Parallel Simulation of Probabilistic P Systems on Multicore Platforms

Miguel A. Martínez-del-Amor[1], Ian Karlin[2], Rune E. Jensen[2], Mario J. Pérez-Jiménez[1], Anne C. Elster[2]

[1] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: `mdelamor@us.es, marper@us.es`

[2] High Performance/Heterogeneous and Parallel Computing Lab
Department of Computer and Information Science
Norwegian University of Science and Technology
Sem Sælands vei 9, NO-7491, Trondheim, Norway
E-mail: `Ian.Karlin@colorado.edu, runeerle@idi.ntnu.no, elster@idi.ntnu.no`

Summary. Ecologists need to model ecosystems to predict how they will evolve over time. Since ecosystems are non-deterministic phenomena, they must express the likelihood of events occurring, and measure the uncertainty of their models' predictions. One method well suited to these demands is Population Dynamic P systems (PDP systems, in short), which is a formal framework based on multienvironment probabilistic P systems. In this paper, we show how to parallelize a Population Dynamics P system simulator, used to model biological systems, on multi-core processors, such as the Intel i5 Nehalem and i7 Sandy Bridge. A comparison of three different techniques, discuss their strengths and weaknesses, and evaluate their performance on two generations of Intel processors with large memory sub-system differences is presented. We show that P systems are memory bound computations and future performance optimization efforts should focus on memory traffic reductions. We achieve runtime gains of up to 2.5x by using all the cores of a single socket 4-core Intel i7 built on the Sandy Bridge architecture. From our analysis of these results we identify further ways to improve the runtime of our simulator.

Key words: Population Dynamics, P systems, Parallel Simulation, Multicore Computing, OpenMP

1 Introduction

Multienvironment probabilistic P systems are used to model species in real ecosystems, such as the bearded vulture in the Catalan Pyrenees [4] and zebra mussels in the Ribarroja reservoir [3]. They conform a formal framework for ecological modelling called Population Dynamics P systems. These models are first validated by a software tool, and can reproduce actual measurements taken in a given number of years [4]. The goal of work is to be able to use P systems simulations to adopt *a priori* management strategies for the real system. However, P systems are computationally and data expensive with large systems, such as the one modelling the zebra mussel ecosystem, taking hours to run on a single set of input parameters on a single core processor.

Due to the probabilistic behaviour of these systems, ecological experts and model designers run many simulations on each set of input parameters to extract statistical information of the likelihood of certain behaviours occurring [4]. This makes the systems large. The more simulations run, the more confident they can be in the model's output. Also, the more input parameters they test, the greater certainty that the real-life experiments they run will yield useful knowledge. Therefore, the overall runtime of the simulations is critical.

This paper describes our initial parallelization work, which includes implementing a C/C++ version of the DCBA algorithm [7]. We have designed an implementation which saves on memory by avoiding the creation of a static table. We also choose C for its similarity to the common GPGPU (General Purpose computations on Graphics Processing Units) languages of OpenCL and CUDA, and the support of many parallel libraries. Our new implementation is parallelized in three ways: 1) simulations, 2) environments and 3) a hybrid approach. All them are implemented using the parallel standard library for multicore platforms, OpenMP [1]. From our analysis of these results we identify further ways to improve the runtime of our simulator, by minimizing memory and cache bottlenecks using data compression and GPU computing. Ideas and references on compression and GPU computing can be found in [2].

The rest of the paper is organized as follows: Section 2 gives an overview of the P systems framework that includes our simulator. Section 3 discusses the three forms of parallelism we introduced into the simulator and the advantages and disadvantages of each. Section 4 contains experimental results from testing out initial parallelization efforts. Finally, conclusions and future work are presented in Section 5.

2 System overview

The simulator we are optimizing and parallelizing is included in a P systems based modelling framework that contains four design levels. An overview of the framework is shown in Figure 1 along with the domain specific knowledge needed to implement each level. At the top level, ecological experts express living systems as inputs based upon observed data and/or conditions they wish to test. Example inputs include the number of organisms living in the ecosystem, temperature data and the probabilities of events happening. These inputs are then fed into probabilistic P systems with the assistance of the model designers, which provide the rules used by the model.

