2 5	3 4 6 8 9 10 11 12
		15		1	5	7 10	2 3 4 6 8 9 11 12
		20		5	7		1 2 3 4 6 8 9 10 11 12

Table 3: Average Execution Times (seconds)

			Model											
ε	α	P	1	2	3	4	5	6	7	8	9	10	11	12
0.20	0.10	1	1.34	1.27	1.63	.768	.114	.115	.154	.115	.113	1.26	.950	.509
		5	1.89	1.88	1.78	1.43	1.23	1.22	1.20	1.24	1.21	1.72	1.71	1.52
		10	2.72	2.81	2.53	2.30	2.25	2.25	2.20	2.25	2.20	2.65	2.53	2.42
		15	3.56	3.45	3.33	3.14	2.97	3.00	2.96	2.98	2.94	3.39	3.30	3.15
		20	5.79	4.48	4.33	4.14	4.42	4.36	4.38	4.44	4.32	4.46	4.40	4.32
	0.05	1	2.22	1.95	2.27	1.12	.114	.115	.197	.117	.113	1.86	1.60	.708
		5	2.11	2.12	2.03	1.59	1.23	1.22	1.22	1.24	1.21	1.95	1.91	1.61
		10	2.96	3.00	2.78	2.41	2.25	2.25	2.22	2.25	2.20	2.88	2.67	2.49
		15	3.82	3.79	3.53	3.27	2.97	3.00	2.98	2.98	2.94	3.63	3.40	3.23
		20	5.92	4.80	4.59	4.26	4.42	4.36	4.41	4.44	4.32	4.73	4.48	4.39
0.10	0.10	1	5.33	5.20	5.44	2.59	.122	.155	.427	.164	.113	4.70	4.11	2.30
		5	2.89	2.92	2.89	1.99	1.23	1.24	1.37	1.26	1.21	2.74	2.55	2.14
		10	3.49	3.54	3.41	2.87	2.26	2.27	2.31	2.28	2.20	3.42	3.04	2.91
		15	4.43	4.38	4.20	3.60	2.97	3.02	3.11	3.01	2.94	4.26	3.73	3.63
		20	6.67	5.48	5.30	4.69	4.42	4.38	4.56	4.47	4.32	5.44	4.86	4.78
	0.05	1	7.94	8.03	8.47	3.92	.160	.207	.628	.221	.113	7.63	6.19	3.29
		5	3.37	3.48	3.60	2.29	1.25	1.26	1.47	1.29	1.21	3.19	3.01	2.55
		10	3.73	3.77	3.68	3.10	2.28	2.30	2.41	2.30	2.20	3.71	3.33	3.25
		15	4.71	4.64	4.46	3.82	2.99	3.04	3.23	3.03	2.94	4.51	3.98	3.89
		20	6.97	5.80	5.64	4.93	4.43	4.41	4.66	4.51	4.32	5.75	5.06	5.08
0.05	0.10	1	21.7	21.3	21.7	8.75	.340	.380	1.53	.427	.113	20.0	14.9	8.21
		5	6.13	6.42	6.21	3.26	1.34	1.35	1.92	1.38	1.21	5.34	5.20	3.87
		10	4.77	4.97	4.84	3.57	2.35	2.38	2.77	2.40	2.20	4.64	4.47	4.25
		15	5.25	5.27	5.01	4.39	3.07	3.13	3.59	3.16	2.94	5.08	4.82	4.73
		20	7.67	6.42	6.28	5.56	4.53	4.52	5.07	4.62	4.32	6.36	5.85	5.94
	0.05	1	32.9	34.0	35.1	13.3	.473	.518	2.21	.536	.113	31.6	21.2	11.8
		5	8.25	8.57	8.15	3.85	1.41	1.43	2.14	1.46	1.21	7.43	6.56	4.84
		10	5.64	5.92	5.64	3.89	2.42	2.43	3.00	2.47	2.20	5.58	5.25	4.76
		15	5.70	5.77	5.47	4.62	3.13	3.19	3.83	3.24	2.94	5.52	5.41	5.15
		20	8.01	6.83	6.60	5.81	4.60	4.60	5.29	4.72	4.32	6.63	6.30	6.32

