Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 820803, 18 pages
http://dx.doi.org/10.1155/2015/820803

*Research Article*

# Teaching Scientific Computing: A Model-Centered Approach to Pipeline and Parallel Programming with C

**Vladimiras Dolgopolovas,[1,2] Valentina Dagienė,[1,3] Saulius Minkevičius,[4] and Leonidas Sakalauskas[4]**

[1] *Informatics Methodology Department, Institute of Mathematics and Informatics, Vilnius University, Akademijos Street 4, LT-08663 Vilnius, Lithuania*
[2] *Department of Software Development, Faculty of Electronics and Informatics, Vilniaus Kolegija University of Applied Sciences, J. Jasinskio Street 15, LT-01111 Vilnius, Lithuania*
[3] *Department of Didactics of Mathematics and Informatics, Faculty of Mathematics and Informatics, Vilniaus University, Naugarduko Street 24, LT-03225 Vilnius, Lithuania*
[4] *Operational Research Sector at System Analysis Department, Institute of Mathematics and Informatics, Vilniaus University, Akademijos Street 4, LT-08663 Vilnius, Lithuania*

Correspondence should be addressed to Vladimiras Dolgopolovas; vdolgopolovas@gmail.com

Received 25 July 2014; Revised 24 February 2015; Accepted 7 March 2015

Academic Editor: Beniamino Di Martino

The aim of this study is to present an approach to the introduction into pipeline and parallel computing, using a model of the multiphase queueing system. Pipeline computing, including software pipelines, is among the key concepts in modern computing and electronics engineering. The modern computer science and engineering education requires a comprehensive curriculum, so the introduction to pipeline and parallel computing is the essential topic to be included in the curriculum. At the same time, the topic is among the most motivating tasks due to the comprehensive multidisciplinary and technical requirements. To enhance the educational process, the paper proposes a novel model-centered framework and develops the relevant learning objects. It allows implementing an educational platform of constructivist learning process, thus enabling learners' experimentation with the provided programming models, obtaining learners' competences of the modern scientific research and computational thinking, and capturing the relevant technical knowledge. It also provides an integral platform that allows a simultaneous and comparative introduction to pipelining and parallel computing. The programming language C for developing programming models and message passing interface (MPI) and OpenMP parallelization tools have been chosen for implementation.

## 1. Background and Introduction

Teaching of scientific and parallel computing, and advanced programming are under permanent attention of scientists and educators. Different approaches, models, and solutions for teaching, assessments, and evaluation are proposed. The constructivist model for advanced programming education is one out of the presented approaches [1, 2]. In the model, a learner constructs the relevant knowledge, experiments with the provided environment, observes the results, and draws conclusions. The constructivist approach incorporates the model-centered learning as well as comparative programming teaching methods. Experimenting with the provided model, "turning" the model and observing it from different sides, comparing different solutions, and analysing and drawing conclusions, the learner improves the relevant knowledge and competences.

Modern technologies widely involve parallel computing, and plenty of scientific and industrial applications use parallel programming techniques. Teaching of parallel computing is one of the most important and challenging topics in the scientific computing and advanced programming education. In this research, we present a methodology for the introduction to scientific and parallel computing. This methodology is based on the constructivist technology and uses a model-centered approach and learning by comparison.

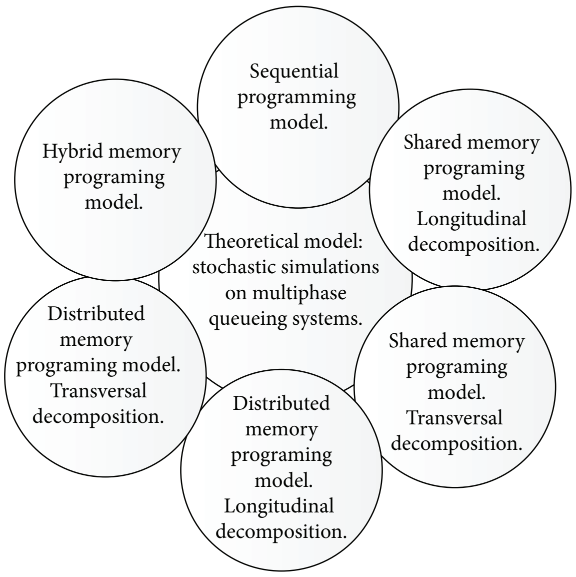

Sequential programming model.
Shared memory programing model. Longitudinal decomposition.
Shared memory programing model. Transversal decomposition.
Distributed memory programing model. Longitudinal decomposition.
Distributed memory programing model. Transversal decomposition.
Hybrid memory programing model.
Theoretical model: stochastic simulations on multiphase queueing systems.

Figure 1: Model-centered approach.

The paper proposes a set of programming models, based on stochastic simulations of the provided model of a multiphase queueing system. The multiphase queueing system is selected due to the simplicity of the primary definitions and wide possibilities for parallelization. We implement different parallelization techniques for programming the model. That allows us to carry out a series of experiments with different programming models, compare the results, and investigate the effectiveness of parallelization and different parallelization methods. Such parallelization methods include shared memory, distributed memory, and hybrid parallelization and are implemented by MPI and OpenMP APIs. Figure 1 presents the model-centered approach to the introduction into scientific computing and parallel programming.

The paper continues the earlier authors' research, presented in [3]. A possible application scope of this research could be the second level course in scientific computing and programming with an emphasis on pipeline and parallel computing programming and modelling.

*Scientific Computing in Science and Engineering Education.* Scientific computing plays an important role in science and engineering education. World leading universities and organizations pay an increasing attention to the curriculum and educational methods. Allen et al. report on a new graduate course in scientific computing that was taught at the Louisiana State University [4]. "The course was designed to provide students with a broad and practical introduction to scientific computing which would provide them with the basic skills and experience to very quickly get involved in research projects involving modern cyber infrastructure and complex real world scientific problems." Shadwick stresses the importance of more comprehensive teaching of scientific computing [5]: "...computational methods should ideally be viewed as a mathematical tool as important as calculus, and receive similar weight in curriculum."

*The Scope of Scientific Computing Education.* One of the tasks in the scientific computing education is to provide a general understanding of solving scientific problems. Heath writes, "...try to convey a general understanding of the techniques available for solving problems in each major category, including proper problem formulation and interpretation of results..." [6]. He offers a wide curriculum to be studied including a system of linear equations, eigenvalue problems, nonlinear equation, optimization, interpolation, numerical integration and differentiation, partial differential equations, fast Fourier transform, random numbers, and stochastic simulation. All these topics require a large amount of computations and could require parallelization solutions to be solved. Karniadakis and Kirby II define, "scientific computing is the heart of simulation science" [7]. "With the rapid and simultaneous advances in software and computer technology, especially commodity computing, the so-called *supercomputing*, every scientist and engineer will have on her desk an advanced simulation kit of tools consisting of a software library and multi-processor computers that will make analysis, product development, and design more optimal and cost-effective." The authors suggest the integration of teaching of MPI tools to the educational process. A large number of MPI implementations are currently available, each of which emphasizes different aspects of high-performance computing or is intended to solve a specific research problem. Other implementations deal with a grid, distributed, or cluster computing, solving more general research problems, but such applications are beyond the scope of this paper. Heroux et al. describe the scientific computing as "...a broad discipline focused on using computers as tools for scientific discovery" [8]. The authors claim, "The impact of parallel processing on scientific computing varies greatly across disciplines, but we can strongly argue that it plays a vital role in most problem domains and has become essential in many."