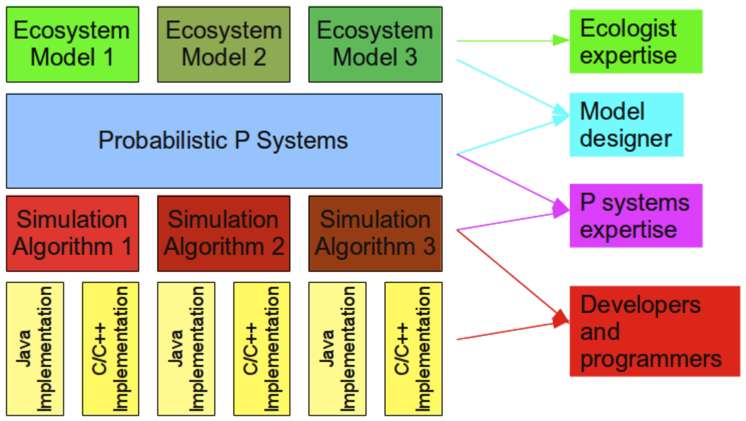


Fig. 1. Framework levels and expertise needed

The probabilistic P system software level currently uses a framework called PLinguaCore [6]. PLinguaCore is written in Java, and contains implementations of different types of P systems, along with a standard specification language for P systems known as P-Lingua. Both ecologists and model designers run their simulations using this framework through a software tool called MeCoSim [9] that provides an abstraction layer for non-experts in P system models.

The execution of the rules input by the model designer occurs in the simulation algorithms, which are implemented in common programming languages. Fast execution of the algorithms is important to ecological experts and model designers. Also, ecological experts often simulate many hypotheses before deciding which hypotheses to experimentally evaluate. The simulation algorithm and implementation levels are the most time consuming parts of the simulations, and thus, where we focus on improving performance.

Algorithm 1 Main loop of the simulator

```
1:  for $sim \leftarrow 0, \ldots, simulations$ do
2:    INITIALIZATION
3:    for $step \leftarrow 0, \ldots, time\_steps$ do
4:      for $env \leftarrow 0, \ldots, environments$ do
5:        SELECTION OF RULES
6:      end for
7:      for $env \leftarrow 0, \ldots, environments$ do
8:        EXECUTION OF RULES
9:      end for
10:   end for
11: end for
```

The last simulation algorithm for PDP systems is the DCBA algorithm [7]. A high level algorithm representation of our simulator is shown in Algorithm 1. The outermost loop runs the simulation multiple times as specified by the user. The next loop performs the discrete time steps that advance the simulation. The time steps are performed in two stages: selection and execution. During the first stage (selection) all environments, which each represent discrete parts of an ecosystem with different properties, have rules selected to be executed on them. All environments have the same rules, but the associated probability indicating the likelihood of them being executed varies between environments. In the second stage (execution) of a time step, the rules are executed. During a time step, environments evolve independently, but between time steps can communicate objects.

Rules are classified into rule blocks by their left-hand sides (consuming the multisets of objects in the same compartments, and according to the same charge of the active membrane) and the charge of the active membrane in the right-hand side (consistent blocks [7]). The probabilities of the rules within a block sum 1 (they have local meaning inside the blocks).

Selection stage is split into three micro-phases (see [7] for more details): phase 1 (object distribution), phase 2 (maximality) and phase 3 (probabilities). The DCBA algorithm uses a table for phase 1 in order to distributes the objects along the rule blocks. This table has one column per each rule block, and one row per each pair object and membrane (also considering the environment itself). Therefore, the size of the table is of order $O(|B| \cdot |\Gamma| \cdot (q+1))$, being $|B|$ the number of rule blocks, $|\Gamma|$ the size of the alphabet (total amount of different objects), and $q+1$ the number of membranes in the system plus the space for the environment in each one.

The implementation of this table can be inefficient in systems with a large number of rule blocks and/or objects. Therefore, our simulator does not really implement the table. The main baseline idea is to translate operations over the table to operations directly to the rule blocks:

- Operations over columns: they can be transformed to operations over the rule blocks and their left-hand sides (LHS in short).
- Operations over rows: they can be transformed to operations over the left-hand sides of rule blocks and storing the partial result in a global variable for each row.

Phase 1 can be implemented as shown in Algorithm 2. Phase 2, phase 3 and execution stage can be directly implemented following the corresponding definitions in [7].

Remark that in this implementation, instead of using a real table, we *virtually* implement it by using operations over the information of the rule blocks. Actually, two extra vectors are only used:

- *Activation vector*: We annotate the blocks that has not been filtered by a boolean value.
- *Addition vector*: We add the values of the rows by using this global vector, one per each pair object and membrane.