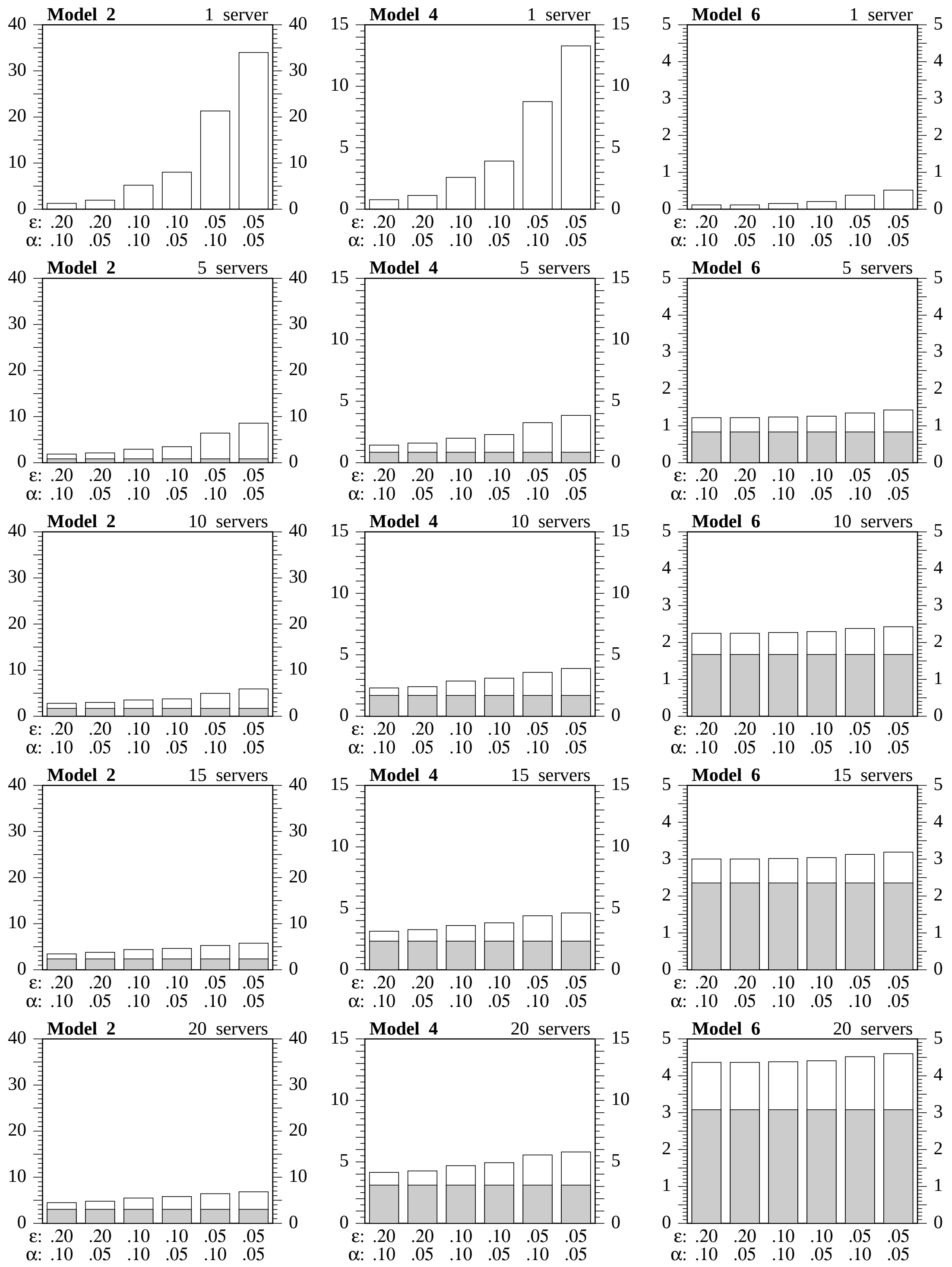


Figure 7: Average Execution Times (seconds)
Shaded are is the time spent in the management functions

