Applying High Performance Computing to Analyzing by Probabilistic Model Checking Mobile Cellular Networks with Spectrum Renting*

Wolfgang Schreiner and Nikolaj Popov
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at, Nikolaj.Popov@risc.jku.at

Tamás Bérczes and János Sztrik
Department of Informatics Systems and Networks, Faculty of Informatics
University of Debrecen, Debrecen, Hungary
sztrik.janos@inf.unideb.hu, berczes.tamas@inf.unideb.hu

Gábor Kusper
Eszterházy Károly College, Eger, Hungary
gkusper@aries.ektf.hu

July 25, 2013

Abstract

We report on the use of high performance computing in order to analyze with the probabilistic model checker PRISM mobile cellular networks, in particular the system described in the paper "A New Finite-Source Queueing Model for Mobile Cellular Networks Applying Spectrum Renting" by Tien v. Do et al. That paper proposes a new finite-source retrial queueing model to consider spectrum renting in mobile cellular networks; numerical results are there produced with the MOSEL-2 tool. Our results show that the model can be also appropriately described and analyzed in PRISM, but that modeling becomes comparatively more cumbersome due to the lack of zero-time/infinite-rate transitions in PRISM. By using a massively parallel non-uniform memory architecture (NUMA), we are able to considerably speed up the analysis of large scale models.

*Supported by the European project FIRST — Future Internet Research, Services and Technology, Research of Future Internet from Theory to Applications (TÁMOP-4.4.2.C) and by the project 85öu8 of the Stiftung Aktion Österreich-Ungarn.

Contents

1. Introduction

We report in this paper on the use of high performance computing resources in order to analyze mobile cellular networks by probabilistic model checking. Our focus is the mobile cellular network system that was introduced in [2] and analyzed there with the help of the performance modeling tool MOSEL-2 [1]. In this model, a number of sources (cell phone subscribers) compete for access to a number of servers (channels). Sources produce requests at rate λ which a free server processes at rate μ. However, the number of available channels is not fixed: if the number of free channels gets to small, the cell phone operator may rent additional frequency blocks from another operator, partition these blocks into channels, and use the new channels for its own subscribers; these blocks may be released, if sufficiently many channels have become free again.

Our own work is based on the probabilistic model checker PRISM [3, 4] which is actively developed and has been used for numerous purposes, among them the performance analysis of computing systems. Based on a prior analysis of the various possibilities to analyze execution times in PRISM [5], we have in [6] reported on initial results of modeling and analyzing the system sketched above in PRISM. In this paper, we present a corrected model and show how high performance computing resources, specifically a massively parallel non-uniform memory architecture (NUMA), can be used to speed up the analysis of such models.

The remainder of this paper is structured as follows: in Section 2, we discuss the revised PRISM model of the system; in Section 3, we present the parallel execution framework that we have devised for performing the analysis; in Section 4 we show the data derived from the analysis and the times required to produce them on a NUMA architecture; Section 5 presents our conclusions and open issues for further work.

2. The Model

In [6], we have already discussed in great detail a PRISM model for the system presented in [2]. However, as was detected by one of the authors (Tamas Berczes) of [2], this PRISM model was incorrect in the way it handled the release from a subscriber from the orbit; while the original MOSEL model gave her immediate access to a channel (if a free channel was available) or placed her in the queue (if no free channel was available), the PRISM model put her back into the set of sources from which she had to issue a new request to get access to a channel. Furthermore, the PRISM model handled the transition from the queue to the server by a transition with "infinite" (very high) rate which was considered as awkward.

Both issues ultimately boiled down to the following problem: MOSEL supports zero-time (infinite-rate) transitions from a state b to other states $c_1, \dots, c_n$ where the transitions are guarded by appropriate conditions to proceed to the the desired target state. This state b may be the target of several non-zero-rate transitions from other states $a_1, \dots, a_m$, i.e., after a certain delay from any of these states the choice to proceed to $c_1, \dots, c_n$ can be made. In PRISM, however, there are no such zero-time (infinite-rate) state transitions; a PRISM model mimicking a MOSEL model has to include $m \cdot n$ transitions from each source state $a_1, \dots, a_m$ to each target state $c_1, \dots, c_n$ without the convenience of a common intermediate state b.

The first issue of our previous paper (the discrepancy between the MOSEL model and the PRISM model) was the result of trying to avoid the combinatorial explosion of transitions arising from the MOSEL model by using another state as such an intermediate state *b*, namely one where an element from the orbit was placed into the queue; the second issue was the result of trying to avoid the combinatorial explosion of transitions by mimicking a zero time MOSEL transition by a corresponding PRISM transition with very high rate.

Appendix A presents an adapted PRISM model where the two issues have been corrected at the price of a substantial increase in the number of transitions. This admittedly makes the model somewhat less transparent than the original one; in particular, two transitions ("success2", and "retrial2") are synchronized between *three* modules (rather than two) corresponding to a simultaneous step performed by three components of the system.

We still leave the "infinite-rate" transition "interrupt", which immediately returns from an attempt to allocate a block, if it is detected that the allocation is not needed any more; as stated by Tamas Berczes, this transition has also a somewhat dubious status in the MOSEL model and should be probably replaced by a transition with a finite rate.