*Teaching of Parallel Computing.* NSF/IEEE-TCPP Curriculum Initiative on parallel and distributed computing (PDC), Core Topics for Undergraduates, contains a comprehensive research on the curriculum for parallel computing education [9]. The authors suggest including teaching of PDC: "In addition to enabling undergraduates to understand the fundamentals of 'von Neumann computing,' we must now prepare them for the very dynamic world of parallel and distributed computing." Zarza et al. report, "high-performance computing (HPC) has turned into an important tool for modern societies, becoming the engine of an increasing number of applications and services. Along these years, the use of powerful computers has become widespread throughout many engineering disciplines. As a result, the study of parallel computer architectures is now one of the essential aspects of the academic formation of students in Computational Science, particularly in postgraduate programs" [10]. The authors notice significant gaps between theoretical concepts and practical experience: "In particular, postgraduate HPC courses often present significant gaps between theoretical concepts and practical experience." Wilkinson et al. offer, "... an approach for teaching parallel and distributed computing (PDC) at the undergraduate level using computational patterns.

The goal is to promote higher-level structured design for parallel programming and make parallel programming easier and more scalable" [11]. Iparraguirre et al. share their experience in a practical course of PDC for Argentina engineering students [12]. One of the suggestions is that "Shared memory practices are easier to understand and should be taught first."

*Constructivist and Model-Centered Learning.* R. N. Caine and G. Caine in their research [13] propose the main principles of constructivist learning. One of the most important principles is as follows: "The brain processes parts and wholes simultaneously." Under this approach, a well-organized learning process should provide details as well as underlying ideas. In his research [1], Ben-Ari develops a constructivist methodology for computer-science education. Wulf [2] reviews "the application of constructivist pedagogical approaches to teaching computer programming in high school and undergraduate courses." The model-centered approach could enhance constructivist learning proposing the tool for studying and experimentation. Using model-centered learning, we first present the goal of the research after providing a model for simulation experiments. That allows us to analyze the results and to draw the relevant conclusions. Gibbons introduced model-centered instruction in 2001 [14]. The main principles are as follows:

(i) learner's *experience* is obtained by interacting with models;
(ii) learner solves scientific and engineering *problems,* using the simulation on models;
(iii) problems are presented in a constructed *sequence*;
(iv) specific instructional *goals* are specified;
(v) all the necessary *information* within a solution environment is provided.

Xue et al. [15] introduce "teaching reform ideas in the 'scientific computing' education by means of modeling and simulation." They suggest, "...the use of the modeling and simulation to deal with the actual problem of programming, simulating, data analyzing...."

## 2. Parallel Computing for Multiphase Queueing Systems

*2.1. Multiphase Queueing Systems and Stochastic Simulations.* A general multiphase queueing system consists of a number of servicing phases that provide service for arriving customers. The arriving customers move through the phases step-by-step from entrance to exit. If the servicing phase is busy with servicing the previous customer, the current customer waits in the queue in front of the servicing phase. The extended Kendall classification of queueing systems uses 6 symbols: $A/B/s/q/c/p$, where $A$ is the distribution of intervals between arrivals, $B$ is the distribution of service duration, $s$ is the number of servers, $q$ is the queueing discipline (omitted for FIFO, first in first out), $c$ is the system capacity (omitted for unlimited queues), and $p$ is the number of possible customers (omitted for open systems) [16, 17]. For example, $M/M/1$ queue represents the model having a single server, where arrivals are determined by a Poisson process, service times have an exponential distribution, and population of customers is unlimited. The interarrival and servicing times both are independent random variables. We are interested in the sojourn time of the customer in the system and its distribution. The general schema of the multiphase queueing system is presented in Figure 2.

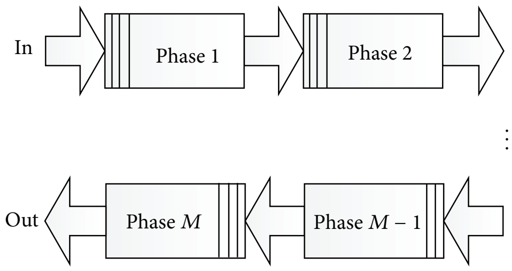


Figure 2: Multiphase queueing system.

We consider both interarrival and servicing times as exponentially distributed random variables. Depending on the parameters of the exponential distributions, we distinguish different traffic conditions for the observed queueing system. Those include ordinary traffic, critical traffic, or heavy traffic conditions. We are interested to investigate a distribution of the sojourn time for these different cases [18–20] and we will use Monte-Carlo simulations for collecting the relevant data.

*2.2. Recurrent Equation for Calculating the Sojourn Time.* In order to design a modelling algorithm of the previously described queueing system, some additional mathematical constructions should be introduced. Our aim is to calculate and investigate the distribution of the sojourn time of the number $n$ customer in the multiphase queueing system of $k$ phases. We can prove the next recurrent equation [19]: let us denote by $t_n$ the time of arrival of the $n$th customer; let us denote by $S_n^{(j)}$ the service time of the $n$th customer at the $j$th phase; $\alpha_n = t_n - t_{n-1}$; $j = 1, 2, \ldots, k$; $n = 1, 2, \ldots, N$. The following recurrence equation for calculation of the sojourn time $T_{j,n}$ of the $n$th customer at the $j$th phase is valid:

$$T_{j,n} = T_{j-1,n} + S_n^{(j)} + \max\left(T_{j,n-1} - T_{j-1,n} - \alpha_n, 0\right);$$
$$j = 1, 2, \ldots, k; \quad n = 1, 2, \ldots, N; \qquad (1)$$
$$T_{j,0} = 0, \quad \forall j; \quad T_{0,n} = 0, \quad \forall n.$$

**Proposition 1.** *The recurrence equation to calculate the sojourn time of a customer in a multiphase queuing system.*

*Proof.* It is true that if the time $\alpha_n + T_{j-1,n} \geq T_{j,n-1}$, the waiting time in the $j$th phase of the $n$th customer is 0. In the case $\alpha_n + T_{j-1,n} < T_{j,n-1}$, the waiting time in the $j$th phase of the $n$th customer is $\omega_j^n = T_{j,n-1} - T_{j-1,n} - \alpha_n$ and $T_{j,n} = T_{j-1,n} + \omega_j^n + S_n^{(j)}$. Taking into account the above two cases, we finally have the proposition results. □

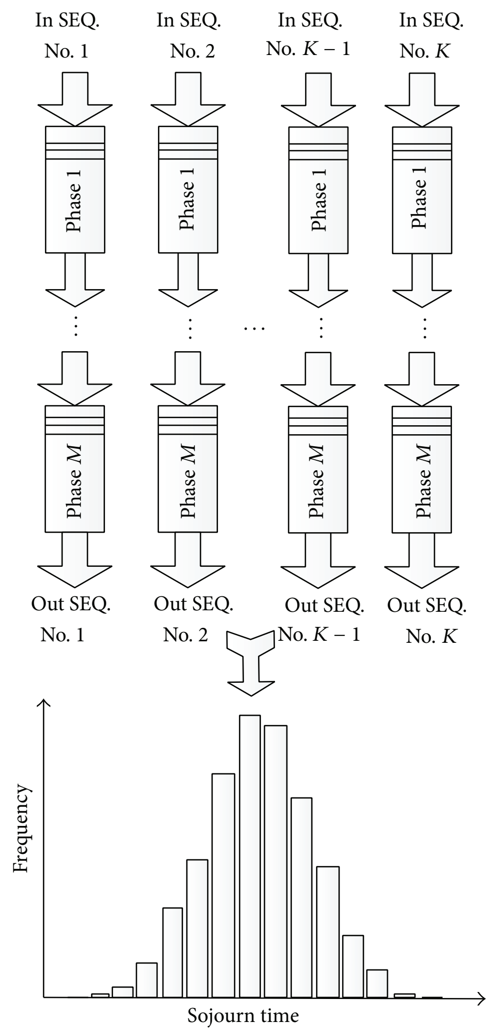


Figure 3: Statistical sampling for modelling the sojourn time distribution.