Algorithm 2 Selection Phase 1 (Distribution)

```
1: Apply *Filter 1*: for each block, if the charge in the LHS is different to the one presented
   in the configuration, then deactivate the block: $activationVector[b] = false$.
2: Apply *Filter 2*: for each block, if one of the objects involved in the left-hand side
   does not exist in $C_t$, then deactivate the block: $activationVector[b] = false$.
3: Check the mutually consistency of blocks.
4: **repeat**
5:    For each active block, and for each object in the left-hand side, add the multi-
      plicity $k$ appearing in the block to a global variable for the corresponding object
      ($addition[object, membrane] += k$).
6:    For each active block, calculate the minimum of the object distributions in the
      left-hand side: $N_b = Min_{[o^k]_m \in LHS(block)}(\frac{1}{k^2} * \frac{1}{addition[o,m]} * C[o,m])$. This is the
      number of applications for block $b$: $NumAppBlocks[b] += N_b$.
7:    Delete objects in the configuration $C$, corresponding with $N_b$.
8:    Apply *Filter 2*.
9:    $a = a + 1$
10: **until** $a == A$ **or** *every* $N_b == 0$
```

3 Design and parallelism

Before we introduced parallelism, we first rewrote the simulator in C/C++ which is advantageous because OpenMP, PThreads and MPI all are supported. In this section, we describe the implementation of the three forms of parallelism added to our simulator. A discussion of the advantages and disadvantages of each is included.

Simulations are parallelized by using the *#pragma omp parallel for* OpenMP directive on the simulation loop from Algorithm 1. The advantage of running simulations in parallel is there are no data dependencies between simulations, and, therefore, the problem is embarrassingly parallel. Also, the users of our simulator typically run 50 to 100 simulations of each set of input parameters, so there are enough simulations to consume all cores. However, there are disadvantages of running simulations in parallel. Each simulation needs its own memory space increasing the amount of memory used. If the number of simulations is not divisible by the number of processors then load balancing issues can occur with the final simulations running while some cores are idle. Also, running simulations in parallel can result in resource conflicts as cores compete for shared resources.

Environments are parallelized by using the *#pragma omp parallel* to generate a thread pool for the simulation. Then the for loops in Algorithm 1 that iterate over environments are parallelized with *#pragma omp for*, which has an implicit barrier that enforces the dependencies between the stages in each time step. Using this design, creating new thread blocks for each for loop is avoided.

The advantage of parallelizing environments over simulations is that memory usage does not increase. However, dependencies occur twice in each time step requiring synchronization steps. Also, since most models use 5 to 30 environments, there are cases where modern machines have more cores than environments and just parallelizing environments cannot take advantage of all computing resources. In addition, as with simulations, load balancing can be an issue if the number of environments is not divisible by the number of cores, or if the runtime of environments varies.

Hybrid parallelization is accomplished by combining parallel environments with parallel simulations. We accomplish hybrid parallelization through command-line flags that allow the specification of how many environments or simulations to run in parallel. By combining both forms of parallelism, we can balance the amount of each resource used. This will become more important as the number of cores within a node increases. For example, the number of simulations can be increased until available memory is used and then environments within each system can be parallelized.

4 Experimental Evaluation

In this section, we describe a series of tests performed on our implementation and the systems they were run on, along with the results from those experiments.

4.1 Test Environment and Methodology

The following experiments were run on the two machines shown in Table 1. The tests used random systems with similar amounts of data to real-life examples. Multiple configurations with environments and simulations varying from 10 to 50 were tested for the parallel environments, simulations and two hybrid combinations. Each of these tests were run on 1 to 8 cores for the Intel i5 machine, and 1 to 4 for the Intel i7. The measurements in this section, except when noted, correspond only to the parallelized part of the code.

Processor	**Speed**	**Bus speed**	**Cache**
i5 Nehalem (2x4)	2 Ghz	3x800 Mhz	2x4 MB
i7 Sandy Bridge (1x4)	3.4 Ghz	2x1333 Mhz	8 MB

Table 1. Specifications of the test machines.