The other open questions raised in [6] could be satisfactorily answered:

- The question concerning the zero time transition in the Queue component was already discussed above. The question of a guard condition which was supposed to be missing was based on a misunderstanding of the MOSEL code semantics, since a previous guard condition in a zero-time transition makes such a condition in the MOSEL code unnecessary (thanks to Tamas Berces for pointing this out).

- The PRISM bug which prevented model generation for $K \geq 60$ can be circumvented by the command line option `-nocompact` respectively by deselecting the option "Use compact schemes" in the graphical user interface (thanks to David Parker for pointing this out).

In the following section, we will discuss how our new PRISM model can be efficiently analyzed in a high performance computing environment.

3. The Parallel Execution Framework

The computing center of the University of Debrecen, whose project is funding the work presented in this paper, hosts a Silicon Graphics International (SGI) Altix ICE8400EX supercomputer with 256 Intel Xeon X5680 processors with 6 cores each; the system thus supports computations with up to 1536 cores. The SGI Altix is a non-uniform memory (NUMA) architecture where all cores have access to the same virtual shared memory; however the time for memory access considerably varies with the physical location of the memory. A thread can access memory within the current node (each node holds 2 processors, i.e., 12 cores) very fast but access to memory on another node requires communication via the internal Infiniband network and is thus an order of magnitude slower. Access to that machine is given via the Open Grid Scheduler batch queueing system.

However, since this machine was very heavily utilized, we performed the experiments reported in this paper on another machine of very similar type, an SGI Altix UltraViolet 1000 supercomputer installed at the Johannes Kepler University Linz. This machine is equipped with 256 Intel Xeon E78837 processors with 8 cores each which are distributed among 128 nodes with 2 processors (i.e. 16 cores) each; the system thus supports computations with up to 2048 cores. Access to this machine is possible via interactive login; by default every user may execute threads on 4 processors with 32 cores and 256 GB memory.

The PRISM model checker is implemented in Java (with core libraries implemented in C); we use for our experiments the Oracle Java Platform, specifically the most recent version Java SE 7u25. Since this platform makes use of parallel threads for garbage collection, it is for good performance crucial that all memory and all threads of the Java process are allocated on the same node. For this purpose, we define the following script `prism-java` (see also Appendix C) and set the environment variable `PRISM_JAVA` (which tells PRISM the location of the Java executable) to the path of this script:

```
#!/bin/sh
t=`date +%N`
n=`expr $t % 4`
numactl --cpunodebind=+$n --membind=+$n java \
  -XX:+UseParallelOldGC -XX:ParallelGCThreads=4 -XX:+AggressiveOpts $*
```

The script calculates a random number $n \in [0,3]$ and applies the `numactl` command to bind all threads and all memory of the Java program to the $(n+1)$-th processor which is available to the current user. The additional Java option `-XX:+UseNUMA` to switch on a NUMA-enhanced version of garbage collection can unfortunately not be used because it lets the Java Virtual Machine crash with a segmentation violation (an apparent bug in the Java platform).

In order to perform our experiments, we need to call PRISM numerous times with different arguments. Rather than using some job scheduling system, we control the rate of command execution by a C program `parallel` which we have written for this purpose. This program (whose source code is given in Appendix D[1]) reads an arbitrary number of command lines from the standard input, executes these commands by a given number of processes, and prints out status information for every command whose execution has terminated.

We then can perform experiments by a shell script that echos all commands that have to be performed for an experiment to the standard input of the `parallel` command; the script terminates when all commands have been executed. Appendix B shows an example of such a script whose core is essentially as follows:

```
(
  for PROPERTY in Pblock mO mTO mB mQ mTQ mC mAS ; do
    for RHO in $(seq 0.6 0.5 4.6) ; do
      for T1 in $(seq 1 1 4) ; do
        echo "prism Spectrum.prism Spectrum.props -prop $PROPERTY \
          -const rho=$RHO,t1=$T1 \
          -exportresults Results/$PROPERTY-$T1-$RHO \
          > Logfiles/$PROPERTY-$T1-$RHO"
      done
    done
```

[1]The source code can be downloaded from `http://www.risc.jku.at/people/schreine/parallel/parallel.tgz`.

P	t_p (s)			T_p (s)	S_p
1	1796	1808	1836	1813	1.0
2	937	940	954	944	1.9
4	491	498	524	504	3.6
8	267	294	299	287	6.3
16	155	158	169	161	11.3
32	120	140	144	135	13.4

Figure 1: Analysis Times and Speedups

```
    done
) | parallel $PROC
```

This script executes PRISM with *PROC* processes to analyze every property *PROPERTY* for all parameters *RHO* and $T1$ and writes the result of every analysis to a separate file; we have used this script to perform the analysis which is going to be presented in the following section.

4. The Analysis

With the help of the parallel execution framework presented in the previous section, we have performed several experiments whose results are illustrated in Figure 2 (the same experiments were performed in [6] for model size $K = 60$, here they are performed for $K = 100$). This figure (which was created with the help of the scripts listed in Appendix E and F) corresponds to Figure 3 of [2]; the results for *Pblock*, *mO*, *mTO*, *mQ* and *mTQ* seem identical, but the results in *mAS*, *mB*, and *mC* are different, especially for small ρ: in particular, we derive a smaller number of used channels (active calls, property *mC*) than reported in [2]. The reasons for this discrepancy are still unclear.