*2.3. Theoretical Background: Parallel Computing.* In this research, we emphasize a multiple instruction, multiple data (MIMD) parallel architecture and presume the high performance computer cluster (HPC) as a target platform for calculations. Such a platform allows us to study different parallelization techniques and implement shared memory, distributed memory, and hybrid memory solutions. The main goal of parallelization is to reduce the program execution time by using the multiprocessor cluster architecture. It also enables us to increase the number of Monte-Carlo simulation trials during the statistical simulation of the queueing system and to achieve more accurate results in the experimental construction of the sojourn time distribution. To implement parallelization, we use OpenMP tools for the shared memory model, MPI tools for the distributed memory model, and the hybrid technique for the hybrid memory model.

For the shared memory decomposition, we use tasks and the dynamic decomposition technique in the case of the pipeline (transversal) decomposition and we use the loop decomposition technique in the case of threads (longitudinal) decomposition. For the distributed memory decomposition, we use the standard message parsing MPI tools. For the hybrid decomposition, we use the shared memory (loop decomposition) for the longitudinal decomposition and MPI for the pipeline decomposition.

*2.4. Parallelization for Multiphase Queueing Systems.* Statistical sampling for modelling the sojourn time distribution (Figure 3) presents the general schema of the imitational experiment on the queueing system.

The programming model of the multiphase queueing system is based on the recurrent equation, presented in one of the upper sections. The Monte-Carlo simulation method is used to obtain the statistical sampling for modelling the sojourn time distribution on the exit of the system.

To introduce the topics as decomposition and granularity we present a space model of the simulation process of the queueing system. First, we start from a one-dimensional model. The one-dimensional model allows introducing a sequential programming model with no parallelism and could serve as a basic model for further improvements. Afterwards, we proceed with a two-dimensional model. Such a model allows introducing the programming models for the

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#define OUT_FILE “out_seq.txt” //results
#define MC 20 //number of Monte-Carlo simulations
#define N 1000 //number of clients
#define M 5 //number of servicing phases
//parameters of the exponential distributions
int lambda[M + 1] = {0};
double tau = 0; //interarrival time
double st = 0; //sojourn time
//sojourn time for the previous client
double st_prev[M] = {0};
//sojourn time for each MC trial
double results[MC] = {0};
int main(int argc, char *argv[]) {
gsl_rng * ran; //random generator
gsl_rng_env_setup();
ran = gsl_rng_alloc(gsl_rng_ranlxs2);
//init lambda (heavy traffic case)
lambda[0] = 30000;
for (int i = 1; i < M; i++)lambda[i] = lambda[i - 1] - 25000/M;
lambda[M] = 5000;
for (unsigned j = 0; j < MC; j++) {
tau = gsl_ran_exponential(ran, 1.0/lambda[0]);
st = 0;
for (unsigned i = 0; i < M; i++) st_prev[i] = 0.;
for (unsigned i = 0; i < N; i++) {
for (unsigned t = 0; t < M; t++) {
//recurrent equation
st += gsl_ran_exponential(ran, 1.0/lambda[t + 1]) + fmax(0.0, st_prev[t] - st - tau);
st_prev[t] = st;
}}
results [j] = st;
}
//printing results to file
FILE *fp;
const char DATA_FILE[] = OUT_FILE;
fp = fopen(DATA_FILE, “w”);
fprintf(fp, “%d%s%d%s%d\n”, N, “,”, lambda[0], “,”, lambda[M]);
for (int i = 0; i < MC - 1; i++) fprintf(fp, “%f%s”, results[i], “,”);
fprintf(fp, “%f\n”, results[MC - 1]);
fclose(fp);
gsl_rng_free(ran);
return(0);
}
```

Listing 1

longitudinal decomposition. As the last step, we introduce a three-dimensional space model. Such a model allows constructing the programming models for the transversal and hybrid decompositions.

*2.5. Stochastic Simulations and Longitudinal Decomposition.* One of the solutions is to use the longitudinal decomposition (trials) and to parallelize the Monte-Carlo trials. Thus we can map each or a group of trials depending on the preferred granularity and the total number of desired trials. The schema of the longitudinal decomposition is presented in Figure 4. A three-dimensional model of the longitudinal decomposition is presented in Figure 5.

*2.6. Pipelining and Transversal Decomposition.* In another dimension (customers) the parallelization technique is not as straightforward as it was in the previous case of parallelization of the statistical trials dimension. There arises a difficulty

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include "mpi.h"
#define OUT_FILE "out_mpi.txt"
//number of Monte-Carlo (MC) simulations in each process
#define MC 100
#define N 10000 //number of clients
#define M 5 //number of phases
#define NP 10 //number of MPI processes
void print_results(double *results, double time, int *lambda) {
FILE *fp;
const char DATA_FILE[] = OUT_FILE;
fp = fopen(DATA_FILE, "w");
fprintf(fp, "%d%s%d%s%d%s%d\n", N, ",", M, ",", lambda[0], ",", lambda[M]);
time = MPI_Wtime() – time;
fprintf(fp, "%f\n", time);
for (int i = 0; i < MC * NP – 1; i++) fprintf(fp, "%f%s", results[i], ",");
fprintf(fp, "%f\n", results[MC * NP – 1]);
fclose(fp);
}
void process(int numprocs, int myid, gsl_rng * ran, int *lambda) {
double time = MPI_Wtime(); //start time
double tau[MC] = {0}; //interarrival time
double st[MC] = {0}; //sojourn time
//sojourn time of the previous customer in each phase
double st_prev[M][MC] = {{0}};
double results[MC * NP] = {0}; //overall results
for (int j = 0; j < MC; j++) {
//init each MC trial
tau[j] = gsl_ran_exponential(ran, 1.0/lambda[0]);
st[j] = 0;
for (int i = 0; i < M; i++)
    for (int j = 0; j < MC; j++) st_prev[i][j] = 0.;
for (int i = 0; i < N; i++) {
for (int t = 0; t < M; t++) {
//recurrent equation
st[j] += gsl_ran_exponential(ran, 1.0/lambda[t + 1]) + fmax(0.0, st_prev[t][j] – st[j] – tau[j]);
st_prev[t][j] = st[j];
}}}
MPI_Gather(&st, MC, MPI_DOUBLE, &results, MC, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (myid == 0) {
print_results(&results[0], time, &lambda[0]);
}}
int main(int argc, char *argv[]) {
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int numprocs;
int myid;
gsl_rng * ran; //random generator
//parameter of the exponential distribution
int lambda[M + 1] = {0};
//init mpi
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &(numprocs));
MPI_Comm_rank(MPI_COMM_WORLD, &(myid));
MPI_Get_processor_name(processor_name, &namelen);
```

Listing 2: Continued.

```
//init parameter for the exponential distribution
lambda[0] = 30000;
for (int i = 1; i < M; i++)lambda[i] = lambda[i – 1] – 25000/M;
lambda[M] = 5000;
fprintf(stdout, "Process %d of %d is on %s\n", myid, numprocs, processor_name);
fflush(stdout);
//init random generator
gsl_rng_env_setup();
ran = gsl_rng_alloc(gsl_rng_ranlxs2);
gsl_rng_set(ran, (long) (myid) * 22);
//process
process(numprocs, myid, ran, lambda);
//finish
gsl_rng_free(ran);
MPI_Finalize();
return (0);
}
```

Listing 2

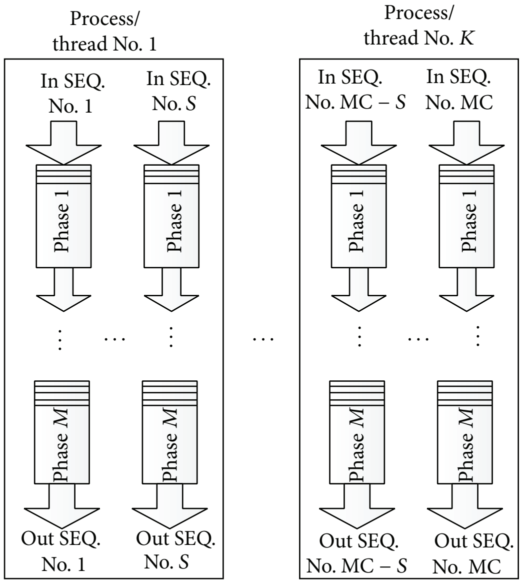


Figure 4: Longitudinal decomposition.