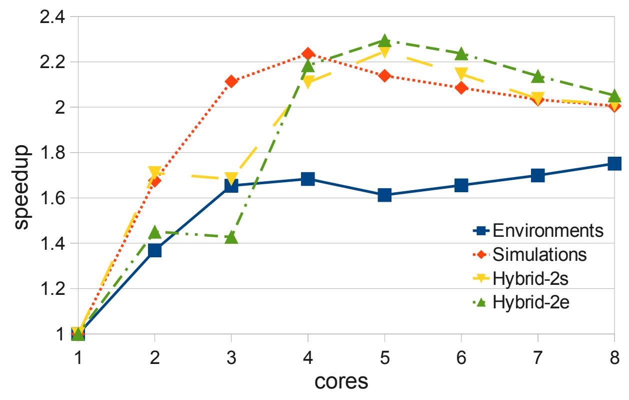
4.2 Results

The serial runtime on both of our test machines is shown in Table 2. *Setup* is the cost of running the serial portion of the code. The other two columns represent the runtime extremes of our test cases when run in serial. From the table, we can see that in serial the setup portion is a small part of the overall runtime, and that the Sandy Bridge processor is about 2.5 times faster than the Nehalem.

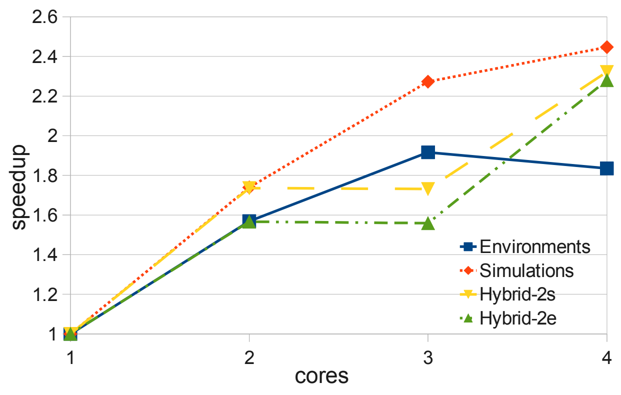
Processor	**Setup**	**10 env & sim**	**50 env & sim**
Nehalem	0.8s	48.0s	251.0s
Sandy Bridge	0.35s	19.9s	97.8s

Table 2. Serial Runtimes

Figures 2 and 3 show the performance improvements of parallelizing our system in various ways. The two figures are representative of the other tests we performed with the best performance either being parallelizing by simulations or the hybrid method (2s), which uses two simulations and then parallelizes by environments. Another trend shown is that as the number of simulations increases, the advantage of parallelizing by simulations increases. The same effect is observed for environments.



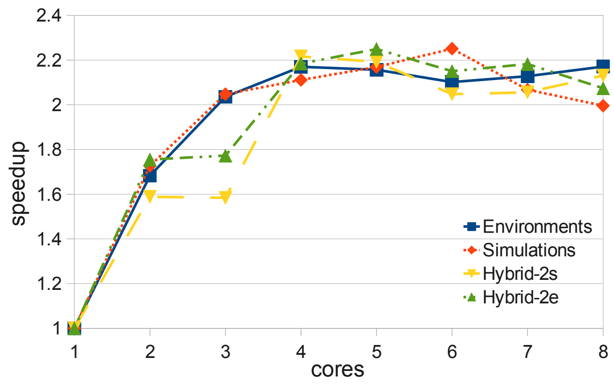
(a) Nehalem



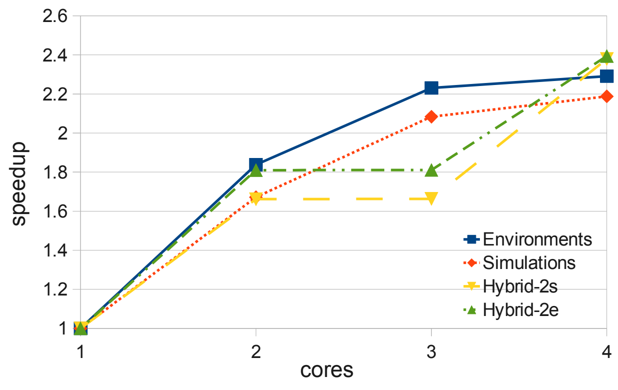
(b) Sandy Bridge

Fig. 2. Speedups running 50 simulations with 10 environments in the system

On the Sandy Bridge system the largest speedup of 2.5x occurs for 50 simulations and 50 environments. However, the maximum speedup on Sandy Bridge



(a) Nehalem



(b) Sandy Bridge

Fig. 3. Speedups running 10 simulations with 50 environments in the system

when going from 3 to 4 processors is only between 0.1 and 0.2, suggesting that the calculation is memory bound for larger core counts. On the Nehalem machine, the maximum parallel speedup was 2.3x for all tests, which is barely greater than the added available bandwidth from using the second socket. These results, led us to suspect we that the Nehalem system's performance was being limited our programming approach. In particular, we did not account for the Non Uniform Memory Access (NUMA) memory subsystem of the two sockets.