As for the time needed for executing the analysis, Figure 1 lists the times (in seconds) for performing all the 256 checks illustrated in Figure 2 with P processes, $1 \leq P \leq 32$ (the maximum number of processor cores available to us for this experiment). The analysis was performed three times (for a slightly different model), leading to three values for the execution time t_p with average execution time T_p; the absolute speedup for this average is reported as S_p. We see that significant speedups up to a maximum of 13.4 can be achieved; for $P = 32$, improvements are still visible but will (for this experiment) certainly become marginal for a larger number of processes.

5. Conclusions

We have shown in this report how a non-trivial mobile cellular network can be modeled and analyzed in PRISM and how this analysis can be efficiently performed on a modern high performance computing system of the NUMA type by shell scripting with the help of a small parallel execution framework. One may wonder why we have described the overall framework in much

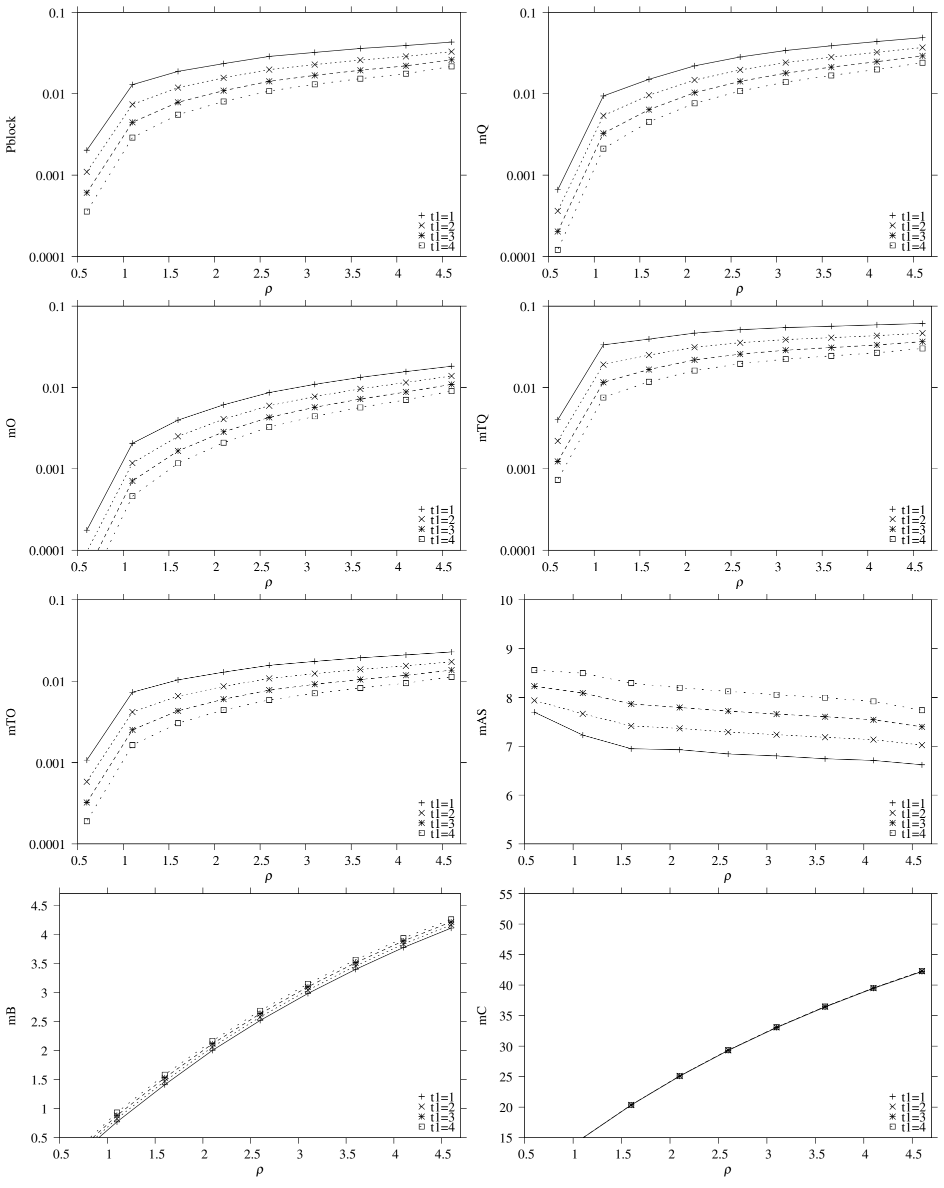

Figure 2: Performance Measures

more detail (including all source and scripting code) than is usual for a scientific paper. The reason is that, while the technicalities are in principle not very difficult, it takes considerable time to have them properly elaborated; we would like to spare in the future others (including ourselves) from this work and let them save valuable time for more fundamental research.

As for the contents of the analysis, we were able to correct some mistakes and problems that arose in a previous presentation of the model. We realized that a crucial difference between MOSEL-2 and PRISM (the existence respectively lack of zero-time/infinite-rate transitions) may cause the PRISM model to become more unhandy than originally thought; more efforts have to be invested to elaborate how to express in PRISM the desired models in a more economical way.

Furthermore, we are now able to reproduce some of the previously reported results but there still remain some discrepancies to be resolved. Once this has been achieved we will focus on further novel models for which no preceding modeling and analysis is available.