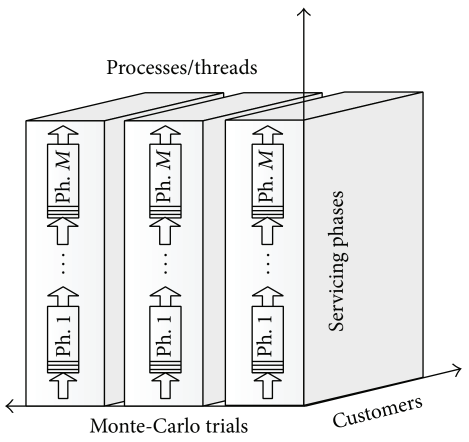


Figure 5: Three-dimensional model of the longitudinal decomposition.

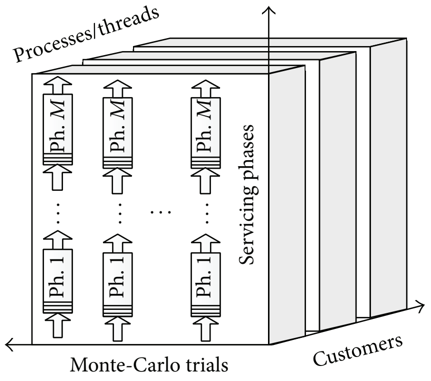


Figure 6: Transversal decomposition.

as we have the pipelining structure of the algorithm. It is obvious, as the customer moves through the system from the previous to current servicing phase, and we need to have all the data from the previous stage for calculations in the current stage. We use the transversal decomposition and the number of customers in each stage depends on the preferred granularity and the total number of customers. Figure 6 presents the decomposition in the case of the customer's dimension.

*2.7. Shared Memory Implementation.* The shared memory implementation is based on the OpenMP tools. The loop parallelization technique is used for the longitudinal decomposition. For the transversal decomposition, the OpenMP tasking model and dynamic scheduling are used.

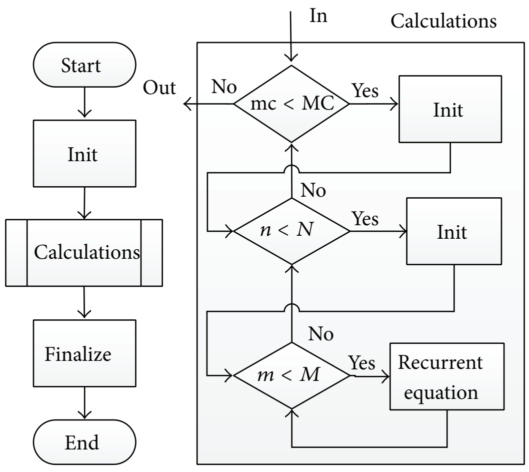


Figure 7: Sequential model flowchart.

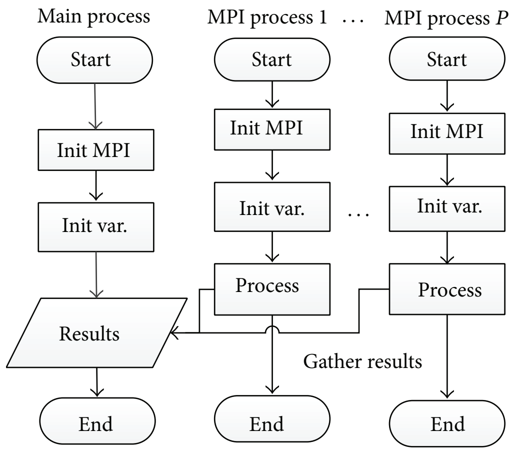


Figure 8: Distributed memory longitudinal decomposition.

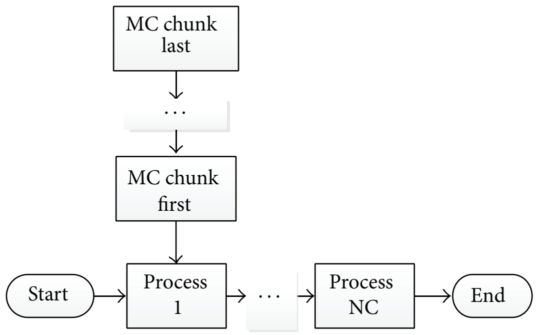


Figure 9: Distributed memory transversal decomposition.

*2.8. Distributed Memory Implementation.* The distributed memory implementation is based on MPI tools. In both cases, that is, longitudinal and transversal decompositions, the message parsing interface provides synchronization tools and there is no need for additional programming constructions.

*2.9. Hybrid Models and HPC.* Hybrid models provide a natural solution to computational platforms, based on high-performance computer clusters. It uses MPI tools to perform a decomposition and OpenMP tools for multithreading.

*2.10. Dynamic and Static Scheduling.* Pipelining requires dynamic scheduling, since there is a connection between various nodes in the pipeline. In the shared memory case we must take care of scheduling. If we use the OpenMP tasking model, the relevant approach could be twofold. First, it is possible to obtain a dynamic scheduling by monitoring a critical shared memory resource and using a task yielding construction. The other one is to use the dynamic task creation technique. In the case of MPI, synchronization is performed by the interface system, and then there is no need for additional programming constructions.

## 3. Sequential Programming Model

The flowchart of the sequential program model is presented in Figure 7. The algorithm uses the recurrent equation and cycles for modelling the queueing system phases, customers, and statistical trials.

The program model of the sequential approach uses the programming language C and GSL (GNU scientific library) and it is presented in Appendix A of this paper including the comments.

## 4. Programming Model for Distributed Memory Parallelization

*4.1. Longitudinal Decomposition.* The programming model for the distributed memory parallelization is based on MPI tools and it is optimal for the multicore/multimode computer architecture. All the processes receive a full copy of the programming code and the rooting is made by using the number of the process. The flowchart of the longitudinal decomposition is presented in Figure 8. The programming language C model with the comments is presented in Appendix B.

*4.2. Transversal Decomposition.* The flowchart for the transversal decomposition is presented in Figure 9. Processes are attached to the customer's axis which is divided into chunks. We use a mutual message parsing technique and MPI is responsible for scheduling. Statistical trials are divided into the relevant chunks and provide the desired granularity. The programming language C model with the comments is presented in Appendix C.