Table 4: Speedups and Efficiencies in Model 2 when Management Overhead Is Eliminated

		Speedup				Efficiency			
		Number of Servers				Number of Servers			
ε	α	5	10	15	20	5	10	15	20
0.20	0.10	1.23	1.15	.87	.88	.246	.115	.058	.044
	0.05	1.54	1.50	1.36	1.10	.307	.150	.070	.055
0.10	0.10	2.51	2.84	2.57	2.13	.502	.284	.171	.106
	0.05	3.05	3.89	3.52	2.90	.611	.389	.235	.145
0.05	0.10	3.83	6.53	7.32	6.28	.765	.653	.488	.315
	0.05	4.40	8.06	9.96	8.94	.880	.806	.664	.447

of creating and terminating the servers. In Figure 7 the shaded areas represent the time spent in the management functions. In order to examine the potential speedup and efficiency ($E = pT_{\text{parallel}}/T_{\text{serial}}$) we give in Table 4 the speedups and efficiencies for Model 2 when the management overhead due to start and stop the servers is eliminated. The speedups in Table 4 indicate that the parallel spectral method is an attractive way of using the parallelism when the generation of observations is the dominant time component.

5 Summary

We have introduced a parallel spectral method and described one possible implementation for a network of diskless workstations. The results reported from simulation experiments indicate that the method should only be used when the number of server processes is moderate, say from 5 to 10, and when the precision requirement is stringent. If the number of servers is high, we have 1) a high possibility to obtain estimates that do not satisfy the target precision, and 2) a high possibility to waste most (or even all) of the speedup in parallel generation to management and communication overhead.

References

BAIK, D. AND ZEIGLER, B. P. 1985. Performance Evaluation of Hierarchical Distributed Simulators. In *Proceedings of the 1985 Winter Simulation Conference*. IEEE, 421–427.

CHANDY, K. M. AND MISRA, J. 1981. Asynchronous Distributed Simulation Via a Sequence of Parallel Simulations. *Commun. ACM 24*, 4 (Apr.), 198–206.

COMFORT, J. C. 1984. The Simulation of a Master-Slave Event Set Processor. *Simulation 42*, 3 (Mar.), 117–124.

DUDA, A. 1989. On the Tradeoff Between Parallelism and Communication. In *Modeling Techniques and Tools for Computer Performance Evaluation*, R. Puigjaner and D. Potier, Editors. Plenum Press, 323–334.

FUJIMOTO, R. M. 1988. Performance Measurements of Distributed Simulation Strategies. In *Proceedings of the SCS Multiconference on Distributed Simulation, 1988*. SCS, 14–20.

HALD, A. 1952. *Statistical Theory with Engineering Applications*. John Wiley & Sons, New York.

HEIDELBERGER, P. AND WELCH, P. D. 1981. A Spectral Method for Confidence Interval Generation and Run Length Control in Simulations. *Communications of the ACM 24*, 4 (Apr.), 233–245.

LAW, A. M. AND KELTON, W. D. 1982. Confidence Intervals for Steady-State Simulations, II: A Survey of Sequential Procedures. *Management Science 28*, 5 (May), 550–562.

LIN, Y.-B. 1994. Parallel Independent Replicated Simulation on a Network of Workstations. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation (PADS'94)*. ACM SIGSIM, 73–80.

NICOL, D. M. 1988a. Parallel Discrete Event Simulation of FCFS Stochastic Queueing Networks. In *Parallel Programming: Experience with Applications, Languages and Systems*. ACM SIGPLAN, 124–137.

NICOL, D. M. 1988b. High Performance Parallelized Discrete-Event Simulation of Stochastic Queueing Networks. In *Proceedings of the 1988 Winter Simulation Conference*. IEEE/ACM/SCS, 306–314.

PAWLIKOWSKI, K. 1990. Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions. *ACM Computing Surveys 22*, 2 (June), 123–170.