Acknowledgments

The author thanks David Parker for help with PRISM and in particular Tamas Berczes for helping to correct the previous version of the PRISM model.

References

[1] K. Begain, G. Bolch, and Herold H. *Practical Performance Modeling Application of the MOSEL Language*. Kluwer Academic Publisher, 2012.

[2] Tien v. Do, Patrick Wüchner, Tamas Berczes, Janos Sztrik, and Hermann de Meer. "A New Finite-Source Queueing Model for Mobile Cellular Networks Applying Spectrum Renting". In: *Asia-Pacific Journal of Operational Research* (2013). To appear.

[3] M. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591.

[4] David A. Parker, ed. *PRISM — Probabilistic Symbolic Model Checker*. Department of Computer Science, University of Oxford, UK. 2013. URL: http://www.prismmodelchecker.org.

[5] Wolfgang Schreiner. *Experiments with Measuring Time in PRISM 4.0*. Technical Report. Johannes Kepler University Linz, Austria: Research Institute for Symbolic Computation (RISC), Mar. 2013. URL: http://www.risc.jku.at/publications/download/risc_4684/main.pdf.

[6] Wolfgang Schreiner. *Initial Results on Modeling in PRISM Mobile Cellular Networks with Spectrum Renting*. Technical Report. Johannes Kepler University Linz, Austria: Research Institute for Symbolic Computation (RISC), Mar. 2013. URL: http://www.risc.jku.at/publications/download/risc_4705/main.pdf.