## 5. Programming Model for Shared Memory Parallelization

*5.1. Longitudinal Decomposition.* In the case of the shared memory model, the OpenMP loop parallelization is the natural solution to the longitudinal decomposition. The scope of MC trials is divided into chunks and each of such chunks

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include “mpi.h”
#define OUT_FILE “out_mpi_pipe.txt”
#define PIPE_MSG 0 //next pipe node
#define END_MSG 1 //finish
//number of Monte-Carlo (MC) simulations in each chunk
#define MC 10
#define NP 10 //number of processes (client axis)
#define CMC 100 //number of MC chunks
#define N 1000 //number of clients
#define M 5 //number of phases
void print_results(double *results, double time, int *lambda) {
FILE *fp;
const char DATA_FILE[] = OUT_FILE;
fp = fopen(DATA_FILE, “w”);
fprintf(fp, “%d%s%d%s%d%s%d\n”, N, “,”, M, “,”, lambda[0], “,”, lambda[M]);
time = MPI_Wtime() – time;
fprintf(fp, “%f\n”, time);
for (int i = 0; i < MC ∗ CMC – 1; i++) fprintf(fp, “%f%s”, results[i], “,”);
fprintf(fp, “%f\n”, results[MC ∗ CMC – 1]);
fclose(fp);
}
void node(int numprocs, int myid, gsl_rng ∗ ran, int *lambda) {
int nmcb = 0;
int nmcb_id = 0;
int i, j, k, t, u, v;
double time = MPI_Wtime(); //start time
MPI_Status Status;
double tau[MC] = {0}; //interarrival time
double st[MC] = {0}; //sojourn time
//sojourn time of the previous customer in each phase
double st_prev[M][MC] = {{0}};
double results[MC ∗ CMC] = {0}; //overall results
while (1) {
nmcb_id = CMC; //aux var. to omit the cycle
if (myid != 0) { //receive data from the previous node
MPI_Recv(&tau, MC, MPI_DOUBLE, myid – 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
if (Status.MPI_TAG == END_MSG) break;
MPI_Recv(&st, MC, MPI_DOUBLE, myid – 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
MPI_Recv(&st_prev, MC ∗ M, MPI_DOUBLE, myid – 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
//eliminate the below line for the other than the main thread
nmcb_id = 1;
}
//nmbc- Number of MC batches (for the main process)
for (k = 0; k < nmcb_id; k++) {
for (j = 0; j < MC; j++) {
if (myid == 0) { //init each MC trial (main process)
tau[j] = gsl_ran_exponential(ran, 1.0/lambda[0]);
st[j] = 0;
for (u = 0; u < M; u++)
      for (v = 0; v < MC; v++) st_prev[u][v] = 0.;
}
for (i = 0; i < N/NP; i++) {
for (t = 0; t < M; t++) {
```

Listing 3: Continued.

```
//recurrent equation
st[j] += gsl_ran_exponential(ran, 1.0/lambda[t + 1]) + fmax(0.0, st_prev[t][j] − st[j] − tau[j]);
st_prev[t][j] = st[j];
}}
results[j + MC ∗ nmcb] = st[j];
}
nmcb++;
if (myid != numprocs − 1) {
//if not the last process – send the data to the next process
MPI_Send(&tau, MC, MPI_DOUBLE, myid + 1, PIPE_MSG, MPI_COMM_WORLD);
MPI_Send(&st, MC, MPI_DOUBLE, myid + 1, PIPE_MSG, MPI_COMM_WORLD);
MPI_Send(&st_prev, MC ∗ M, MPI_DOUBLE, myid + 1, PIPE_MSG, MPI_COMM_WORLD);
}}
//if the main process - go out of while cycle
if (myid == 0) break;
}
//if finished - send the end msg. to the next pipe node
if (myid != numprocs − 1)
MPI_Send(&tau, MC, MPI_DOUBLE, myid + 1, END_MSG, MPI_COMM_WORLD);
//if last process - send results
if (myid == numprocs − 1)
MPI_Send(&results, MC ∗ CMC, MPI_DOUBLE, 0, PIPE_MSG, MPI_COMM_WORLD);
//print results
if (myid == 0) {
MPI_Recv(&results, MC ∗ CMC, MPI_DOUBLE, numprocs − 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
print_results(&results[0], time, &lambda[0]);
}}
int main(int argc, char *argv[]) {
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int numprocs;
int myid;
gsl_rng ∗ ran; //random generator
//parameter of the exponential distribution
int lambda[M + 1] = {0};
//init MPI
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &(numprocs));
MPI_Comm_rank(MPI_COMM_WORLD, &(myid));
MPI_Get_processor_name(processor_name, &namelen);
//init parameter for the exponential distribution
lambda[0] = 30000;
for (int i = 1; i < M; i++)lambda[i] = lambda[i − 1] − 25000/M;
lambda[M] = 5000;
fprintf(stdout, "Process %d of %d is on %s\n", myid, numprocs, processor_name);
fflush(stdout);
//init random generator
gsl_rng_env_setup();
ran = gsl_rng_alloc(gsl_rng_ranlxs2);
gsl_rng_set(ran, (long) (myid) ∗ 22);
//process
node(numprocs, myid, ran, lambda);
//finish
gsl_rng_free(ran);
MPI_Finalize();
return (0);
}
```

Listing 3

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <omp.h>
#define OPENMP 12
#define OUT_FILE "out_openmp.txt"
//number of Monte-Carlo simulations in one thread
#define MC 200
#define N 10000 //number of clients
#define M 5 //number of phases
//parameters of the exponential distribution
int lambda[M + 1] = {0};
double tau = 0; //interarrival time
double st = 0; //sojourn time
//sojourn time for the previous client
double st_prev[M] = {0};
//results- sojourn time for all threads trials
double results[MC * OPENMP] = {0};
//results- sojourn time for each thread trial
double th_results[MC] = {0};
gsl_rng * ran; //random generator
int main(int argc, char *argv[]) {
lambda[0] = 30000;
for (int i = 1; i < M; i++)lambda[i] = lambda[i - 1] - 25000/M;
lambda[M] = 5000;
unsigned long int i, t, j;
int th_id; //thread number
#pragma omp parallel num_threads(OPENMP)\
private(th_id, ran, j, i, t, tau, st, st_prev)\
firstprivate(th_results) shared(results,lambda)
{
th_id = omp_get_thread_num();
//printf("Hello World from the thread %d\n", th_id);
gsl_rng_env_setup();
ran = gsl_rng_alloc(gsl_rng_ranlxs2);
gsl_rng_set(ran, (long) th_id * 22); //seed
for (j = 0; j < MC; j++) {
tau = gsl_ran_exponential(ran, 1.0/lambda[0]);
st = 0.;
for (i = 0; i < M; i++) st_prev[i] = 0.;
for (i = 0; i < N; i++) {
for (t = 0; t < M; t++) {
//recurrent equation
st += gsl_ran_exponential(ran, 1.0/lambda[t + 1]) + fmax(0.0, st_prev[t] - st - tau);
st_prev[t] = st;
}}
th_results [j] = st;
}
for (i = 0; i < MC; i++) results[i + th_id * MC] = th_results [i];
gsl_rng_free(ran);
}
//printing results
FILE *fp;
const char DATA_FILE[] = OUT_FILE;
fp = fopen(DATA_FILE, "w");
fprintf(fp, "%d%s%d%s%d\n", N, ",", lambda[0], ",", lambda[M]);
for (i = 0; i < MC * OPENMP - 1; i++) fprintf(fp, "%f%s", results[i], ",");
fprintf(fp, "%f\n", results[MC * OPENMP - 1]);
fclose(fp);
```

Listing 4: Continued.

```
return EXIT_SUCCESS;
}
```

Listing 4

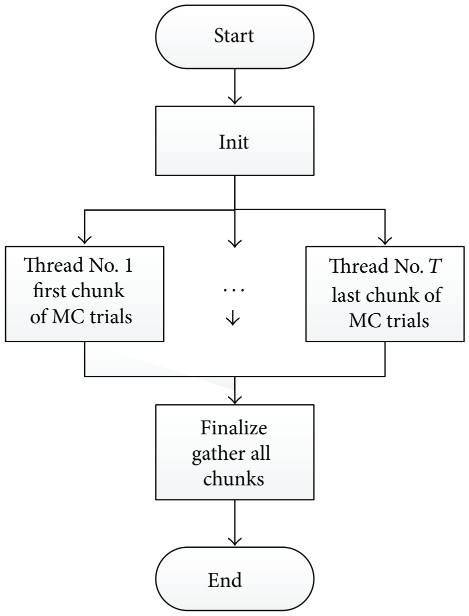

Start
Init
Thread No. 1 first chunk of MC trials
. . .
Thread No. T last chunk of MC trials
Finalize gather all chunks
End

Figure 10: Shared memory longitudinal decomposition.