REED, D. A. 1985. Parallel Discrete Event Simulation: A Case Study. In *Proceedings of the 18th Annual Simulation Symposium*. IEEE Computer Society, 95–107.

REED, D. A. AND MALONY, A. D. 1988. Parallel Discrete Event Simulation: The Chandy-Misra Approach. In *Proceedings of the SCS Multiconference on Distributed Simulation, 1988*. SCS, 8–13.

REED, D. A., MALONY, A. D., AND MCCREDIE, B. D. 1988. Parallel Discrete Event Simulation Using Shared Memory. *IEEE Trans. Soft. Eng. SE-14*, 4 (Apr.), 541–553.

REGO, V. J. AND SUNDERAM, V. S. 1992. Experiments in concurrent stochastic simulation: the EcliPSe paradigm. *Journal of Parallel and Distributed Computing 14*, 1 (Jan.), 66–84.

Reynolds, P. F. and Kuhn, C. S. 1987. A Performance Study of Three Protocols for Distributed Simulation. In *Proceedings of the Conference on Methodology and Validation, 1987*. SCS, 26–31.

Schruben, L. W. 1987. Using Simulation to Solve Problems: A Tutorial on the Analysis of Simulation Output. In *Proceedings of the 1987 Winter Simulation Conference*. ACM/IEEE/SCS, New York, 40–42.

Sun Microsystems, Inc. 1990. *Network Programming Guide*, revision a of 27 march, 1990. part number: 800-3850-10 edition.

Wagner, D. B. and Lazowska, E. D. 1989. Parallel Simulation of Queueing Networks: Limitations and Potentials. In *Proceedings of the 1989 ACM SIGMETRICS and Performance '89*. ACM SIGMETRICS, 146–155.

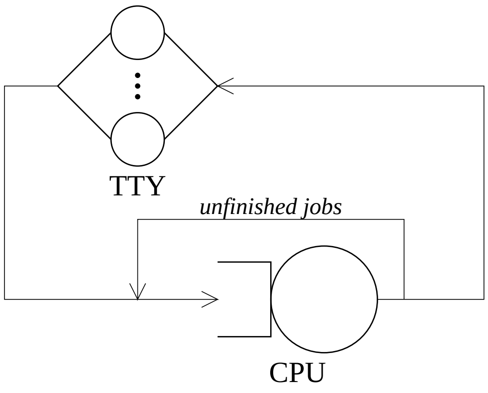
Appendix A Description of Models Examined

Model 1: Mean waiting time in M/M/1-queue: $\lambda = 1$, $\mu = 1.25$, FCFS

Model 2: Mean waiting time in M/M/1-queue: $\lambda = 1$, $\mu = 1.25$, LCFS

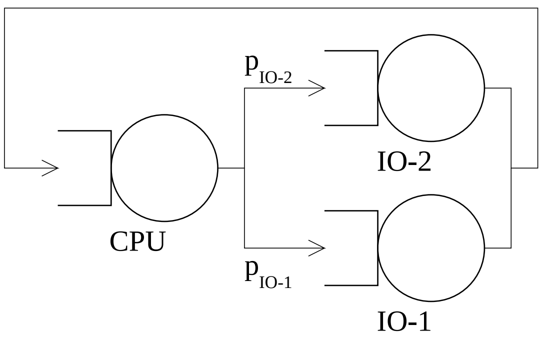
Model 3: Mean waiting time in M/M/1/M/1-tandem queue: $\lambda = 1$, $\mu_1 = \mu_2 = 1.25$, FCFS

Model 4: Mean turnaround time in Time-Shared Computer Model:



Population 35
$\mu_{\mathrm{TTY}} = 0.04$
$\mu_{\mathrm{CPU}} = 1.25$
round robin scheduling at CPU
– quantum 0.1
– switch overhead 0.015

Model 5–8: Mean round trip time in Central-Server Computer Model:



Model	5	6	7	8
Population	4	8	8	8
μ_{CPU}	1	1	1	1
$\mu_{\mathrm{IO\text{-}1}}$	0.5	0.5	0.45	1.8
$\mu_{\mathrm{IO\text{-}2}}$	0.5	0.5	0.05	0.2
$p_{\mathrm{IO\text{-}1}}$	0.5	0.5	0.9	0.9
$p_{\mathrm{IO\text{-}2}}$	0.5	0.5	0.1	0.1

Model 9: Expected total cost during one period in (s,S) Inventory System:

Let X_i be the amount of inventory on hand before ordering, Y_i the amount of inventory on hand after ordering, and Q_i the demand, each in period i.