A. The PRISM Model and Properties

```
// ----------------------------------------------------------------
// Spectrum.prism
// A model for mobile cellular networks applying spectrum renting.
//
// The model is described in
//
//   Tien v. Do, Patrick Wüchner, Tamas Berczes, Janos Sztrik,
//   Hermann de Meer: A New Finite-Source Queueing Model for
//   Mobile Cellular Networks Applying Spectrum Renting,
//   September 2012.
//
// Use for fastest checking the "Sparse" engine and the "Gauss-Seidel" solver;
// for larger K, it may be necessary to increase the CUDD maximum memory size
// to more than 1 GB, otherwise model construction fails.
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// Copyright (C) 2013, Research Institute for Symbolic Computation
// Johannes Kepler University, Linz, Austria, http://www.risc.jku.at
// ----------------------------------------------------------------

// continuous time markov chain (ctmc) model
ctmc

// ----------------------------------------------------------------
// system parameters
// ----------------------------------------------------------------

// renting tresholds
const int t1;      // block renting treshold
const int t2 = 6;  // block release treshold

// bounds
const int K = 100;   // population size
const int r = 8;     // number of servers/channels per block
const int m = 5;     // maximum number of blocks that can be rented
const int n = 2*r;   // minimum number of servers/channels
const int M = n+r*m; // maximum number of simultaneous calls

// rates
const double rho;                  // normalized traffic intensity
const double mu     = 1/53.22;     // service rate
const double lambda = rho*n*mu/K;  // call generation rate
const double nu     = 1;           // retrial rate
const double eta    = 1/300;       // rate of queueing users getting impatient
const double lam_r  = 1/5;         // block renting rate
const double nu_r   = 1/7;         // block rental retrial rate
const double mu_r   = 1;           // block release reate

// probabilities
const double p_b  = 0.1;       // prob. that user gives up (-> sources)
const double p_q  = 0.5;       // prob. that user presses button (-> queue)
const double p_o  = 1-p_b-p_q; // prob. that user retries later (-> orbit)
const double p_io = 0.8;       // prob. that impatient user retries later (-> orbit)
const double p_is = 1-p_io;    // prob. that impantient user gives up (-> sources)
const double p_r  = 0.8;       // block rental success probability
const double p_f  = 1-p_r;     // block rental failure probability

// ----------------------------------------------------------------
// system model
// note that the order of the modules influences model checking time
// heuristically, this seems to be the best one
// ----------------------------------------------------------------

// number of currently available servers/channels
formula servAvail = n+blocks*r;

// blocks are rented at rate lam_r and released at rate mu_r
// renting is successful with probability p_r and fails with probability p_f
// retrying a failed attempt is performed at rate nu_r
module Blocks
  blocks: [0..m] init 0;
  trial: [0..1] init 0;
  [success1] trial = 0 & servAvail-servers <= t1 & blocks < m & queue = 0 ->
    lam_r*p_r: (blocks' = blocks+1);
  [success2] trial = 0 & servAvail-servers <= t1 & blocks < m ->
    lam_r*p_r: (blocks' = blocks+1);
  [failure] trial = 0 & servAvail-servers <= t1 & blocks < m ->
    lam_r*p_f: (trial' = 1);
  [retrial1] trial = 1 & servAvail-servers <= t1 & blocks < m & queue = 0 ->
    nu_r*p_r : (trial' = 0) & (blocks' = blocks+1) ;
  [retrial2] trial = 1 & servAvail-servers <= t1 & blocks < m ->
    nu_r*p_r : (trial' = 0) & (blocks' = blocks+1) ;
  [interrupt] trial = 1 & servAvail-servers > t1  ->
    9999 : (trial' = 0); // "immediately"
  [release] servAvail-servers >= t2+r & blocks > 0 ->
    mu_r : (blocks' = blocks-1);
endmodule

// available servers accept requests
module Servers
  servers: [0..M] init 0;
  [sservers] servers < servAvail -> (servers' = servers+1);
  [oservers] servers < servAvail -> (servers' = servers+1);
  [success2] servers < M         -> (servers' = servers+1);
  [retrial2] servers < M         -> (servers' = servers+1);
  [ssources1] servers > 0 & queue = 0 ->
    servers*mu : (servers' = servers-1);
  [ssources2] servers > 0 ->
     servers*mu : true ;
endmodule

// generate requests at rate sources*lambda
module Sources
  sources: [0..K] init K;
  [sservers] sources > 0 ->
    sources*lambda : (sources' = sources-1);
  [sorbit]   sources > 0 & servers = servAvail ->
    sources*lambda*p_o : (sources' = sources-1);
  [squeue]   sources > 0 & servers = servAvail ->
    sources*lambda*p_q : (sources' = sources-1);
  [ssources1] sources < K -> (sources' = sources+1);
  [ssources2] sources < K -> (sources' = sources+1);
  [qsources]  sources < K -> (sources' = sources+1);
  [osources]  sources < K -> (sources' = sources+1);
endmodule

// if no server is available, requests are redirected
// with probability p_o to the orbit
formula orbit = K-(sources+servers+queue); // make variable virtual
module Orbit
  // orbit: [0..K-n] init 0;
  [sorbit]   orbit < K-n -> true;
  [qorbit]   orbit < K-n -> true;
  [oservers] orbit > 0 -> orbit*nu : true;
  [oqueue]   orbit > 0 & servers = servAvail -> orbit*nu*p_q : true;
  [osources] orbit > 0 & servers = servAvail -> orbit*nu*p_b : true;
endmodule

// if no server is available, requests are redirected
// with probability p_q to the queue
module Queue
  queue: [0..K-n] init 0;
  [squeue]    queue < K-n -> (queue' = queue+1);
  [oqueue]    queue < K-n -> (queue' = queue+1);
  [qorbit]    queue > 0 & servers = servAvail ->
    queue*eta*p_io : (queue' = queue-1);
  [qsources]  queue > 0 & servers = servAvail ->
    queue*eta*p_is : (queue' = queue-1);
  [ssources2] queue > 0 -> (queue' = queue-1);
  [success2]  queue > 0 -> (queue' = queue-1);
  [retrial2]  queue > 0 -> (queue' = queue-1);
endmodule

// -------------------------------------------------------------------
// system rewards
// -------------------------------------------------------------------

// mean number of active requests
rewards "mM"
  true : max(0, orbit)+queue+servers;
endrewards

// mean number of calls in orbit
rewards "mO"
  true : max(0, orbit);
endrewards

// mean number of calls in queue
rewards "mQ"
  true: queue;
```

```
endrewards

// mean number of active calls
rewards "mC"
  true: servers;
endrewards

// mean number of active blocks
rewards "mB"
  true: blocks;
endrewards


// ------------------------------------------------------------------
// Spectrum.props
// ------------------------------------------------------------------

// mean number of active requests
"mM" : R{"mM"}=? [ S ] ;

// mean number of active sources
"mK" : K-"mM" ;

// mean throughput (served and unserved)
"m1" : "mK"*lambda ;

// mean number of active calls
"mC" : R{"mC"}=? [ S ] ;

// mean goodput
"m1good" : "mC"*mu ;

// probability that arriving customer gets served
"Pgood" : "m1good"/"m1" ;

// mean response time (served and unserved)
"mT" : "mM"/"m1" ;

// mean number of rented blocks
"mB" : R{"mB"}=? [ S ] ;

// mean number of available servers
"mS" : n+"mB"*r ;

// mean number of idle servers
"mAS" : "mS"-"mC" ;

// utilization of available servers
"Sutil" : "mC"/"mS" ;

// blocking probability
"Pblock" : S=? [ servers = servAvail ] ;

const int B;
```

```
// probability that B blocks are partially utilized
"Pb" : S=? [ n+r*(B-1) < servers & servers <= n+r*B ] ;

// mean queue length
"mQ" : R{"mQ"}=? [ S ] ;

// mean time spent in queue
"mTQ" : "mQ" / "m1" ;

// mean orbit length
"mO" : R{"mO"}=? [ S ] ;

// mean time spent in orbit
"mTO" : "mO" / "m1" ;
```

B. The Parallel Execution Script

```
#
# Perform in parallel on PROC processors the PRISM analysis of
# various properties with numerous parameter combinations.
#
#!/bin/sh

# the number of processes to be used
PROC=32

# the program locations
export PRISM_JAVA="prism-java"
PRISM="prism"
PARALLEL="./parallel"
TIME="time"

# the input/output locations
MODELFILE="Spectrum.prism"
PROPSFILE="Spectrum.props"
RESULTDIR="Results"
LOGDIR="Logfiles"
LOGFILE="LOGFILE"

# the checker settings
PRISMOPTIONS="-sparse -gaussseidel -nocompact"

(

# the properties to be checked and the parameters for the experiment
for PROPERTY in Pblock mO mTO mB mQ mTQ mC mAS ; do
  for RHO in $(seq 0.6 0.5 4.6) ; do
    for T1 in $(seq 1 1 4) ; do
      echo "$PRISM $PRISMOPTIONS $MODELFILE $PROPSFILE -prop $PROPERTY \
        -const rho=$RHO,t1=$T1 \
        -exportresults $RESULTDIR/$PROPERTY-$T1-$RHO \
         > $LOGDIR/$PROPERTY-$T1-$RHO"
    done
  done
done

# execute the experiments in parallel with PROC processes
) | $TIME -p $PARALLEL $PROC > $LOGDIR/$LOGFILE 2>&1
```