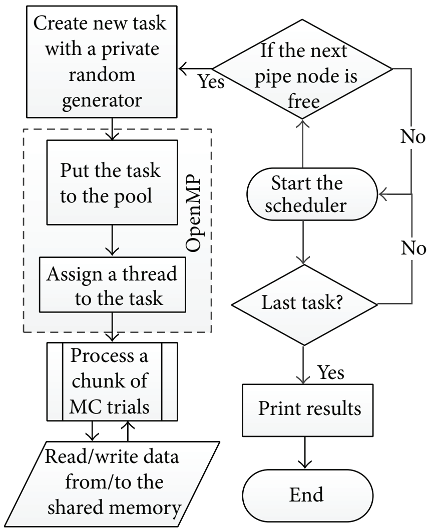

Create new task with a private random generator
Yes
If the next pipe node is free
No
Put the task to the pool
OpenMP
Start the scheduler
No
Assign a thread to the task
Last task?
Process a chunk of MC trials
Yes
Print results
Read/write data from/to the shared memory
End

Figure 11: Shared memory transversal decomposition.

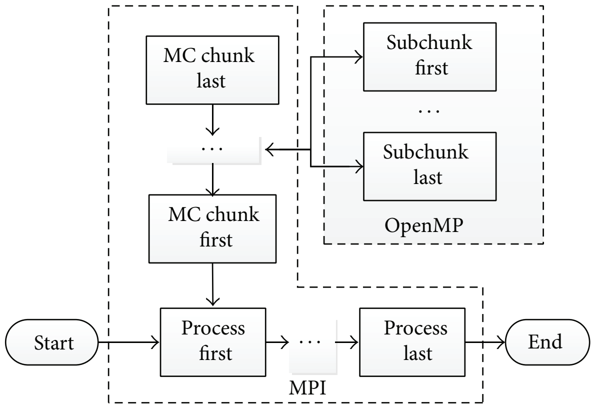


Figure 12: Hybrid decomposition.

is attached to the relevant thread. The flowchart of the shared memory model is presented in Figure 10.

The programming model for the shared memory longitudinal decomposition uses the programming language C, GSL (GNU scientific library), and OpenMP libraries. The model and comments are presented in Appendix D of this paper.

*5.2. Transversal Decomposition.* One of the most comprehensive programming models is the model of the shared memory pipelining. We use the transversal decomposition to construct such a model. The model uses the OpenMP tasking technique and dynamic scheduling of tasks. The scheduler plays the central role in the model and it is responsible for creating new tasks and finishing the program, after completing all the tasks. Each task uses its own random generator which allows avoiding time-consuming critical sections. The flowchart is presented in Figure 11. Programming model for the shared memory transversal decomposition uses C programming language, GSL (GNU scientific library), and OpenMP libraries. The model and comments are presented in Appendix E of this paper.

## 6. Programming Model of Hybrid Parallelization

The MPI transversal decomposition model could be transformed into a hybrid model by adding the OpenMP threads to the MC trials axis. The flowchart of the hybrid model is presented in Figure 12. The programming model with comments is presented in Appendix F.

## 7. Conclusions

*7.1. Theoretical and Programming Models: The Basis of the Model-Centered Approach.* The paper provides a number of programming models for the introduction to scientific and parallel computing. All these programming models, sequential, distributed memory, distributed memory pipelining,

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <omp.h>
#define OUT_FILE "out_omp_pipe.txt"
//total number of Monte-Carlo (MC) simulations
#define MC 10000
#define CMC 100 //chunks per MC axis
#define N 10000 //total number of clients
#define M 5 //total number of phases
#define CN 100 //chunks per clients axis
#define OPENMP 12 //OMP threads
//parameters of exponential distributions
int lambda[M + 1] = {0};
//interarrival time for each MC trial
double tau[MC] = {0};
double st[MC] = {0}; //sojourn time
//sojourn time of the previous client
//each MC trial and each phase
double st_prev[MC][M] = {{0}};
//aux variable - each MC chunk flag
int flag[CMC] = {0};
//aux variable - each MC chunk counter
int task_counter[CMC] = {0};
int main(int argc, char *argv[]) {
time_t t1, t2;
t1 = time(NULL);
//init exponential distribution parameters
lambda[0] = 30000;
for (int i = 1; i < M; i++)lambda[i] = lambda[i – 1] – 25000/M;
lambda[M] = 5000;
gsl_rng ∗ ran; //random generator
gsl_rng_env_setup();
ran = gsl_rng_alloc(gsl_rng_ranlxs2);
gsl_rng_set(ran, (long) (CN ∗ CMC + 10) ∗ 22.); //seed
//set interarrival time for each MC trial
for (int i = 0; i < MC; i++)
tau[i] = gsl_ran_exponential(ran, 1.0/lambda[0]);
gsl_rng_free(ran);
omp_set_num_threads(OPENMP);
#pragma omp parallel //start threads
{
#pragma omp single //one thread to create tasks
{
int i, j, t, c; //aux variables
int v = 0; //local variable - MC chunk number
int sum = 0; //local variable - number of tasks
int while_flag = 1; //var. to stop external while
//dynamic task creation in each of MC chunks
while (while_flag) {
if (!flag[v]) {
flag[v] = 1;
//create a new task if previous task had finished
#pragma omp task default(none)\private(i, j, t, c, ran)\
firstprivate(tau, lambda, v, gsl_rng_ranlxs2)\
shared(st, st_prev, flag, task_counter)
{
```

Listing 5: Continued.