One final test was run to see if NUMA was hurting the performance of our code on the Nehalem machine. First one and then two instances of the code was run with 4 threads each (affinity locked to different sockets) on the machine. With two instances a 2x speedup was achieved over the best parallel results from running one instance of the OpenMP version. Confirming this result is that locking all 4 threads to a single socket performance results in a 50% performance increase when compared to locking 2 threads to each socket. For the current tests, however, affinity is controlled by the operating system and performance is similar to when two threads were locked to each socket. These preliminary tests also indicates that the code is memory bound since overall speedups on 8 cores were less than 5x.

5 Conclusions and future work

In this paper, we showed how P systems simulations can take advantage of modern multi-core architectures. Our implementation included three forms of parallelism. Experiments ran to test the simulator indicate the simulations are memory bound and the portion of the code we parallelized consumes over 98% of the runtime in serial. From this initial work we conclude that parallelizing by simulations or hybrid techniques yields the largest speedups. Also, using hardware, such as Intel's Sandy Bridge, that has more memory bandwidth is an easy way for scientists to improve the speed of our simulator. It can also be concluded that performance tuning to decrease data movement is important for P-system simulators.

Future Work

This paper leaves open many research questions that we plan to explore by building on this work. We have experience overlapping communication and computation [8] and compressing data [2] both of which will be especially important on GPUs. We anticipate large speedups from using GPUs due to their increased memory bandwidth and computational capabilities. In addition, we plan to leverage our experience tuning shared memory systems to other optimizations for NUMA processors [5]. Open research questions on NUMA machines include how our hybrid parallel approach will best map to various processor configurations. A hybrid GPU and CPU code will be able to take advantage of all compute resources on a given system. Either an OpenCL or CUDA/C hybrid implementation will be used. While not a high priority because the simulator is usually run on scientists workstations an MPI version would offer large speedups due to simulations being embarrassingly parallel.

As core counts continue to increase, exploiting parallelism within the environments may be profitable. Also, of interest is eliminating or reducing the synchronization required at each time step. While dependencies exist between environments, not all environments depend on all other environments. For an environment to begin executing its next step, all environments from which organisms can migrate into it must be finished. We believe that for simulations with few migration paths between environments, or with a large number of environments, it is important balance workloads better.

Acknowledgments

M.A. Martínez-del-Amor and M.J. Pérez-Jiménez acknowledge the support of "Proyecto de Excelencia con Investigador de Reconocida Valía" of the "Junta de Andalucía" under grant P08-TIC04200, and the support of the project TIN2009-13192 of the "Ministerio de Educación y Ciencia" of Spain, both co-financed by FEDER funds.

References

1. The OpenMP specification. "`http://www.openmp.org`".
2. A. A. Aqrawi and A. C. Elster. Bandwidth reduction through multithreaded compression of seismic images. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1730 –1739, may 2011.
3. M. Cardona, M. Colomer, A. Margalida, A. Palau, I. Pérez-Hurtado, M. Pérez-Jiménez, and D. Sanuy. A computational modeling for real ecosystems based on p systems. *Natural Computing*, 10:39–53, 2011.
4. M. A. Colomer, A. Margalida, D. Sanuy, and M. J. Pérez-Jiménez. A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. *Ecological Modelling*, 222(1):33 – 47, 2011.
5. A. C. Elster and J. C. Meyer. A super-efficient adaptable bit-reversal algorithm for multithreaded architectures. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009.
6. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of p-lingua 2.0. *Lecture Notes in Computer Science*, 5957:264–288, 2010.
7. M. Martínez-del Amor, I. Pérez-Hurtado, M. García-Quismondo, L. Macías-Ramos, L. Valencia-Cabrera, A. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez, M. Colomer, and M. Pérez-Jiménez. Dcba: Simulating population dynamics P systems with proportional object distribution. In *this volume*.
8. T. Natvig and A. C. Elster. Run-time analysis and instrumentation for communication overlap potential. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 42–49, Berlin, Heidelberg, 2010. Springer-Verlag.
9. I. Pérez-Hurtado, L. Valencia-Cabrera, M. J. Pérez-Jiménez, M. A. Colomer, and A. Riscos-Núñez. Mecosim: A general purpose software tool for simulating biological phenomena by means of p systems. In *IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications (BIC-TA 2010)*, volume I, pages 637–643, Changsha, China, 2010. IEEE, Inc.