C. The PRISM Java Configuration

```
#
# prism-java
# Execute on a Non-Uniform Memory Architecture (NUMA) all threads of a Java
# program on the same random node with all memory allocated on that node.
#
# Assumes that we have four nodes (processors) available.
#
#!/bin/sh
t=`date +%N`
n=`expr $t % 4`
numactl --cpunodebind=+$n --membind=+$n java \
  -XX:+UseParallelOldGC -XX:ParallelGCThreads=4 -XX:+AggressiveOpts $*
```

D. The Parallel Execution Framework

```
// ---------------------------------------------------------------------------
// parallel.c
// A program that reads command lines and executes them by a given number
// of processes in parallel.
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
// Copyright (C) 2013, Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria, http://www.risc.jku.at
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
// --------------------------------------------------------------------------

#define _POSIX_C_SOURCE 199309L

#include <stdio.h>
```

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

// ---------------------------------------------------------------------------
// the data
// ---------------------------------------------------------------------------

// a task description to be processed by a thread
typedef struct
{
  pthread_t       thread;  // the thread executing the task
  pthread_mutex_t mutex;   // the mutex associated to the thread
  pthread_cond_t  cond;    // the condition associated to the thread
  int             counter; // a unique identifier for the task
  char*           command; // the command to be executed
  uint64_t        time;    // the time for executing the command in ns
  bool            ready;   // indication that thread is ready to accept task
} task_t;

// the result delivered by a task
static task_t *taskResult;
static pthread_mutex_t resultMutex;
static pthread_cond_t fullCond;
static pthread_cond_t emptyCond;

// a command to be executed
typedef struct command_t
{
  char*  line;
  struct command_t* next;
} command_t;

// the queue of commands to be executed
static command_t* commandsHead;
static command_t* commandsTail;
static int commandsNumber;
static pthread_cond_t commandsCond;
static pthread_mutex_t commandsMutex;

// a buffer for reading lines of text
static char* buffer;
static int bufferLength;
static int bufferPos;

// true if eof encountered
static bool eof;

// initialize the data
static void initialize(void)
{
  pthread_mutex_init(&resultMutex, NULL);
  pthread_cond_init(&fullCond, NULL);
  pthread_cond_init(&emptyCond, NULL);
  commandsHead = NULL;
  commandsTail = NULL;
  commandsNumber = 0;
  pthread_cond_init(&commandsCond, NULL);
  pthread_mutex_init(&commandsMutex, NULL);
  bufferLength = 512;
  bufferPos = 0;
  buffer = (char*)malloc(bufferLength*sizeof(char));
  eof = false;
}

// finalize the data
static void finalize()
{
  free(buffer);
}

// ---------------------------------------------------------------------------
// the main program
// ---------------------------------------------------------------------------

static void* read(void *arg);
static void* execute(void *command);
static char* getLine(void);
static uint64_t timespecDiff(struct timespec *timeA_p,
  struct timespec *timeB_p);

static void usage(void)
{
  printf(
    "Usage: parallel <PROC>\n"
    "  PROC ... the number of parallel processes (>=1)\n\n"
    "  This command reads from its standard input (also from a pipe)\n"
    "  arbitrarily (also infinitely) many command lines and executes\n"
    "  them by the denoted number of processes in parallel.\n\n"
    "  Author: Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>\n"
    "  Copyright (C) 2013, Research Institute for Symbolic Computation (RISC)\n"
    "  Johannes Kepler University, Linz, Austria, http://www.risc.jku.at\n\n"
    "  This program is free software under the terms of the\n"
    "  GNU General Public License (GPL) version 3 or later.\n\n"
    "  The source code of this program can be downloaded from\n"
    "  http://www.risc.jku.at/people/schreine/parallel/parallel.tgz\n");
  exit(-1);
}

// the main function which manages the tasks
int main(int argc, char** argv)
{
  // process arguments
  if (argc != 2) usage();
  int taskLength = atoi(argv[1]);
```

```
if (taskLength < 1) usage();

// initialize data
initialize();

// create thread for handling input (terminates if there is no more input)
pthread_t rthread;
int result = pthread_create(&rthread, NULL, read, NULL);
if (result != 0)
{
  printf("could not create read thread (%d)\n", result);
  exit(-1);
}

// create worker threads
task_t* tasks = (task_t*)malloc(taskLength*sizeof(task_t));
int taskNumber = 0;
int taskCounter = 0;
int openNumber = 0;
bool done = false;
int i;
for (i=0; i<taskLength; i++)
{
  pthread_mutex_init(&tasks[i].mutex, NULL);
  pthread_cond_init(&tasks[i].cond, NULL);
  char* command = getLine();
  done = command == NULL;
  if (done) break;
  tasks[i].counter = taskCounter;
  tasks[i].command = command;
  tasks[i].ready = false;
  taskCounter++; taskNumber++; openNumber++;
  result = pthread_create(&tasks[i].thread, NULL, execute, tasks+i);
  if (result != 0)
  {
    printf("could not create worker thread (%d)\n", result);
    exit(-1);
  }
}