```
gsl_rng * ran; //random generator for this task
gsl_rng_env_setup();
ran = gsl_rng_alloc(gsl_rng_ranlxs2);
//seed with the task number
gsl_rng_set(ran, (long)(task_counter[v] + v * CN) * 22.);
for (j = 0; j < MC/CMC; j++) {
c = j + v * (MC/CMC); //MC trial number
for (i = 0; i < N/CN; i++) {
for (t = 0; t < M; t++) {
//recurrent equation
st[c] += gsl_ran_exponential(ran, 1.0 /lambda[t + 1]) + fmax(0.0, st_prev[c][t] − st[c] − tau[c]);
st_prev[c][t] = st[c];
}}}
//end of the current task
gsl_rng_free(ran);
task_counter[v]++;
flag[v] = 0;
}}
//if all tasks of this chunk of Monte-Carlo trials
//had finished -then stop this chunk
//v numbered MC chunk is over
if (task_counter[v] == CN) flag[v] = 1;
v++;
if (v == CMC) v = 0; //again a new loop
sum = 0; //variable to test all chunks
for (i = 0; i < CMC; i++) sum += task_counter[i];
//if all task had finished - exit while cycle
if (sum == CN * CMC) while_flag = 0;
}}}
//print results
FILE *fp;
const char DATA_FILE[] = OUT_FILE;
fp = fopen(DATA_FILE, "w");
fprintf(fp, "%d%s%d%s%d%s%d\n", N, ",", M, ",", lambda[0], ",", lambda[M]);
t2 = time(NULL);
fprintf(fp, "%f\n", difftime(t2, t1));
for (int j = 0; j < MC − 1; j++) fprintf(fp, "%f%s", st[j], ",");
fprintf(fp, "%f\n", st[MC − 1]);
fclose(fp);
return (0);
}
```

Listing 5

shared memory, shared memory pipelining, and the hybrid model, are based on statistical simulations of the theoretical model of the multiphase queueing system. After providing a theoretical background to the learner and explaining the main features of the theoretical model, we start experiments with programming models. The relevant problems could be provided to the learners. These could include the comparative investigation of the effectiveness of programming models, taking into account different computational platforms as well as different input parameters of the queueing system. For the advanced learner, the emphasis could be put on variation of the parameters of interarrival and servicing time exponential distributions, moving from the heavy traffic to the nonheavy traffic case, since that could fundamentally change the distribution of the sojourn time of the customer.

*7.2. Introduction to Scientific Computing: Research Tasks and Research Methods.* While investigating the theoretical model, studying the recurrent equation, and experimenting with the input parameters of the queueing system, we provide an introduction to the scientific research tasks and methods. It includes studying the distribution of the sojourn time of the customer, varying the parameters of interarrival and servicing time exponential distributions, comparing the results, analyzing the provided theoretical constructions, and studying the Monte-Carlo method for statistical simulations, which is one of the basic methods in studying the topics related with probability.

*7.3. Introduction to Parallel Computing: Terminology and Methodology.* Studying and experimenting with the programming models, the introduction to parallel computing

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <omp.h>
#include “mpi.h”
#define OUT_FILE “out_hybrid.txt”
#define PIPE_MSG 0 //next pipe node
#define END_MSG 1 //finish
#define OPENMP 12 //number of OMP threads
//number of Monte-Carlo (MC) simulations in each chunk
#define MC 100
#define NP 10 //number of processes (client axis)
#define CMC 100 //number of MC chunks
#define N 1000 //number of clients
#define M 5 //number of phases
void print_results(double *results, double time, int *lambda) {
FILE *fp;
const char DATA_FILE[] = OUT_FILE;
fp = fopen(DATA_FILE, “w”);
fprintf(fp, “%d%s%d%s%d%s%d\n”, N, “, ”, M, “, ”, lambda[0], “, ”, lambda[M]);
time = MPI_Wtime() − time;
fprintf(fp, “%f\n”, time);
for (int i = 0; i < MC ∗ CMC − 1; i++) fprintf(fp, “%f%s”, results[i], “, ”);
fprintf(fp, “%f\n”, results[MC ∗ CMC − 1]);
fclose(fp);
}
void node(int numprocs, int myid, gsl_rng ∗ ran, int *lambda) {
int nmcb = 0; //nmbc- Number of MC batches
int nmcb_id = 0;
int i, j, k, t, u, v;
double time = MPI_Wtime(); //program start time
MPI_Status Status;
double tau[MC] = {0}; //interarrival time
double st[MC] = {0}; //sojourn time
//sojourn time of the previous customer in each phase
double st_prev[M][MC] = {{0}};
double results[MC ∗ CMC] = {0}; //overall results
double temp; //aux variable
while (1) {
nmcb_id = CMC; //aux var. to omit the cycle
if (myid != 0) { //receive data from the previous node
MPI_Recv(&tau, MC, MPI_DOUBLE, myid − 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
if (Status.MPI_TAG == END_MSG) break;
MPI_Recv(&st, MC, MPI_DOUBLE, myid − 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
MPI_Recv(&st_prev, MC ∗ M, MPI_DOUBLE, myid − 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
//eliminate below for in case of not the main thread
nmcb_id = 1;
}
for (k = 0; k < nmcb_id; k++) {
omp_set_num_threads(OPENMP);
{
#pragma omp parallel for default(shared)
\private(j, i, t, u, v)
for (j = 0; j < MC; j++) {
if (myid == 0) { //init each MC trial (main process)
#pragma omp critical
{
```

Listing 6: Continued.

```
tau[j] = gsl_ran_exponential(ran, 1.0/lambda[0]);
}
st[j] = 0;
for (u = 0; u < M; u++)
    for (v = 0; v < MC; v++) st_prev[u][v] = 0.;
}
for (i = 0; i < N/NP; i++) {
for (t = 0; t < M; t++) {
//recurrent equation
temp = gsl_ran_exponential(ran, 1.0/lambda[t + 1]);
#pragma omp critical
{
temp = gsl_ran_exponential(ran, 1.0/lambda[t + 1]);
}
st[j] += temp + fmax(0.0, st_prev[t][j] – st[j] – tau[j]);
st_prev[t][j] = st[j];
}}
results[j + MC * nmcb] = st[j];
}}
nmcb++;
if (myid != numprocs – 1) {
//if not the last process send data to the next process
MPI_Send(&tau, MC, MPI_DOUBLE, myid + 1, PIPE_MSG, MPI_COMM_WORLD);
MPI_Send(&st, MC, MPI_DOUBLE, myid + 1, PIPE_MSG, MPI_COMM_WORLD);
MPI_Send(&st_prev, MC * M, MPI_DOUBLE, myid + 1, PIPE_MSG, MPI_COMM_WORLD);
}
}
//if the main process - go out of while cycle
if (myid == 0) break;
}
//if finished - send the end msg. to the next pipe node
if (myid != numprocs – 1)MPI_Send(&tau, MC, MPI_DOUBLE, myid + 1, END_MSG, MPI_COMM_WORLD);
//if last process - send results
if (myid == numprocs – 1)
MPI_Send(&results, MC * CMC, MPI_DOUBLE, 0, PIPE_MSG, MPI_COMM_WORLD);
//print results
if (myid == 0) {
MPI_Recv(&results, MC * CMC, MPI_DOUBLE, numprocs – 1, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
print_results(&results[0], time, &lambda[0]);
}
}
int main(int argc, char *argv[]) {
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int numprocs;
int myid;
gsl_rng * ran; //random generator
//parameter for the exponential distribution
int lambda[M + 1] = {0};
//init mpi
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &(numprocs));
MPI_Comm_rank(MPI_COMM_WORLD, &(myid));
MPI_Get_processor_name(processor_name, &namelen);
//init parameter for the exponential distribution
lambda[0] = 30000;
for (int i = 1; i < M; i++)lambda[i] = lambda[i – 1] – 25000 / M;
lambda[M] = 5000;
fprintf(stdout, "Process %d of %d is on %s\n", myid, numprocs, processor_name);
fflush(stdout);
```

Listing 6: Continued.

```
//init random generator
gsl_rng_env_setup();
ran = gsl_rng_alloc(gsl_rng_ranlxs2);
gsl_rng_set(ran, (long) (myid) * 22);
//process
node(numprocs, myid, ran, lambda);
//finish
gsl_rng_free(ran);
MPI_Finalize();
return (0);
}
```

Listing 6

terminology and methodology is provided. It includes the basic concepts such as shared and distributed memory parallelization techniques, homogenous and heterogeneous computational platforms, and HPC and multicore programming and it explains scheduling, mapping, and granularity. Using MPI and OpenMP tools, we provide the introduction to OpenMP tasking, MPI programming methods, synchronization, load balancing, decomposition techniques, and other important topics of parallel computing.

*7.4. Problems: Studying Effectiveness, Debugging, and Benchmarking.* We could enhance the learner's understanding by providing a set of problems such as debugging, benchmarking, and comparative studying of the effectiveness of programming models. That includes variation of different computing platforms for one of the models as well as testing different models for a definite platform. It could be done by applying a single processor, multicore, and multiprocessor machines and computer clusters. As an example, the model with efficient results achieved by a single-processor machine could be inefficient on the other platforms. When modifying the model, the relevant debugging tools and methods must be implemented. So the learner could proceed by modifying the model, that is, changing the distribution parameters and tuning the granularity.

*7.5. Further Studies: Queueing Networks for Constructing Learning Objects.* The theory of queueing systems renders wide possibilities for relevant theoretical constructions. The next obvious step could be studies of queueing networks of various types including open, closed, or mixed networks, constructing the relevant learning objects, and investigating the respective theoretical and programming models.

## Appendices

## A. Sequential Programming Model

See Listing 1.

## B. Distributed Memory Programming Model

See Listing 2.

## C. Distributed Memory Pipeline Model

See Listing 3.

## D. Shared Memory Programming Model

See Listing 4.

## E. Shared Memory Pipeline Programming Model

See Listing 5.

## F. Hybrid Programming Model

See Listing 6.

## Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

## References

[1] M. Ben-Ari, "Constructivism in computer science education," *Journal of Computers in Mathematics and Science Teaching*, vol. 20, pp. 45–73, 2001.