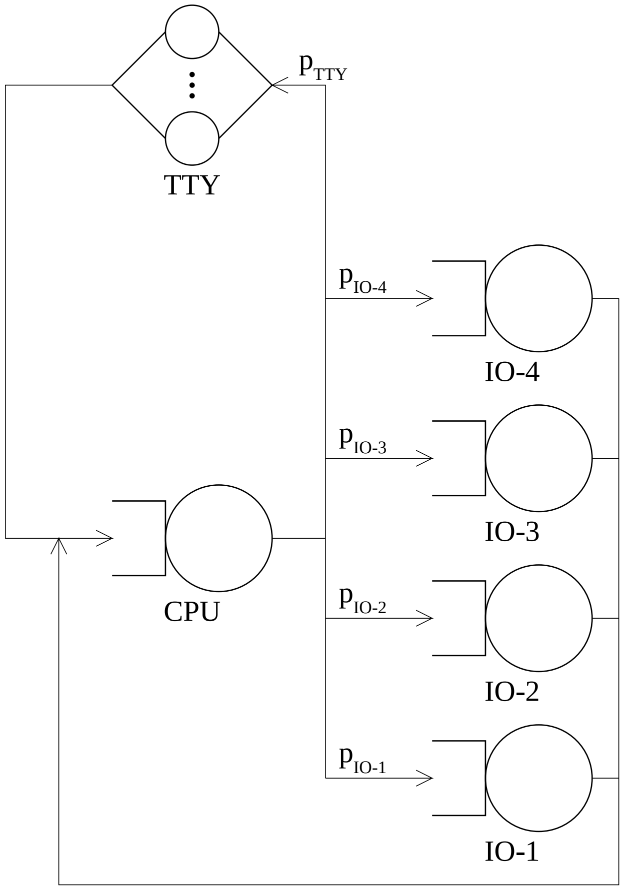
If $X_i < s$, then $S - X_i$ items are ordered ($Y_i = S$) and an ordering cost $K + c(S - X_i)$ is incurred. If $X_i \le S$, then no order is placed ($Y_i = X_i$) and no ordering cost is incurred.

After Y_i has been determined, then a demand Q_i occurs. If $Y_i \geq Q_i$, then a holding cost $h(Y_i - Q_i)$ is incurred. If $Q_i > Y_i$, then a shortage cost $\pi(Q_i - Y_i)$ is incurred. In either case, $X_{i+1} = Y_i - Q_i$.

Parameters: $s = 17$, $S = 57$, $K = 32$, $c = 3$, $h = 1$, $\pi = 5$, $X_1 = S$. The demands (Q_i's) are independent Poisson random variables with mean 25.

Model 10: Mean queue length at arrival instance in M/M/1-queue: $\lambda = 1$, $\mu = 1.25$, FCFS

Model 11–12: Mean response time in Central-Server Computer Model:



Model	11	12
Population	25	25
μ_{TTY}	0.01	0.01
μ_{CPU}	1.0	1.0
$\mu_{\mathrm{IO\text{-}1}}$, $\mu_{\mathrm{IO\text{-}2}}$	0.72	0.18
$\mu_{\mathrm{IO\text{-}3}}$, $\mu_{\mathrm{IO\text{-}4}}$	0.08	0.04
$p_{\mathrm{IO\text{-}1}}$, $p_{\mathrm{IO\text{-}2}}$	0.36	0.36
$p_{\mathrm{IO\text{-}3}}$, $p_{\mathrm{IO\text{-}4}}$	0.04	0.04
p_{TTY}	0.2	0.2