// process tasks
while (openNumber > 0)
{
  // wait for any result
  pthread_mutex_lock(&resultMutex);
  while (taskResult == NULL)
    pthread_cond_wait(&fullCond, &resultMutex);
  task_t* tresult = taskResult;
  taskResult = NULL;
  pthread_cond_signal(&emptyCond);
  pthread_mutex_unlock(&resultMutex);
  openNumber--;

  // print result information
```

```
    printf("\n");
    printf("task:      %d\n", tresult->counter);
    printf("command:   %s\n", tresult->command);
    printf("time (ms): %lu\n", tresult->time/1000000);
    printf("executing: %d\n", openNumber);
    printf("queued:    %d\n", commandsNumber);
    free(tresult->command);
    tresult->command = NULL;

    // get new task
    if (done) continue;
    char* command = getLine();
    done = command == NULL;
    if (done) continue;

    // let thread execute the new task
    pthread_mutex_lock(&tresult->mutex);
    tresult->counter = taskCounter;
    tresult->command = command;
    tresult->ready = false;
    pthread_cond_signal(&tresult->cond);
    pthread_mutex_unlock(&tresult->mutex);
    taskCounter++; openNumber++;
  }

  // terminate threads
  for (i=0; i<taskNumber; i++)
  {
    result = pthread_cancel(tasks[i].thread);
    if (result != 0)
    {
      printf("could not cancel thread (%d)\n", result);
      exit(-1);
    }
    result = pthread_join(tasks[i].thread, NULL);
    if (result != 0)
    {
      printf("could not join canceled thread (%d)\n", result);
      exit(-1);
    }
  }
  result = pthread_join(rthread, NULL);
  if (result != 0)
  {
    printf("could not join reader thread (%d)\n", result);
    exit(-1);
  }

  // finalize data
  finalize();
  return 0;
}

// forever execute assigned commands
```

```
static void* execute(void *arg)
{
  task_t* task = (task_t*)arg;
  while(true)
  {
    // execute the assigned command
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    int result = system(task->command);
    clock_gettime(CLOCK_MONOTONIC, &end);
    task->time = timespecDiff(&end, &start);
    if (result == -1)
    {
      printf("could not execute: %s\n", task->command);
      exit(-1);
    }

    // deliver result
    pthread_mutex_lock(&resultMutex);
    while (taskResult != NULL)
      pthread_cond_wait(&emptyCond, &resultMutex);
    taskResult = task;
    pthread_cond_signal(&fullCond);
    pthread_mutex_unlock(&resultMutex);

    // get next assignment
    pthread_mutex_lock(&task->mutex);
    task->ready = true;
    do
      pthread_cond_wait(&task->cond, &task->mutex);
    while (task->ready);
    pthread_mutex_unlock(&task->mutex);
  }
}

// ----------------------------------------------------------------------------
// time handling
// ----------------------------------------------------------------------------

// compute the difference of times in nanoseconds
static uint64_t timespecDiff(struct timespec *timeA_p, struct timespec *timeB_p)
{
  return (timeA_p->tv_sec * 1000000000 + timeA_p->tv_nsec) -
         (timeB_p->tv_sec * 1000000000 + timeB_p->tv_nsec);
}

// ----------------------------------------------------------------------------
// input handling
// ----------------------------------------------------------------------------

static char* readLine(void);
static void addLine(char* line);
static void noMoreLine(void);
```

```
// read input into command buffer
static void* read(void *arg __attribute__((unused)))
{
  while (true)
  {
    char* line = readLine();
    if (line == NULL) break;
    addLine(line);
  }
  noMoreLine();
  return NULL;
}

// add line to commands queue
static void addLine(char* line)
{
  command_t* command = (command_t*)malloc(sizeof(command_t));
  command->line = line;
  command->next = NULL;
  pthread_mutex_lock(&commandsMutex);
  if (commandsTail == NULL)
    commandsHead = command;
  else
    commandsTail->next = command;
  commandsTail = command;
  commandsNumber++;
  pthread_cond_signal(&commandsCond);
  pthread_mutex_unlock(&commandsMutex);
}

// indicate that there is no more line
static void noMoreLine(void)
{
  pthread_mutex_lock(&commandsMutex);
  pthread_cond_signal(&commandsCond);
  pthread_mutex_unlock(&commandsMutex);
}

// get line from commands array
static char* getLine(void)
{
  pthread_mutex_lock(&commandsMutex);
  while (commandsHead == NULL)
  {
    if (eof)
    {
      pthread_mutex_unlock(&commandsMutex);
      return NULL;
    }
    pthread_cond_wait(&commandsCond, &commandsMutex);
  }
  char* result = commandsHead->line;
  command_t* head = commandsHead;
  commandsHead = commandsHead->next;
  free(head);
  if (commandsHead == NULL) commandsTail = NULL;
  commandsNumber--;
  pthread_mutex_unlock(&commandsMutex);
  return result;
}

static void addChar(char ch);

// read line from standard input and return it
static char* readLine(void)
{
  if (eof) return NULL;
  while (true)
  {
    int ch = getchar();
    if (ch == EOF)
    {
      eof = true;
      if (bufferPos == 0) return NULL;
      break;
    }
    if (ch == '\n') break;
    addChar((char)ch);
  }
  char *line = (char*)malloc((bufferPos+1)*sizeof(char));
  memcpy(line, buffer, bufferPos*sizeof(char));
  line[bufferPos] = 0;
  bufferPos = 0;
  return line;
}

// add character to buffer
static void addChar(char ch)
{
  if (bufferPos == bufferLength)
  {
    int bufferLength0 = 2*bufferLength;
    char *buffer0 = (char*)malloc(bufferLength0*sizeof(char));
    memcpy(buffer0, buffer, bufferLength*sizeof(char));
    free(buffer);
    buffer = buffer0;
    bufferLength = bufferLength0;
  }
  buffer[bufferPos] = ch;
  bufferPos++;
}

// ----------------------------------------------------------------------------
// end of file
// ----------------------------------------------------------------------------
```

E. The Data Combination Script

```
#!/bin/sh

# the input/output locations
RESULTDIR="Results"

# combine the individual results for each property
for PROPERTY in Pblock mO mTO mB mQ mTQ mC mAS ; do
  rm -f $RESULTDIR/$PROPERTY
  for RHO in $(seq 0.6 0.5 4.6) ; do
    echo -n "$RHO " >> $RESULTDIR/$PROPERTY
    for T1 in $(seq 1 1 4) ; do
      VALUE=`tail -1 $RESULTDIR/$PROPERTY-$T1-$RHO`
      echo -n "$VALUE " >> $RESULTDIR/$PROPERTY
    done
    echo >> $RESULTDIR/$PROPERTY
  done
done
```

F. The Figure Generation Script

```
# file locations
PROPERTY="Pblock"
RESULTDIR="./Results"
FIGUREDIR="./Figures"

for PROPERTY in Pblock mO mTO mQ mTQ ; do

gnuplot << EOF

# figure parameters
set terminal fig textspecial
set output "$FIGUREDIR/$PROPERTY.fig"
set xlabel '$\rho$'
set ylabel "$PROPERTY"
set logscale y
set tics out
set xtics (0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5)
set ytics (0.0001,0.001,0.01,0.1)
set key right bottom Left reverse sample 0

# plotting command
plot [0.5:4.7] [0.0001:0.1]\
  "$RESULTDIR/$PROPERTY" using (\$1):(\$2) title "t1=1" with linesp ls 1, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$3) title "t1=2" with linesp ls 2, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$4) title "t1=3" with linesp ls 3, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$5) title "t1=4" with linesp ls 4

EOF

fig2dev -m 0.5 -L pdftex -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf
```

```
fig2dev -m 0.5 -L pdftex_t -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf_t

done

PROPERTY=mB

gnuplot << EOF

# figure parameters
set terminal fig textspecial
set output "$FIGUREDIR/$PROPERTY.fig"
set xlabel '$\rho$'
set ylabel "$PROPERTY"
set tics out
set xtics (0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5)
set ytics (0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5)
set key right bottom Left reverse sample 0

# plotting command
plot [0.5:4.7] [0.5:4.7]\
  "$RESULTDIR/$PROPERTY" using (\$1):(\$2) title "t1=1" with linesp ls 1, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$3) title "t1=2" with linesp ls 2, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$4) title "t1=3" with linesp ls 3, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$5) title "t1=4" with linesp ls 4

EOF

fig2dev -m 0.5 -L pdftex -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf
fig2dev -m 0.5 -L pdftex_t -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf_t

PROPERTY=mAS

gnuplot << EOF

# figure parameters
set terminal fig textspecial
set output "$FIGUREDIR/$PROPERTY.fig"
set xlabel '$\rho$'
set ylabel "$PROPERTY"
set tics out
set xtics (0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5)
set key right bottom Left reverse sample 0

# plotting command
plot [0.5:4.7] [5:10]\
  "$RESULTDIR/$PROPERTY" using (\$1):(\$2) title "t1=1" with linesp ls 1, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$3) title "t1=2" with linesp ls 2, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$4) title "t1=3" with linesp ls 3, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$5) title "t1=4" with linesp ls 4

EOF

fig2dev -m 0.5 -L pdftex -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf
fig2dev -m 0.5 -L pdftex_t -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf_t
```

```
PROPERTY=mC

gnuplot << EOF

# figure parameters
set terminal fig textspecial
set output "$FIGUREDIR/$PROPERTY.fig"
set xlabel '$\rho$'
set ylabel "$PROPERTY"
set tics out
set xtics (0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5)
set key right bottom Left reverse sample 0

# plotting command
plot [0.5:4.7] [15:55]\
  "$RESULTDIR/$PROPERTY" using (\$1):(\$2) title "t1=1" with linesp ls 1, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$3) title "t1=2" with linesp ls 2, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$4) title "t1=3" with linesp ls 3, \
  "$RESULTDIR/$PROPERTY" using (\$1):(\$5) title "t1=4" with linesp ls 4

EOF

fig2dev -m 0.5 -L pdftex -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf
fig2dev -m 0.5 -L pdftex_t -p $PROPERTY $FIGUREDIR/$PROPERTY.fig $FIGUREDIR/$PROPERTY.pdf_t
```