[2] T. Wulf, "Constructivist approaches for teaching computer programming," in *Proceedings of the 6th ACM SIG-Information Technology Education Conference (SIGITE '05)*, pp. 245–248, ACM, October 2005.

[3] V. Dolgopolovas, V. Dagienė, S. Minkevičius, and L. Sakalauskas, "Python for scientific computing education: modeling of queueing systems," *Scientific Programming*, vol. 22, no. 1, pp. 37–51, 2014.

[4] G. Allen, W. Benger, A. Hutanu, S. Jha, F. Löffler, and E. Schnetter, "A practical and comprehensive graduate course

preparing students for research involving scientific computing," *Procedia Computer Science*, vol. 4, pp. 1927–1936, 2011.

[5] B. Shadwick, "Teaching scientific computing," in *Computational Science—ICCS 2004*, vol. 3039, pp. 1234–1241, Springer, Berlin, Germany, 2004.

[6] M. T. Heath, *Scientific Computing*, McGraw-Hill, 1997.

[7] G. E. Karniadakis and R. M. Kirby II, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*, Cambridge University Press, Cambridge, UK, 2003.

[8] M. A. Heroux, P. Raghavan, and H. D. Simon, *Parallel Processing for Scientific Computing*, Society for Industrial and Applied Mathematics, 2007.

[9] S. K. Prasad, A. Chtchelkanova, S. Das et al., "NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing—core topics for undergraduates," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*, pp. 617–618, ACM, March 2011.

[10] G. Zarza, D. Lugones, D. Franco, and E. Luque, "An innovative teaching strategy to understand high-performance systems through performance evaluation," *Procedia Computer Science*, vol. 9, pp. 1733–1742, 2012.

[11] B. Wilkinson, J. Villalobos, and C. Ferner, "Pattern programming approach for teaching parallel and distributed computing," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*, pp. 409–414, ACM, March 2013.

[12] J. Iparraguirre, G. R. Friedrich, and R. J. Coppo, "Lessons learned after the introduction of parallel and distributed computing concepts into ECE undergraduate curricula at UTN-Bahía Blanca Argentina," in *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12)*, pp. 1317–1320, IEEE, Shanghai, China, May 2012.

[13] R. N. Caine and G. Caine, *Making Connections: Teaching and the Human Brain*, 1991.

[14] A. S. Gibbons, "Model-centered instruction," *Journal of Structural Learning and Intelligent Systems*, vol. 14, pp. 511–540, 2001.

[15] L. Xue, M.-H. Wu, H. Zheng, H.-Z. Zhang, and W.-B. Huang, "Modeling and simulation in scientific computing education," in *Proceedings of the International Conference on Scalable Computing and Communications—8th International Conference on Embedded Computing (ScalCom-EmbeddedCom '09)*, pp. 577–580, IEEE, September 2009.

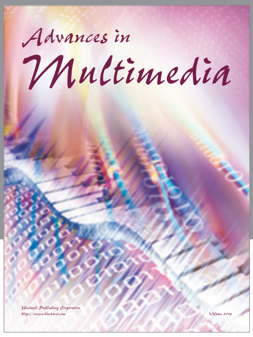
[16] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain," *The Annals of Mathematical Statistics*, vol. 24, no. 3, pp. 338–354, 1953.

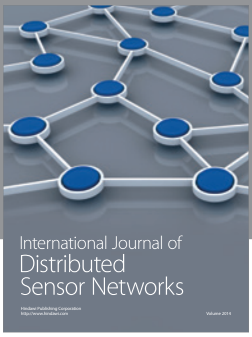
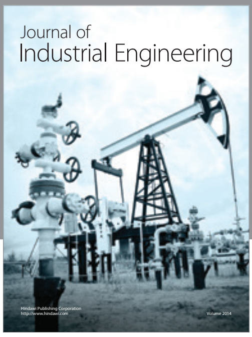
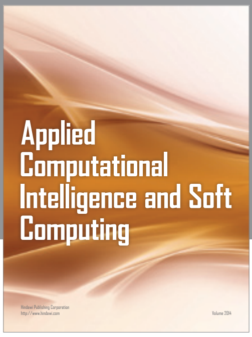
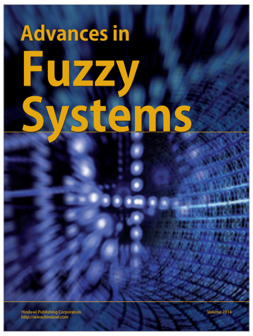
[17] R. P. Sen, *Operations Research: Algorithms and Applications*, PHI Learning, 2010.

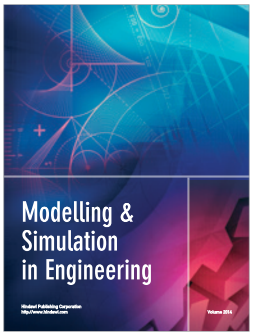
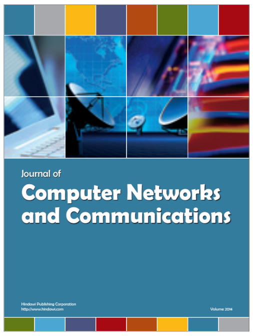
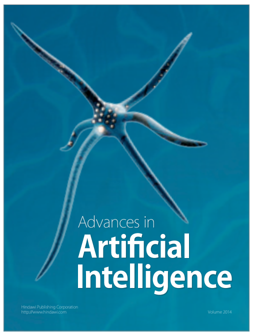
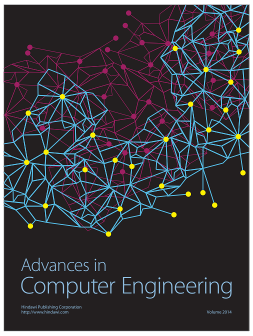
[18] U. N. Bhat, *An Introduction to Queueing Theory Modeling and Analysis in Applications*, Birkhäuser, Boston, Mass, USA, 2008.

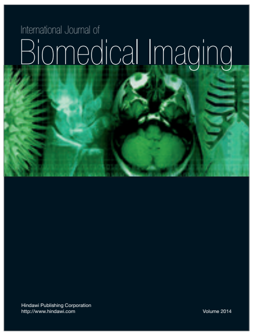
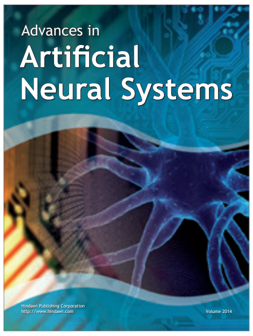
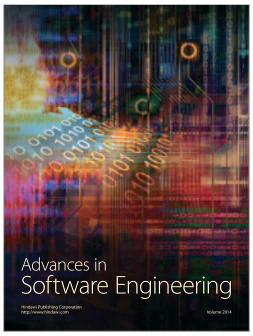
[19] G. I. Ivcenko, V. A. Kastanov, and I. N. Kovalenko, *Queueing System Theory*, Vishaja Skola, Moscow, Russia, 1982.

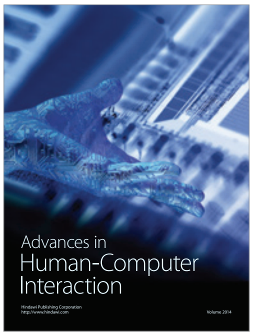
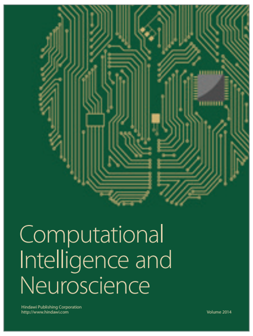
[20] S. Minkevičius and V. Dolgopolovas, "Investigation of the fluid limits for multiphase queues in heavy traffic," *International Journal of Pure and Applied Mathematics*, vol. 66, no. 2, pp. 177–182, 2011.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems


Modelling & Simulation in Engineering

Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering