IT 09 017
Examensarbete 45 hp
Maj 2009

CellMC

An XSLT-based SBML Model Compiler for Cell/BE and IA32

Emmet Caulfield

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

CellMC: An XSLT-based SBML Model Compiler for Cell/BE and IA32

Emmet Caulfield

In-silico simulation of chemical reactions is playing an increasingly important rôle in furthering understanding of cellular biochemistry, simulating in-vivo chemical reactions such as genetic and enzymatic action, and giving rise to the field of computational systems biology.

The Systems Biology Markup Language (SBML) has been defined to provide a standard way to describe models of biochemical reaction networks, and the Stochastic Simulation Algorithm is an effective and popular method for simulating systems with many reacting species, which cannot be simulated using most numerical solution techniques whose time-complexity grows exponentially with number of chemical species.

CellMC, an open source program generator based on XML Stylesheet Language Transformation (XSL-T), is presented and evaluated. It produces a SIMD-ised and parallelised executable realising SSA for any homogeneous biochemical model expressed as SBML. The compiler works on a variety of Cell/BE and IA32 platforms, including a Sony PlayStation 3 cluster. The IA32 executables produced are shown to outperform others in the literature by a considerable multiple, and they, in turn, are outperformed by those on Cell/BE.

Handledare: Andreas Hellander
Ämnesgranskare: Per Lötstedt
Examinator: Anders Jansson
IT 09 017
Tryckt av: Reprocentralen ITC

Acknowledgements

I am profoundly grateful both to my supervisor, Andreas Hellander, for his generous encouragement, genial patience, and boundless enthusiasm; and to Per Lötstedt for his valuable input and cordial advice.

To my parents, Jim and Phil Caulfield, I am deeply indebted for their unstinting love and support.

Contents

1. Introduction

1.1 Background

1.1.1 Limitation of Deterministic Macroscopic Models

Conventional macroscopic quantitative chemical reaction models are systems of deterministic nonlinear ODEs for the concentrations of the compounds of interest. In cellular biochemistry, however, the volume of a cell may be of the order of 10^{-15}l, the number of reacting molecules (*copy number*) is correspondingly small — hundreds or even tens of molecules — and the bulk assumptions of such deterministic macroscopic models fail [1, 15].

Amongst the important phenomena that deterministic models fail to capture are multistability (switching) and stochastic resonance, causing the models to fail to switch or oscillate as they should [5, pp.13–14].

1.1.2 The Chemical Master Equation

On the mesoscale, the *chemical master equation* (CME), a time-dependent difference-differential equation for the probability distribution of the copy numbers of each distinct molecule, or *species*, can be rigorously derived from molecular kinetics — in a "bottom up" fashion — under reasonable assumptions that the system is well-stirred and in thermal equilibrium [9].

Equation 1.1 shows the general form of the chemical master equation for a system of N species and M reactions, R_μ ($\mu \in \{1 \dots M\}$), where: $\boldsymbol{n}$ is an integer population vector of length N of the copy numbers of the species; each reaction has a probability rate constant, c_μ, a vector, $\boldsymbol{\nu}$, of length N that specifies the change in the population for reaction μ (such that, if reaction μ occurs when the population is $\boldsymbol{n}$, the new population will be $\boldsymbol{n} + \boldsymbol{\nu}_\mu$), and a function $\mathrm{h}_\mu(\boldsymbol{n}) : \mathbb{N}^M \to \mathbb{N}$, which gives the number of combinations of the reactant molecules for reaction μ when the system is in state $\boldsymbol{n}$; as usual, t is time; and finally, P is the probability distribution function of interest, $\mathrm{P}(\boldsymbol{n}, t|\boldsymbol{n}_0, t_0)$ being the probability that the system will be in state $\boldsymbol{n}$ at time t given that it is in state $\boldsymbol{n}_0$ at time t_0.

The "combination counting" function, $\mathrm{h}_\mu(\boldsymbol{n})$ is the only term whose meaning is not immediately clear, but it is surprisingly simple: for a reaction $S_1 + S_2 \to S_3$, it is simply the product $n_1 n_2$, where n_1 and n_2 are the copy numbers of S_1 and S_2 respectively, since this is the number of combinations of ways that a S_1 molecule can meet a S_2 molecule.

$$\frac{\partial}{\partial t}\mathrm{P}(\boldsymbol{n}, t|\boldsymbol{n}_0, t_0) = \sum_{\mu=1}^{M} \left(c_\mu \mathrm{h}_\mu(\boldsymbol{n} - \boldsymbol{\nu}_\mu)\mathrm{P}(\boldsymbol{n} - \boldsymbol{\nu}_\mu, t|\boldsymbol{n}_0, t_0) - c_\mu \mathrm{h}_\mu(\boldsymbol{n})\mathrm{P}(\boldsymbol{n}, t|\boldsymbol{n}_0, t_0)\right) \tag{1.1}$$

Numerically, the CME is found to be much more accurate than equivalent macroscopic models, but suffers from the "curse of dimensionality": if there are 61 participating species, as there are in the heat-shock reaction of *E. coli* (§ 3.3), the system of equations is 61-dimensional. Although techniques exist for approximating and reducing the dimensionality of high-dimensional models, they ultimately still suffer from this limitation. [21]

1.1.3 The Stochastic Simulation Algorithm

Rather than trying to solve the CME for a system, Gillespie's *stochastic simulation algorithm* (*SSA*) is a Monte Carlo technique which numerically simulates the underlying Markov process. It is valid for low copy numbers and does not suffer from the curse of dimensionality [8].

In short, SSA is the predominant method of mesoscale stochastic simulation in computational cellular biochemistry; it is important for three reasons:

- Microscale (molecular dynamics, biophysics) simulations are completely intractable for systems of interest at the biochemical level.
- Other mesoscale methods, based on the CME (or approximations to it) suffer to a greater or lesser degree from the curse of dimensionality and can only be applied to relatively simple systems.
- Deterministic macroscale simulations do not capture crucial stochastic phenomena (e.g. the stochastic resonance exhibited by the circadian rhythm model of § 3.4)

SSA is described in detail in Chapter 2.

1.2 Motivation

The desire for speed for SSA simulations is not new, and is motivated by the desire for a better understanding of ever more complex cellular biochemistry, both to further human knowledge, and for applications such as modeling cancer gene expression.

In the search for speed *field-programmable gate arrays* (FPGAs) have been used to implement SSA in the last few years and, although it is a promising idea, it is not clear exactly how well these perform in practice with realistic biochemical models. [20, 26, 31, 28, 30, 29]

Most recently, but after this project was underway, GPUs have been used for the first time to implement SSA, and with considerable success [19].

Originally, the project was intended to be a vehicle for comparing the *Cell/BE* processor used in the *Sony PlayStation 3*, with respect to SSA, to conventional desktop processors — such as the diverse *Intel* and *AMD* families of "x86" processors (collectively referred to here as *IA32*) used in desktop PCs — in order to evaluate how well-suited *Cell/BE* is to SSA.

Obviously, it would not be valid to compare hand-tuned, SIMD-ised code for the *Cell/BE* with a naïve implementation on the PC that did not take advantage of the SSE vector instructions or multiple cores, so an optimised PC implementation was deemed necessary.

As it became clear that the PC code, too, was very fast compared to other SSA implementations, including the popular *StochKit* [17], the project evolved into a dual-platform model compiler using SBML — an XML-based standard for representing chemical models — as input.

CellMC works by transforming an SBML model to `C` via XSL-T, then compiling the `C` code with *gcc*. Although simple in principle, and relatively simple on *IA32*, it is more challenging on *Cell/BE* due to the more involved compilation process and complex binary format. Nevertheless, it would not be difficult to extend *CellMC* to include support for other architectures, such as GPUs and future multicore processors, by adding an additional XSL-T transformation and support for the prescribed build chain.

It is hoped that the release of *CellMC* on *SourceForge*, its speed, and its ease-of-use will encourage the broader computational systems biology community to use it.

CellMC is an open-source project, registered and hosted at *SourceForge.net* — `http://cellmc.sourceforge.net/`

2. The Stochastic Simulation Algorithm

2.1 Basic Algorithm

The *direct method* (DM) and the less efficient *first reaction method* (FRM) were first presented by Gillespie in 1976 [8].

Here, DM and FRM are described in a somewhat unconventional manner that seeks to avoid certain implicit assumptions in their conventional presentations. For example, it is usual to show the timestep computed as $\tau \leftarrow -\ln(\lambda)/\tilde{\mathrm{U}}(0,1)$, where $\tilde{\mathrm{U}}(0,1)$ denotes a value drawn from a uniform distribution on $(0,1)$, but this presupposes the inversion method of generating exponentially distributed numbers (what is wanted) because it is assumed that a cheap source of uniformly, but not exponentially, distributed random numbers is available. Similarly, the conventional presentation tacitly assumes that the computation of reaction propensities and their summation are performed iteratively, an *a priori* assumption that is better avoided if a parallel implementation is intended[1]

If, as before, we assume M reactions involving N chemical species, the state of the system at time t is a vector, $X_i^{(t)} = \{x_1^{(t)}, x_2^{(t)}, \ldots, x_N^{(t)}\} \in \mathbb{N}^N$ where each $x_i^{(t)}$ represents the number of molecules of chemical species i present at time t, beginning with a stipulated state $X^{(0)}$. An $N \times M$ stoichiometry matrix over the integers[2], V, wherein $V_{i\mu}$ is the increase or decrease in copy number of species i due to the occurrence of reaction μ, making the columns $V_{\cdot\mu}$ exactly the $\boldsymbol{\nu}_\mu$ vectors in the CME (1.1).

The reaction propensities may be viewed as a map $\mathrm{h}^* : \mathbb{N}^N \to (\mathbb{R}^+)^M$ from the current state to a vector of non-negative real numbers having one element for each of the M reactions. In essence, this is a vector of all $c_\mu \mathrm{h}_\mu(\boldsymbol{n})$ in the CME (1.1), marked with a "$*$" as a reminder that it is not quite the same, having had c_μ "rolled in".

If the reaction propensities are stochastic, time evolution of $X^{(t)}$ is a continuous time, discrete-space Markov chain, which implies that the interval between reactions is exponentially distributed and thus the probability of a particular reaction occurring in a given interval is Poisson. Gillespie motivates this physically from a hard-spheres model of kinetic chemistry in detail [8, 9].

Now, a single trajectory can be described by Algorithm 1, where we try not to presuppose any optimisations whatever by presenting the *First Reaction Method* [8, pp.419–422] (*FRM*) abstractly.

A random number drawn from an exponential distribution with parameter λ is denoted $\tilde{\mathrm{E}}(\lambda)$, and one drawn from a uniform distribution between α and β is denoted $\tilde{\mathrm{U}}(\alpha, \beta)$.

[1] It turns out that this *is* the way we do it in practice, but it is arguably better not to assume it so soon.

[2] Very often a sparse matrix over $\{-1, 0, 1\}$ in practice

Algorithm 1 First Reaction Method

Begin at time 0: $t \leftarrow 0$
Initialise the population vector: $X^{(0)} \leftarrow \{x_1^{(0)}, x_2^{(0)}, \ldots, x_N^0\}$
while $t < t_{stop}$ **do**
  Pick random timesteps: $\tau_\kappa \leftarrow \tilde{\mathrm{E}}(\mathrm{h}_\kappa^*(X^{(t)})) \qquad \forall \kappa \in \{1 \ldots M\}$
  Identify first reaction: $\exists! \mu : \tau_\mu < \tau_\kappa \; \forall \kappa \neq \mu$
  Update population: $X^{(t+\tau_\mu)} \leftarrow X^{(t)} + \boldsymbol{\nu}_\mu$
  Update timestep: $t \leftarrow t + \tau_\mu$
end while

Although Gillespie does not present it as such, the more usual *Direct Method* of Algorithm 2, which he presents first, may be viewed as optimisation of FRM.

Algorithm 2 Direct Method

Begin at time 0: $t \leftarrow 0$
Initialise the population vector: $X_i^{(0)} = \{x_1^{(0)}, x_2^{(0)}, \ldots, x_N^0\}$
while $t < t_{stop}$ **do**
  Sum propensities: $\lambda \leftarrow \sum_{\mu=1}^{M} \mathrm{h}_\mu^*(X^{(t)})$
  Pick random numbers: $u \leftarrow \tilde{\mathrm{U}}(0, \lambda),\ \tau \leftarrow \tilde{\mathrm{E}}(\lambda)$
  Pick reaction: $\exists! \mu \in \{1 \ldots M\} : \sum_{\kappa=1}^{\mu-1} \mathrm{h}_\kappa^*(X^{(t)}) < u < \sum_{\kappa=1}^{\mu} \mathrm{h}_\kappa^*(X^{(t)})$
  Update population: $X^{(t+\tau)} \leftarrow X^{(t)} + \boldsymbol{\nu}_\mu$
  Update timestep: $t \leftarrow t + \tau$
end while

A more conventional presentation of the direct method, which assumes the availability of a cheap source of uniform random numbers on $(0, 1)$, but not exponentially distributed numbers, and that certain operations are performed iteratively is given in Algorithm 3.

In reality, of course, the implementation is never quite as obtuse as Algorithm 3, which shows every possible iteration. For example, in practice as in Gillespie's original presentation of DM, only those populations actually affected by the executed reaction are updated.

2.2 Variations of Homogeneous SSA

In the conventional direct method, all propensities are recalculated at each timestep, a full summation of these propensities is performed to compute the propensity sum, and a linear search is conducted to determine which reaction occurs given a random number on $(0, \lambda)$. In FRM, each reaction is assigned a putative time, and the reaction occurring soonest is executed, obviating the need for propensity summation, but necessitating a search for the smallest reaction time.

Without fundamentally changing the method, propensity recalculation may be limited to only those reactions involving species which have changed. This is easily achieved with a dependency graph. Similarly, more efficient search strategies than

Algorithm 3 Direct Method

▶ Initialise variables:
$t \leftarrow 0$
for $i \leftarrow 1 : N$ **do**
  $X_i = X_i^{(0)}$
end for
while $t < t_{stop}$ **do**
  ▶ Compute and sum reaction propensities:
  $\lambda \leftarrow 0$
  for $\mu \leftarrow 1 : M$ **do**
    $p_\mu \leftarrow \mathrm{h}_\mu^*(X^{(t)})$
    $\lambda \leftarrow \lambda + p_\mu$
  end for
  ▶ Pick reaction selector and timestep
  $u \leftarrow \lambda \tilde{\mathrm{U}}(0,1)$
  $\tau \leftarrow -\ln(\lambda)/\tilde{\mathrm{U}}(0,1)$
  ▶ Determine which reaction was picked
  $\mu \leftarrow 1$
  while $u > 0$ **do**
    $u \leftarrow u - p_\mu$
    $\mu \leftarrow \mu + 1$
  end while
  ▶ Update populations:
  for $i \leftarrow 1 : N$ **do**
    $X_i^{(t+\tau)} = X_i^{(t)} + V_{i\mu}$
  end for
  ▶ Update timestep:
  $t \leftarrow t + \tau$
end while

a simple linear search can be applied, whether that search is for a timestep or a reaction.

Since all methods only update only those populations affected by a reaction (rather than "adding on zeros"), SSA optimisations in the literature can be classified, along similar lines to McCollum [24], as optimising:

- searching for the executed reaction;
- propensity recalculation; or
- propensity summation.

In addition, each of these optimisations may be implemented either statically or dynamically.

2.2.1 Next Reaction Method

Gibson and Bruck's *Next Reaction Method* (*NRM*) [7] is based on FRM and thus avoids the propensity summation altogether. It attacks the searching problem dynamically by maintaining an indexed priority queue of reactions so that the reaction with the smallest timestep is always first. NRM introduces limiting propensity recalculation to those reactions involving populations affected by the executed reaction via a dependency graph.

2.2.2 Optimised Direct Method

Cao, Li and Petzold's *Optimised Direct Method* (*ODM*) [2] borrows the dependency graph to limit propensity recalculation from NRM, but, being based on DM, must perform propensity summation, which it achieves by adjusting the propensity sum with the few updated propensities rather than performing a full summation. The key feature of ODM, however, is that it dispenses with the expensive dynamic indexed priority queue in favour of statically ordering reactions, based on a profiling run, to speed up linear searching on average.

ODM is particularly well suited to very stiff systems such as HSR (3.3), where the most likely reactions occur first and the average search depth is low. For less stiff systems, ODM offers less advantage over the direct method, and in the extreme where the reaction propensities are equal, or nearly so, none at all.

2.2.3 Sorting Direct Method

The *Sorting Direct Method* (*SDM*) [24] of McCollum *et al.* adopts an approach somewhere between NRM and ODM in that it maintains a linearly searched list, like ODM, but sorts it dynamically, like NRM. It does this incrementally at low-cost (as compared to NRM's indexed priority queue) by simply exchanging each executed reaction with that nearer the head of the list. Thus, the most commonly executed reactions bubble toward the head of the list, reducing the average search depth, but adapting to changes in reaction propensities.

Although there is some overhead associated with maintaining data structures in NRM and SDM, in principle they should be faster than ODM, which uses static ordering based on profiling, since profiling cannot capture large changes in reaction propensities in periodic models, bistable models, or even models that get progressively more or less stiff. However, in testing LDM (below) Li and Petzold found little difference between ODM and SDM [18].

2.2.4 Logarithmic Direct Method

Li and Petzold's *Logarithmic Direct Method* (*LDM*) [18], like all of the other methods except NRM, is a propensity summing method. It attacks the searching problem by maintaining an array of partial sums of propensities as the summation is performed, and conducts a binary search on this array for reaction selection. Thus, the time to

find the next reaction is the $O(\lg M)$, and it is claimed to be insensitive to reaction ordering, thus avoiding the pre-simulation of ODM. Li and Petzold find the method to be significantly ($\approx$20%) faster than ODM or SDM.

Note that, by the same rationale that motivates ODM, it should be advantageous to statically organise the reactions so that the most likely reaction is in the middle of the array and therefore most likely to be found first by binary search, although the gain may be small.

3. Biochemical Model Systems

In order to develop and characterise *CellMC*, 4 popular and representative biochemical models were used: two simple systems, one non-stiff (DD) and one slightly stiff (ME), and two very stiff realistic systems, one exhibiting stochastic resonance (CR), and the other commonly used as a benchmark in the literature (HSR).

In the context of SSA, *stiff* systems are characterised by extreme disparity in their reaction propensities — in other words, some reactions being very much more likely than others. The few very fast reactions require very small timesteps, not needed to capture the dynamics of the slower reactions in the system. Figure 3.1 shows the reactions of HSR in ODM order — Hellander shows an equivalent unordered diagram [11, p.9] — the first six (≈10%) reactions account for ≈95% of all reactions executed, while ≈75% of reactions are almost never executed (< 0.1% of the time).

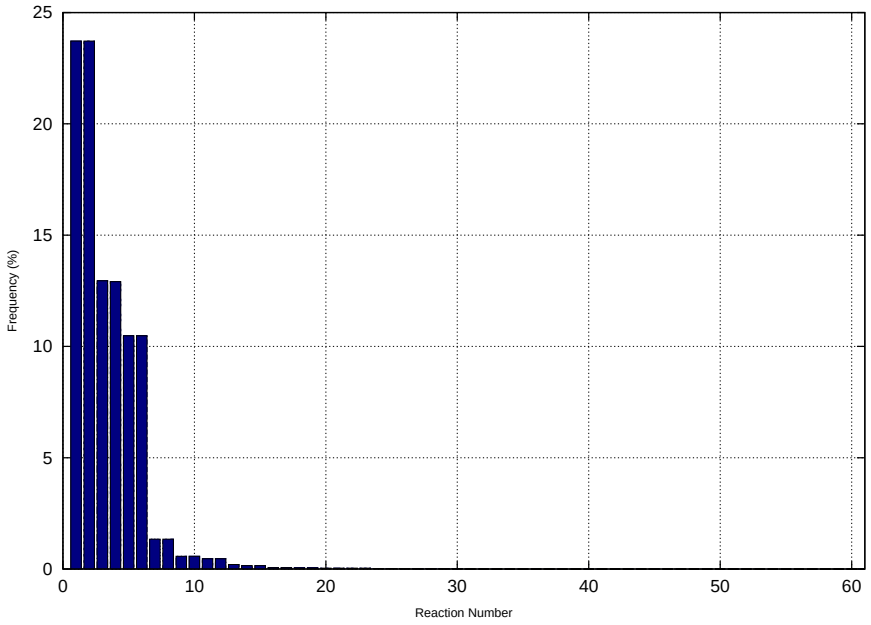


Figure 3.1: Reaction Frequencies of HSR

The *CellMC* distribution contains all 4 models expressed as SBML Level 2, translated from earlier SBML Level 1 versions mostly culled from the *StochKit* distribution [17]. The models are described in full in Appendix A, and only short descriptions are given here after a brief introduction to the notation.

3.1 Decay Dimerisation Reaction

The decay dimerisation (DD) model is non-stiff, and the simplest model system used, with 3 species and 4 reaction channels. The full details are given in Appendix A.2.

DD was included for comparison with *StochKit* [17], as it ships as an ODM example for both the serial and MPI versions, has results in a recent summary [16], and is used in Li & Petzold's recent work on the GPU [19].

3.2 Metabolite-Enzyme Reaction

With a mean timestep of just 250ms, the metabolite-enzyme (ME) model is a simple, slightly stiff, generic model of 4 species — two metabolites, X and Y, whose production is controlled by corresponding enzymes, E_X and E_Y — with 9 reaction channels. Full details are given in Appendix A.3.

Metabolite-enzyme is chosen because it is a small system, easy to work with in development, but stiff enough that reaction ordering matters. It has also been solved with high-resolution methods, so there are good comparisons in the literature[6].

3.3 *E. Coli* Heat-shock Reaction

The heat-shock reaction of *E. coli* (HSR) is a very stiff system of 28 species and 61 reaction channels which models the response of *E. coli* to heat stress. At elevated temperatures, proteins begin to denature; the response is the activation of several genes that produce chaperone enzymes, some of which help to refold denaturing proteins, whilst others help to degrade denatured proteins [24, 2].

A generally stiff system, HSR gets progressively stiffer as the simulation progresses[1] with large changes in the propensities of some reactions. Systems like this were one of the principal motivations for the Sorting Direct Method (§ 2.2.3). The details are given in Appendix A.4.

HSR is chosen principally because it is a real-world system whose characteristics have made it popular as a benchmark, and its inclusion here facilitates ready comparisons of execution times [24][2][11].

3.4 Circadian Rhythm

The circadian rhythm is a well-known cellular phenomenon, also known as the "biological clock", and modeled by the Vilar oscillator [27][1]. The version of the Vilar oscillator used here it is a system of 9 species and 16 reaction channels. The details of the model are given in Appendix A.5.

It is chosen, not so much for its stiffness, which is on a par with HSR (below), but because its stochastic resonance means that it is particularly ill-suited to deterministic simulation, and particularly well-suited to SSA. Its oscillatory nature lends itself to attractive animation.

[1] With the given initial conditions and sufficiently long simulation time

4. Platforms

An overview of the *Cell/BE* architecture is provided, followed by detailed descriptions of all the hardware and software used for the development and profiling of *CellMC*.

A total of 7 different *Cell/BE* and *IA32* systems were used during development and test. Tables 4.1 and 4.2 summarise these systems and their corresponding software installations, respectively.

Name	Processor	Cores/SPUs	GHz
esprit	*Intel Core™ 2*	2	1.86
scrat	*AMD Athlon™ 64*	1	2.00
arich	*AMD Opteron™ 2216*	2×2	2.40
grad	*Intel Xeon®* E5240	2×4	2.50
skara	*Sony/IBM/Toshiba Cell/BE*	6	3.19
cell2	*IBM PowerXCell™ 8i*	16	3.20
Cluster	*Sony/IBM/Toshiba Cell/BE*	4×6	3.19

Table 4.1: Hardware Platforms

Name	Distribution	Ver.	gcc	libxml2	libxslt
esprit	*CentOS*	5.3	4.1.2	2.6.26	1.1.17
scrat	*Ubuntu*	8.04	4.2.4	2.6.31	1.1.22
arich	*Scientific Linux*	4.7	3.4.6	2.6.26	1.1.17
grad	*Scientific Linux*	5.3	4.1.2	2.6.26	1.1.17
skara	*Yellow Dog Linux*	6.0	4.1.1	2.6.26	1.1.17
cell2	*Fedora*	9	4.1.1	2.7.2	1.1.24
Cluster	*Fedora*	9	4.1.1	2.6.26	1.1.17

Table 4.2: Software Installations

4.1 The Cell Broadband Engine

The *Cell Broadband Engine* (*Cell/BE*) is a heterogeneous multicore microprocessor developed by *IBM*, *Toshiba*, and *Sony Computer Entertainment* to power the *Sony Playstation 3* (PS3) games console [3].

In essence, a single *Cell/BE* processor consists of 9 processor cores and associated hardware on a single die[1]: the *PowerPC Processor Element* (*PPE*), with a 64-bit,

[1]The architecture documentation is technically neutral on the number of cores, but the only *Cell/BE* processors actually available have 1 PPE and 8 SPEs

dual-thread *P*owerPC core and conventional L1 and L2 cache; and 8 *Synergistic Processor Elements* (*SPE*s), each consisting of a core, called the *Synergistic Processing Unit* (*SPU*), 256KiB *local store* (LS), and *memory flow controller* (*MFC*). The 9 processing elements can communicate with each other via the high-speed *Element Interconnect Bus* (*EIB*), or with off-chip memory and peripherals via memory and I/O controllers connected to the EIB. Each SPU is a cut-down *PowerPC* processor with 128 vector registers, each 128 bits, and *VMX* (better known as *AltiVec*) vector instructions. [12, pp.27–31][13, pp.37–42,44][3]

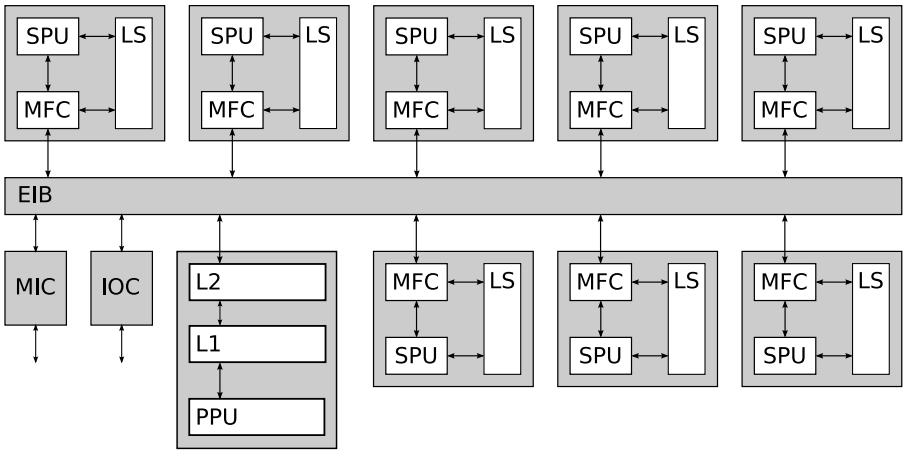


Figure 4.1: *Cell/BE* architecture showing 8 SPEs, the PPE, I/O controller, and main memory interface controller connected to the element interconnect bus.

4.1.1 *Sony PlayStation® 3*

The *Sony PlayStation® 3* games console (PS3) contains a single first-generation *Cell/BE* processor at 3.192GHz, 512MiB of RAM, 1 Gbps ethernet, and a customised *Nvidia* GPU — the so-called "Reality Synthesizer" (RSX).

Only 6 SPUs are available to Linux® applications, since, to increase wafer yields, one SPU is disabled on all PS3s, and the *Sony Game OS* (hypervisor) locks a further SPU.

The hypervisor also mediates all communication to the graphics and network subsystems, disabling access to accelerated graphics entirely, and adding significant latency to network communications.

The first-generation *Cell/BE* processor has undergone two die-shrinks, from 90nm to 65nm to 45nm. Accordingly, *when* a PS3 is purchased determines its power consumption to some degree, with the peak power consumption of the 90nm and 65nm versions being ≈200W and ≈135W respectively. It is likely that all the PS3s used have 65nm *Cell/BE* processors.

4.1.2 Sony PlayStation 3 Cluster

The UPPMAX PS3 cluster consists of 8 *Sony PlayStation 3* games consoles, of which 4 were commissioned and available.

User-local installations of *libxml2* and *libxslt* were used in order to overcome biarch difficulties (only the 64-bit versions of the packages were installed and without the corresponding development packages). A user-level installation of `OpenMPI` 1.3.1 was used after some difficulty was encountered with the installed MPI version.

4.1.3 IBM QS22

An IBM blade with two 3.2GHz *IBM PowerXCell*™ 8i processors (second-generation *Cell/BE* processors), with all 8 SPUs available on both processors, for a total of 16 SPUs.

No explicit programmatic communication between the two *Cell/BE* processors is necessary, and all 16 SPUs can be used transparently.

4.2 x86 Platform

4.2.1 Modest PC Workstation

Esprit is a generic PC with an 1.86GHz *Intel Core*™ *2* CPU with 2MiB of L2 cache (family 6, model 15, stepping 2) and 2GiB of RAM, running *CentOS* release 5.3 (Final).

In addition to the libraries required by *CellMC*, *mpich2* 1.0.8p1, needed by *StochKit*, in addition to its corresponding development packages, was installed from the ordinary *CentOS* distribution and updated as of April 2009.

4.2.2 *AMD Opteron*™ Multiprocessor

A dual dual-core (4 cores in total) 2.4GHz *AMD Opteron*™ 2216 (family 15, model 65, stepping 2) machine with 1MiB of L2 cache per core (4MiB total), running *Scientific Linux* release 4.7 (*Beryllium*).

4.2.3 *Intel Xeon*® Multiprocessor

A dual quad-core (8 cores in total) 2.5GHz *Intel Xeon*® E5420 (family 6, model 23, stepping 6) machine with 3MiB of L2 cache per core (6MiB shared on every 2-core die for 12MiB per dual-die processor package) for a machine total of 24MiB, with 16GiB of RAM, running *Scientific Linux* release 5.3 (*Boron*).

5. CellMC Compiler

5.1 Basic Operation

A homogeneous chemical system is expressed as an SBML model (the SBML models of § 3 are included in the distribution).

CellMC is invoked in two passes, with an intervening use of an XSL-T processor to re-order unoptimised models. If the document order of the reactions in an SBML model already coincide with ODM order, only the normal 2nd pass is required.

5.1.1 Pass 1

1. *CellMC* is invoked with the `-p` (profiling) flag to generate `C` code from the SBML file via *XSL-T*.

2. *CellMC* compiles the `C` code and supporting libraries and produces a native executable.

5.1.2 SBML Model Transformation

1. The executable is run from the command-line for a representative final simulation time.

2. The executable emits an *XSL-T* transformation which, when applied to the original model, re-orders the reactions in descending order by total number of occurrences over the simulation time (ODM order).

3. The transformation is applied to the original SBML model, using any XSL-T processor[1], producing a new reordered SBML model,

5.1.3 Pass 2

- The new model is transformed via *XSL-T* into `C` code, compiled, and linked, as in Pass 1, producing the optimised executable.

A brief user guide for *CellMC* is included in Appendix B, with an example of operation in § B.4.

[1]The GNU XSL-T processing library, *libxslt*, is required by *CellMC*, and `xsltproc`, an easy-to-use command-line XSL-T processor, is installed with it by default.

5.2 Output

The first-pass (profiling) output is a XSL transformation which, when applied to the original SBML file, re-orders the reactions into ODM order.

The usual (second pass) output is the final population vector preceded by a header containing extensive metadata about the model, compilation, and invocation, including run timing data; see Appendix B.3 for details.

5.3 Implementation

On all supported platforms, *CellMC* uses XSL-T to transform source SBML into `C` code and compiles it with the local *gcc*. The XSL-T transformation is done using GNU *libxslt*, since this is installed by default by all common Linux distributions. On *Cell/BE*, the *IBM Cell SDK 3.0* is required.

CellMC requires *gcc* and will not work with the *IBM* `xlc` or *Intel* `icc` compilers.

Although some code is shared, it is important to differentiate clearly between the source-code that is compiled to produce *CellMC* and the source-code that *CellMC* uses to compile executables realising SSA from an SBML file.

Cross-platform libraries were written for common *CellMC* functionality, such as command-line argument processing, performing XML validation and pre-conditioning, and XSL-T transformation.

The support code, needed by *CellMC* at runtime, also has substantial shared code for command-line argument processing, writing final populations to disk, generating the metadata header for the output file, and SIMD vector representation.

The similarity ends, however, when it comes to the code that actually implements SSA. For each architecture, there are a few distinct XSL-T files although, again, platform-neutral transformations are shared. *CellMC* passes a number of parameters into the XSL-T transformations, and passes the same parameters via the *gcc* "command-line" as preprocessor constants. In the `C` support files, only preprocessor directives are available to tailor functionality to the platform and *CellMC* options. In the XSL-T files, however, both XSL-T parameters and preprocessor directives are available and are mixed freely and rather arbitrarily.

5.3.1 Parallel Pseudo-Random Number Generation

SSA requires a vast number of uniformly distributed pseudo-random numbers, perhaps 10^{11} for a typical simulation. *CellMC* uses a fast Mersenne Twister (19937) on both *IA32* (see § 5.5.1) and PS3 (IBM's `mc_rand` library). For the multi-(core/processor) case, the PRNGs are boot-strapped with different seeds at startup from the POSIX® `rand48` family LFSR PRNG, itself seeded with a master seed (settable on the command-line with a hard-coded default). In principle, there is a very small risk of correlation with this scheme. A better solution is to use a distributed

PRNG, but this would have added a significant extra burden to the software development effort for no apparent gain — it appears to make no difference in practice — and the current scheme is easily replaceable with a "proper" distributed PRNG should one become available.

5.3.2 Floating-Point Precision

Single-precision (32-bit) floating-point representation of reaction constants and propensities has not been found to be problematic in GPU or FPGA implementations, nor was it found to be in *CellMC* testing. However, it was found that single-precision representation of *time* was highly problematic. In early tests on *Cell/BE*, it was found for 500s HSR (§ 3.3) that when time was reckoned starting at zero and adding on timesteps up to the final time, the average number of timesteps was over 10^8, whilst when time was reckoned starting at the final time and subtracting timesteps down to zero, the average number of timesteps was $\approx 4.8 \times 10^7$. Representing time in double precision, we found the number to be $\approx 6.2 \times 10^7$ with excellent agreement between different implementations.

This can be explained by the extreme stiffness of HSR often leading to small timesteps of the order of 10^{-7}s, which differ from the final time by as much as 2^{28} leading to the problematic loss of significant digits due to additive roundoff, or entire timesteps, in single precision, where the significand is just 23 bits, and necessitating the use of (Kahan) *compensated summation* [14] for time. This alleviates the problem, and the number of timesteps for single-precision time with compensated summation are found to agree with double-precision results.

5.4 *Cell/BE* Platform Specifics

On the *Cell/BE* platform, the build process is somewhat involved. *CellMC* follows the pattern reverse-engineered from Makefiles supplied with the *IBM Cell SDK 3.0*.

1. Compile the SPU code with *spu-gcc* to object code
2. Transform SPU object code to a PPU-embeddable module with *ppu-embedspu*
3. Transform the embeddable module to an archive with *ppu-ar*
4. Compile PPU code with *ppu-gcc*
5. Link PPU code with embeddable archive with *ppu-gcc*

Peculiarities of the SPUs on *Cell/BE* mean that certain intuitions about computational cost do not apply. For example, just doing arithmetic is often cheaper than deciding whether to do it or not, because branches are expensive on the SPU. Similarly, the relative expense of indirection on the SPU and the huge file of 128 vector registers means that the working dataset of many models, up to about the size of HSR, can be kept entirely in registers.

After climbing the *Cell/BE* learning curve, and considerable experimentation, it was clear that ODM was likely to give the best results — it is quicker to do a linear search of 10 registers than to so much as fetch a value from the local store.

5.4.1 Intrinsics

The *IBM Cell SDK 3.0* provides SPU *intrinsics*, `C`-language functions that, to a large degree, map directly to SPU assembly language statements. These are used extensively in the SPU code, which is the part that actually realises SSA.

5.5 *IA32* Platform Specifics

When it was decided that a fast *IA32* version was required, a great deal of time had been spent optimising the *Cell/BE* code. The following were identified as likely bottlenecks on *IA32*:

- Random number generation — two needed per reaction
- The `log()` function

5.5.1 A Re-entrant SIMD-oriented Fast Mersenne Twister

When development on *IA32* commenced, Saito and Matsumoto's *SIMD-oriented Fast Mersenne Twister* (*SFMT*) [25] was the fastest available high-quality PRNG for *IA32*. Unfortunately, its extensive use of global variables made it unsuitable for multithreaded operation, so it was modified for re-entrancy. It was also profiled and found to have an extraordinarily sharp profile, with execution time strongly dominated by a single function: `mm_recursion()`. Disassembly revealed considerable register spill, which could not be entirely ameliorated using the `r`egister storage-class modifier, so sequence of 9 intrinsics was re-coded as inline assembly language, and the overall speed of SFMT was doubled. This heavily modified version of SFMT-1.3.3 was dubbed *RSMT* for *Re-entrant SIMD-oriented Mersenne Twister*.

CellMC's `-n` option allows the POSIX® `rand48` family of PRNGs to be used instead of RSMT, but its use is unnecessary and discouraged.

5.5.2 SIMD `log()`

On *Cell/BE*, the *IBM Cell SDK 3.0* provides an inline SIMD log function, `_logf4()`, which computes the natural logarithm of a SIMD vector of four floats simultaneously using an eighth-order polynomial approximation[2] algorithm. This was ported to SSE2 assembly language to avoid the slower option of calling the library `logf()` function 4 times.

[2] Attributed to "C. Hastings Jr, 1955", in the IBM source-code file

CellMC's `-l` option allows the POSIX® `logf()` function or the FPU to be used directly on *IA32* in place of the SSE2 assembly language version, but its use is unnecessary and discouraged.

5.6 Code and Build Management

The code is maintained in a *Subversion* repository and uses the *GNU Autotools* build chain [22, 23, 4].

Accordingly, building from source is extremely simple and follows a familiar and well-established procedure, for example:

```
$ tar xzf cellmc-0.2.16.tar.gz
$ cd cellmc-0.2.16
$ ./configure
$ make
```

6. Results

6.1 Correctness of Results

We demonstrate that programs generated by *CellMC* produce results consistent with expectations. We do this by running simulations with parameters from the literature and showing that the results are comparable.

6.1.1 Decay Dimerisation

Figure 6.1 shows the means of the final populations for 1000 trajectories of the decay dimerisation model versus simulated time. The initial conditions $X_1 = 10^5$, $X_2 = X_3 = 0$ are taken directly from Gillespie's tau-leap paper, and the curves of Figure 6.1 are directly comparable with Figure 4(b) found there [10, p.1724].

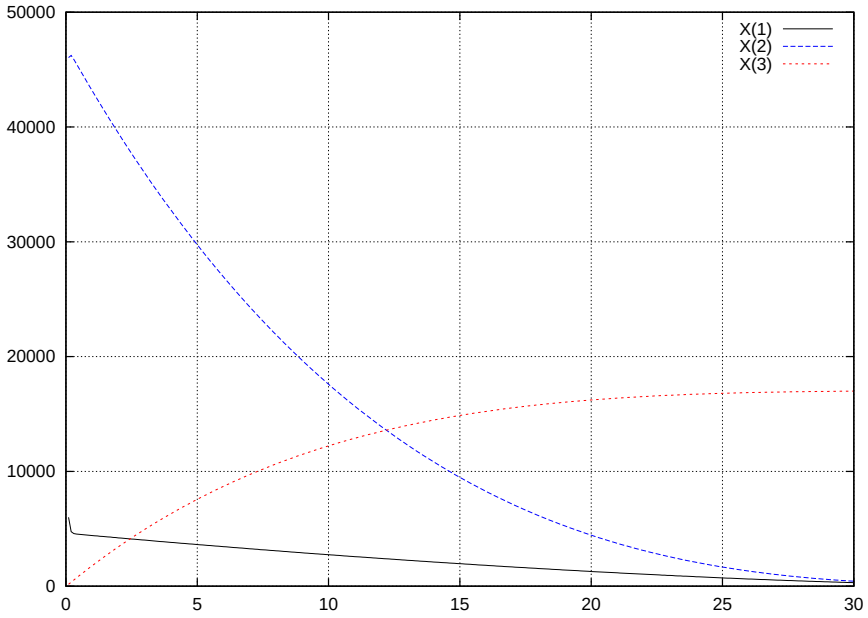
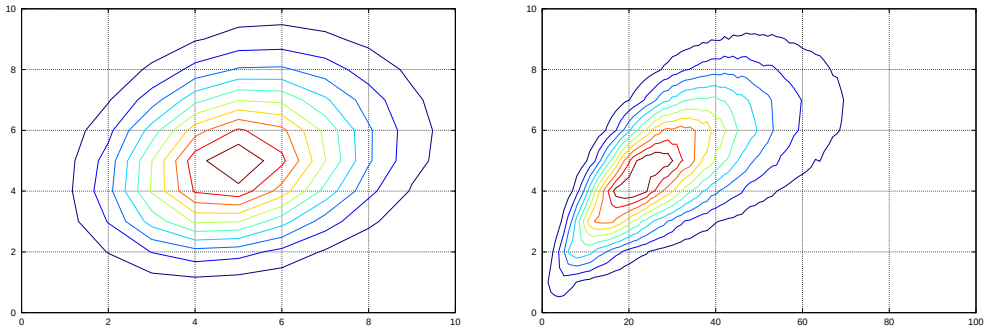


Figure 6.1: Population means vs. time for decay dimerisation

6.1.2 Metabolite Enzyme

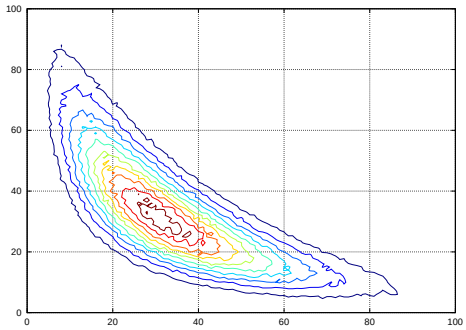
Figure 6.2 shows isolines of selected 2D marginal PDFs for 10^6 trajectories at a final time of 1500s. The plots are labeled a vs. b according to the species on the vertical and horizontal axes respectively.

The final time and plots were chosen for direct comparison with Engblom [6, p.887].

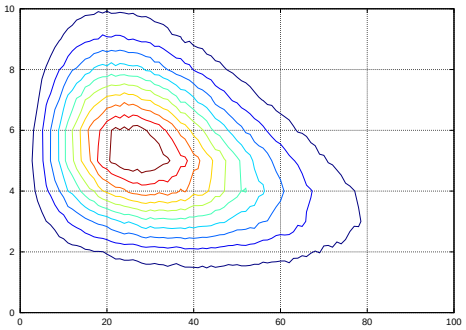


(a) e_y vs. e_x

(b) e_x vs. x



(c) y vs. x



(d) e_y vs. x

Figure 6.2: Isolines of selected 2D marginal PDFs for 10^6 metabolite-enzyme trajectories of 1500s

6.1.3 *E. Coli* Heat-shock Reaction

Figure 6.3 shows a marginal PDF for the σ_{32} factor at 50s for different numbers of trajectories. The curve for 10^3 trajectories is comparable with Hellander[11, p.7].

6.1.4 Circadian Rhythm

Figure 6.4 shows the cyclic oscillation of the two most populous species in the Vilar oscillator model by plotting their means against each other with time. Although the oscillation is clear, it appears here that the oscillation is damped. In reality, the variance of the populations is merely increasing along the path of the limit cycle, which makes the mean appear to converge toward the centre. The stability of the oscillation is clearer in animation[1].

[1]See `http://cellmc.org/examples/vo/`

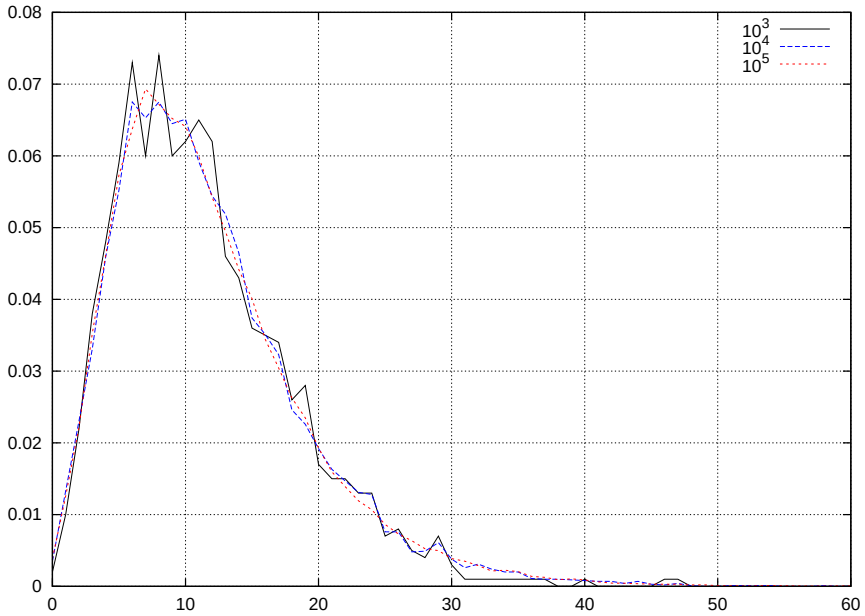


Figure 6.3: 1D marginal PDF of σ_{32} factor for 10^3 and 10^4 HSR trajectories at 50s

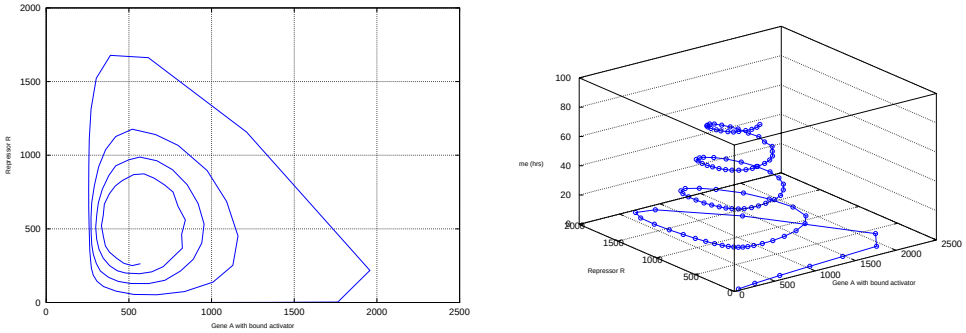


Figure 6.4: Means of D'_a vs. R in the Vilar oscillator for 100hrs

6.2 Performance

All timings are "wall-clock" times recorded by *CellMC* (or, rather, by the programs produced by *CellMC*). As this information is written into a header in the output file (B.3), it does not include the time taken to write the final populations to disk. The time recorded is, therefore, the total time taken by the setup, computation, and communications. In practice, the additional time taken to save the final populations is seldom significant, a few tenths of a second, but for extremely large numbers of trajectories, the difference is considerable. For example, 3×10^6 trajectories of the metabolite enzyme problem (§ 3.2) was observed to take approximately 10s to write

to disk on the PS3 cluster (§ 4.1.2).

In addition, on the PS3 cluster, *mpirun* uses *ssh* to connect to the other nodes on the cluster and launch a daemon on each one, which adds another second or two of unrecorded latency.

In most cases, timings are averages of 5–10 identical runs conducted when the machines were otherwise quiet. The consistency of timing between runs under these circumstances is quite remarkable, in many cases, timings for all 10 runs were identical to the millisecond.

6.3 Comparison of Results

6.3.1 Comparison on PC Platform

PC Versions in the Literature

Table 6.1 shows the average time to compute one 500s HSR (§ 3.3) trajectory[2] and "speed" in millions of reactions per second (Mrps), where available, from ODM results in the literature; also shown are analogous results from *CellMC* on three similar *IA32* machines.

Although the hardware is not identical, so a direct comparison cannot be drawn because of differences in clock-speed and instruction latency, it is nevertheless clear that *CellMC* is considerably faster than published results[3].

Description	Processor	GHz	Time (s)	Mrps
Cao *et al.* [2]	*Intel Pentium™ 4*	1.4	76.5	–
McCollum *et al.* [24]	*Intel Pentium™ 4*	2.0	52.56	0.88
Yoshimi *et al.* [29]	*Intel® Core™ 2 Quad*	2.4	–	1.61
CellMC	*AMD Athlon™ 64*	2.0	7.63	8.09
CellMC	*Intel Core™ 2*	1.86	6.11	10.38
CellMC	*Intel® Core™ 2 Quad*	2.5	4.22	14.74

Table 6.1: Single Core x86 Performance Comparison for 500s HSR

Comparison with *StochKit*

StochKit is a capable and popular toolkit for SSA with an active user community of over 100 users. In addition to the exact SSA methods described here, it implements approximate methods such as tau-leaping and slow-scale SSA. Importantly, it is occasionally used as a benchmark for other implementations [29].

[2]Note that in most tables, the "Time" column is the total runtime.

[3]The results listed are from papers comparing algorithms, rather than attempting to achieve the fastest implementation.

Table 6.2 shows a direct comparison between *StochKit* and *CellMC*, "out of the box" on the same modest workstation (§ 4.2.1), with respect to the decay dimerisation model (§ 3.1). This model is chosen because *StochKit* ships with an ODM example for both the serial and MPI versions. Although *StochKit* also ships with a HSR example, it uses the slower direct method (§ 2.1), which would not make for fair comparison. The parameters (10,000 10-second trajectories) are those used in the *StochKit* examples.

Software	Cores	Runtime (s)	Speedup
StochKit	1	144.3	1
CellMC	1	14.7	9.8
StochKit (MPI)	2	90.7	1
CellMC (pthreads)	2	7.4	12.3

Table 6.2: Software comparison for 10,000 10s DD Simulations

6.3.2 Comparison with GPU & FPGA

The sole reported implementation of SSA on the GPU is by Li and Petzold [19]. Unfortunately, there is insufficient detail to compare their implementation. They do assert that their GPU implementation is about 200 times faster than a baseline running on the host PC, but since we don't know how fast it is either, we can draw no conclusions.

On the FPGA, again, "speedup" claims over a particular PC are made without giving enough detail of the chemical model to allow it to be reconstructed, without any details whatever of the PC software, or comparing with arcane PC hardware. As is clear from Table 6.2, implementation can make an order-of-magnitude difference on the same machine, which means that most speedup claims in the literature, to be charitable, do not form a valid basis for comparison [31, 30, 28].

One very recent FPGA result that may be used as a basis for direct comparison achieves 8.98 millions of reactions per second (Mrps) when running 500s HSR (3.3) trajectories, which they also report as 5.5 times faster that *StochKit* on an *Intel® Core™ 2 Quad* [29]. Table 6.3 shows results for *CellMC* for the same simulation; a single *PlayStation® 3*, for example, is 7 times faster with *CellMC*.

Machine	Type	Time (m:s)	Mrps
esprit	PC (dual)	49:49	20.77
arich	PC (quad)	28:32	36.50
skara	PS3	15:46	66.01
grad	PC (octo)	9:13	112.27
cell2	QS22	6:03	174.43
Cluster	4×PS3	4:10	261.14

Table 6.3: Platform Comparison for 10^3 500s HSR (§ 3.3)

6.4 Scalability

6.4.1 *Cell/BE*

Table 6.4 shows the runtimes and speedup against number of SPUs for a *CellMC*-produced program realising the decay dimerisation model[4]. Speedup is linear with no sign of saturation: at worst, it is better than 99% of perfectly linear speedup.

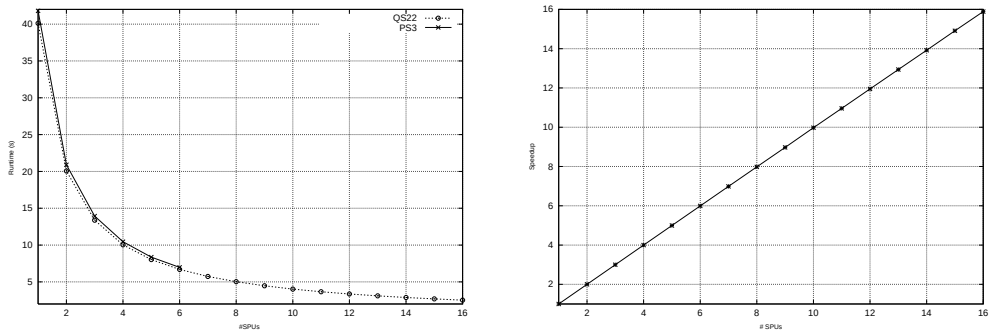
SPUs	Runtime (s)		Speedup	
	PS3	QS22	PS3	QS22
1	41.80	40.12	1.00	1
2	20.90	20.06	2.00	2.00
3	13.94	13.38	3.00	3.00
4	10.45	10.03	4.00	4.00
5	8.36	8.03	5.00	5.00
6	6.97	6.69	6.00	6.00
7		5.74		6.99
8		5.03		7.98
9		4.47		8.97
10		4.02		9.97
11		3.66		10.96
12		3.36		11.95
13		3.10		12.94
14		2.88		13.92
15		2.69		14.91
16		2.53		15.89

Table 6.4: Speedup for 30,000 10s DD Simulations on *Cell/BE*

Figure 6.5 depicts the data of Table 6.4 graphically. Since the speedup is identical for PS3 and QS22 up to 6 SPUs, only the data for the QS22 is plotted in the speedup

[4] *CellMC*-produced programs accept a `-c`*n* argument to change the number of cores used.

graph.



(a) Runtime (b) Speedup

Figure 6.5: Runtime and Speedup vs. Number of SPUs on *Cell/BE*

PlayStation®3 Cluster

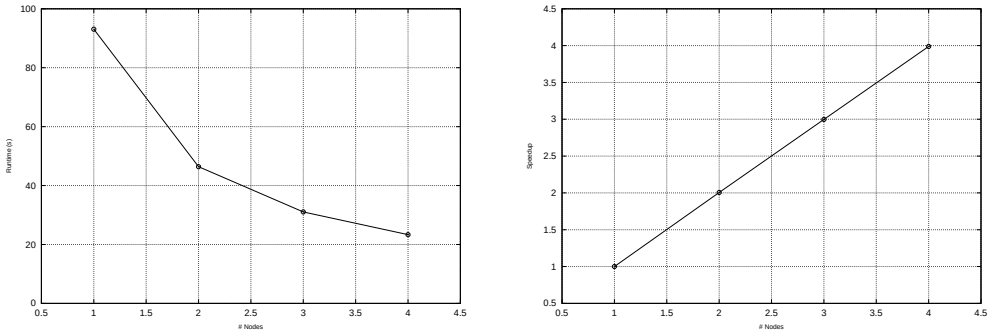
Table 6.5 shows the runtimes and speedup against number of nodes for a *CellMC*-produced program realising the metabolite-enzyme model[5], running on a cluster of 4 *Sony PlayStation®3*s. Each node uses all 6 available SPUs. Speedup is linear with no sign of saturation.

Nodes	Runtime (s)	Speedup
1	93.09	1
2	46.42	2.01
3	31.05	3.00
4	23.34	3.99

Table 6.5: Speedup for 1.5×10^6 1,500s ME Simulations on PS3 Cluster

Figure 6.6 plots the data in Table 6.5.

[5] *CellMC* has a `-M` option to produce MPI code



(a) Runtime (b) Speedup

Figure 6.6: Runtime and Speedup vs. Number of Nodes on PS3 Cluster

6.4.2 x86

It is clear from Table 6.2 that *CellMC*-produced programs take full advantage of the 2nd core on a dual-core *IA32* processor. Table 6.6 shows the running time and speedup vs. number of cores[6] for a *CellMC*-produced program realising HSR (§ 3.3) running on a quad-core *AMD Opteron*™ and a dual *Intel® Core™ 2 Quad* (8 cores).

Speedup is almost perfectly linear, but to maintain linear speedup on the final processor, it is necessary to use LINUX®-specific thread-CPU affinity system calls to "attach" each thread of simulation to a specific CPU. In the absence of this, performance degrades, rather than improves, when the fourth core of four is put to use. This is a known problem with the default kernel scheduler of some LINUX® kernels. The "starred" entry of Table 6.6 shows what happens with the default scheduling affinity.

CPUs	Runtime (s)		Speedup	
	arich	*grad*	*arich*	*grad*
1	22:14	13:52	1	1
2	11:31	7:04	1.93	1.97
3	7:51	4:44	2.83	2.95
4	∗8:16	3:35	∗2.69	3.91
5		2:50		4.91
6		2:23		5.86
7		2:04		6.78
8		1:52		7.52

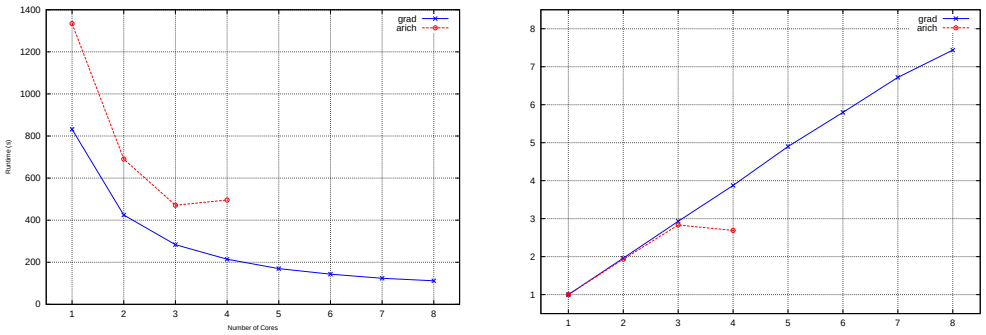
Table 6.6: Speedup for 2,500 50s HSR Simulations on *IA32*

Figure 6.7 plots the data in Table 6.6 using the 4-processor value for the (bad) case

[6] Again using the `-cn` option to the *CellMC*-produced program

with default thread/CPU affinity.

Speedup is, to some degree, dependent on the number of trajectories chosen, since the number of trajectories actually computed is the smallest integer larger than the requested number that is a multiple of the number of SIMD slots (4) and the number of CPUs. The data of Table 6.6 and Figure 6.7 is not corrected for this. For large numbers of trajectories, apparent speedup is generally better even than the 94% of perfect speedup shown here.



(a) Runtime (b) Speedup

Figure 6.7: Runtime and speedup vs. number of CPUs on *IA32* showing (solved) scheduling issue

6.5 Platform Comparison

Tables of timings are presented below for a representative selection of different problems over timescales from seconds to many tens of minutes to give a picture of the comparative performance of *CellMC* on different platforms. One set of results is included for each of the 4 models.

Timings for *grad* marked with an asterisk are estimated from shorter simulations because they would exceed a 30-minute CPU time limit.

Machine	Type	Time (s)	Mrps
esprit	PC (dual)	22.03	35.89
grad	PC (octo)	3.65	218.23
skara	PS3	6.97	113.24
cell2	QS22	2.53	312.83
Cluster	4×PS3	1.80	439.60

Table 6.7: Platform Comparison for 3×10^4 10s DD (§ 3.1)

Machine	Type	Time (s)	Mrps
esprit	PC (dual)	303.68	13.03
grad	PC (octo)	28.03	141.12
skara	PS3	44.03	89.82
cell2	QS22	16.50	239.78
Cluster	4×PS3	11.24	352.00

Table 6.8: Platform Comparison for 10^6 1000s ME (§ 3.2)

Machine	Type	Time (m:s)	Mrps
esprit	PC (dual)	39:05	20.68
grad	PC (octo)	∗7:25	112.30
skara	PS3	12:59	62.40
cell2	QS22	4:53	166.52
Cluster	4×PS3	3:18	247.02

Table 6.9: Platform Comparison for 10^4 50s HSR (§ 3.3)

Machine	Type	Time (m:s)	Mrps
esprit	PC	17:34	30.87
grad	PC (octo)	3.03	178.47
skara	PS3	5:39	96.08
cell2	QS22	2:07	256.74
Cluster	4×PS3	1:25	383.79

Table 6.10: Platform Comparison for 10^5 25hr CR (§ 3.4)

7. Conclusions & Future Work

7.1 General Conclusions

Programs produced by *CellMC* yield simulation results comparable with the literature. Initial fears about single-precision arithmetic being insufficient were unfounded for the model systems tested (§ 6.1) [19], but precision-compensated (Kahan) summation is necessary in single-precision (§ 5.3.2), particularly for long simulations of stiff systems.

On PC, *CellMC* programs perform extremely well when compared to extant PC implementations, being almost 10 times faster on a single core than *StochKit* and reported speeds in the literature on comparable single core CPUs (§ 6.3.1). On up to 8 core systems, *CellMC* programs scale better than 94% perfectly (§ 6.4.2) and further outperform both *StochKit* and implementations in the literature.

The *Cell/BE* programs produced by *CellMC* are, roughly speaking, the same speed per core as those on a typical PC. Accordingly, a single *Sony PlayStation® 3*, with 6 available SPUs, is over 3 times faster than the "typical desktop" dual-core *IA32* programs (§ 6.5), and performance scales linearly both with number of SPUs (§ 6.4.1) and on a cluster (§ 6.4.1).

Given that a *PlayStation® 3* costs somewhat less than a typical PC workstation and has approximately the same power consumption (≈200W), the *PlayStation® 3* offers at least 3 times the performance per watt, and at least 5 times the performance per unit hardware cost when compared to contemporary PC hardware.

CellMC programs, particularly on the *PlayStation® 3*, appear to outperform the most recent work on FPGAs by a considerable multiple (§ 6.3.2).

The ODM version of SSA is "embarrassingly parallel" and well-suited to *Cell/BE* and other multicore processors, although it is not trivially SIMDisable due to the branching necessary at each step.

7.2 Future Work

CellMC's principal weakness is that the programs it produces are unable to store whole trajectories or save populations at intervals, rather than just final populations. This should be remedied.

CellMC should have the ability to compute marginal PDFs of selected species, rather than having to rely on external software.

CellMC should compile models to shared objects with a consistent interface, rather than compiling a monolithic application, so that compiled models could be linked at runtime with a generic application stub that computes and displays marginal PDFs. This is, however, difficult to do in a portable way without compromising performance.

A complete redesign could see *CellMC* just-in-time compiling SBML models into dynamically loadable shared objects, and loading these into a generic feature-rich front-end.

CellMC could easily be extended to do MCMC simulations other than SSA.

While there are good reasons to prefer ODM over SDM or LDM on *Cell/BE*, *CellMC* should implement LDM on *IA32*.

Bibliography

[1] Naama Barkai and Stanislas Leibler. Circadian clocks limited by noise. *Nature*, 403(6767):267–268, January 2000. Available from `http://www.nature.com/nature/journal/v403/n6767/pdf/403267b0.pdf`.

[2] Yang Cao, Hong Li, and Linda Petzold. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *Journal of Chemical Physics*, 121(9), September 2004. Available from `http://link.aip.org/link/?JCPSA6/121/4059/1`.

[3] T. Chen, R. Raghavan, J.N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation — a performance view. *IBM J. Res. & Dev.*, 51(5), September 2007.

[4] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, Inc., Sebastopol, California, June 2004. ISBN: 0-596-00448-6, available from `http://svnbook.org/`.

[5] Stefan Engblom. *Numerical Methods for the Chemical Master Equation, Licentiate thesis 2006-07*. PhD thesis, Uppsala University, Division of Scientific Computing, Department of Information Technlogy, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden, August 2006.

[6] Stefan Engblom. Galerkin spectral method applied to the chemical master equation. *Communications in Computational Physics*, 5(5):871–896, May 2009. Also *Paper III* in *Numerical Solution Methods in Stochastic Chemical Kinetics*, Acta Universitatis Upsaliensis, ISBN 978-91-554-7322-8.

[7] MA Gibson and J. Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *Journal of Chemical Physics*, 104(9):1876–1889, March 2000. `http://dx.doi.org/10.1021/jp993732q`, `http://tinyurl.com/6lwatr`.

[8] Daniel T. Gillespie. A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions. *Journal of Computational Physics*, 22(4):403–434, December 1976. `http://dx.doi.org/10.1016/0021-9991(76)90041-3`, `http://tinyurl.com/5bmodz`.

[9] Daniel T. Gillespie. A rigorous derivation of the chemical master equation. *Physica A: Statistical Mechanics and its Applications*, 188(1-3):402–425, 1992. Available from `http://dx.doi.org/10.1016/0378-4371(92)90283-V`.

[10] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *Journal of Chemical Physics*, 115(4), July 2001. Available from `http://www.soe.ucsc.edu/~msmangel/Gillespie01.pdf`.

[11] Andreas Hellander. Numerical simulation of well stirred biochemical reactions governed by the master equation. Licentiate thesis 2000-003, Uppsala University, Division of Scientific Computing, Department of Information Technlogy, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden, 2008.

[12] IBM Corporation, Toshiba Corporation, and Sony Computer Entertainment Inc. *Cell Broadband Engine Architecture*. IBM Systems and Technology Group, October 2007. Version 1.02.

[13] IBM Corporation, Toshiba Corporation, and Sony Computer Entertainment Inc. *Cell Broadband Engine Programming Handbook*. IBM Systems and Technology Group, April 2007. Version 1.1.

[14] W. Kahan. Further remarks on reducing truncation errors. *CACM*, 8(1):40, January 1965.

[15] Thomas R. Kiehl, Robert M. Mattheyses, and Melvin K. Simmons. Hybrid simulation of cellular behaviour. *Bioinformatics*, 20(3):316–222, 2004. Available from `http://bioinformatics.oxfordjournals.org/cgi/reprint/20/3/316.pdf`.

[16] Hong Li, Yang Cao, Linda R. Petzold, and Daniel T. Gillespie. Algorithms and Software for Stochastic Simulation of Biochemical Reacting Systems. *Biotechnology Progress*, 24(1):56–61, January 2008. Available from `http://people.cs.vt.edu/~ycao/publication/Biotechnology_Progress07.pdf`.

[17] Hong Li, Yang Cao, Kevin Sanft, Min K Roh, Marc B. Griesemer, and Fenglin Liao. StochKit: A Stochastic Simulation Toolkit, 2005. Available from `http://www.engineering.ucsb.edu/~cse/StochKit/`.

[18] Hong Li and Linda Petzold. Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems. Technical report, Department of Computer Science, UCSB, Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA, 93106, July 2006. Available from `http://www.cs.ucsb.edu/~cse/Files/ldm0513.pdf`.

[19] Hong Li and Linda Petzold. Efficient Parallelization of Stochastic Simulation Algorithm for Chemically Reacting Systems on the Graphics Processing Unit. *International Journal of High Performance Computing Applications*, 2009. Submitted. Preprint at `http://www.cs.ucsb.edu/~cse/Files/GPUSSA.pdf`.

[20] Larry Lok. The need for speed in stochastic simulation. *Nature Biotechnology*, 22(8):964–965, August 2004. Article in "News and Views" section.

[21] Per Lötstedt and Lars Ferm. Dimensional reduction of the Fokker-Planck equation for stochastic simulation of chemical reactions. *Multiscale Model. Simul.*, 5(2):593–614, 2006.

[22] David MacKenzie, Ben Elliston, and Akim Demaille. *Autoconf: Creating Automatic Configuration Scripts*. The Free Software Foundation, 51 Franklin Street,

Boston, MA 02111, November 2006. Confirmed to be at http://www.gnu.org/software/autoconf/manual/autoconf.pdf for version 2.61 on 2007-05-24.

[23] David MacKenzie, Tom Tromey, and Alexandre Duret-Lutz. *GNU Automake*. The Free Software Foundation, 51 Franklin Street, Boston, MA 02111, October 2006. Confirmed to be at http://sources.redhat.com/automake/automake.pdf for version 1.10 on 2007-05-24.

[24] James M. McCollum, Gregory D. Peterson, Chris D. Cox, Michael L. Simpson, and Nagiza F. Samatova. The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behaviour. *Computational Biology and Chemistry*, 30(1):39–49, February 2006. http://dx.doi.org/10.1016/j.compbiolchem.2005.10.007, http://tinyurl.com/6pm9eh.

[25] Mutsuo Saito and Makoto Matsumoto. SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In Alexander Keller, Stefan Heinrich, and Harald Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer-Verlag, Berlin Heidelberg, December 2007. Proceedings of the Seventh International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing. Available from http://www.springerlink.com/content/j741302101h6615k/fulltext.pdf.

[26] Lukasz Salwinski and David Eisenberg. *In silico* simulation of biological network dynamics. *Nature Biotechnology*, 22(8):1017–1019, August 2004.

[27] José M. G. Vilar, Hao Yuan Kueh, Naama Barkai, and Stanislas Leibler. Mechanisms of noise-resistance in genetic oscillators. *Proc. Nat. Acad. Sci. USA*, 99(9):5988–5992, April 2002. Available from http://www.pnas.org/cgi/reprint/99/9/5988.

[28] M. Yoshimi, Y. Osana, T. Fukushima, and H. Amano. *Stochastic Simulation for Biochemical Reactions on FPGA*, pages 105–114. Springer-Verlag, 2004.

[29] Masato Yoshimi, Yuri Nishikawa, Yasunori Osana, and Akira Funahashi. Practical Implementation of a Network-based Stochastic Biochemical Simulation System on an FPGA. In *FPL 2008: International Conference on Field Programmable Logic and Applications*. IEEE, September 2008. Available from http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=4630034.

[30] Masato Yoshimi, Yasunori Osana, Yow Iwaoka, Akira Funahashi, Noriko Hiroi, Yuichiro Shibata, Naoki Iwanaga, Hiroaki Kitano, and Hideharu Amano. An FPGA Implementation of High Throughput Stochastic Simulator For Large-Scale Biochemical Systems. In *FPL'06: International Conference on Field Programmable Logic and Applications*. IEEE, August 2006. Available from http://ieeexplore.ieee.org/iel5/4095018/4100939/04100980.pdf.

[31] Masato Yoshimi, Yasunori Osana, Yow Iwaoka, Akira Funahashi, Noriko Hiroi, Yuichiro Shibata, Naoki Iwanagaand, Hiroaki Kitano, and Hideharu Amano.

The Design of Scalable Stochastic Biochemical Simulator on FPGA. In *International Conference on Field-Programmable Techhnology*. IEEE, 2005. Available from `http://ieeexplore.ieee.org/iel5/10488/33244/01568590.pdf`.

A. Model Details

A.1 Notation

A system of reactions is denoted by a sequence of arrows with reactants shown at the head, products at the nock[1], and reaction constants over, or under, the shaft. The null or empty-set symbol, $\emptyset$, is used to denote the disappearance or appearance of a reactant or product, respectively, from the purview of the model; in cellular biochemistry, for example, this could mean that a molecule decays or appears from the cytoplasm, but that the precise details of how it arises or vanishes are not modeled.

For example, Reaction A.1 says that a molecule of species S_1 disappears with constant c_1, while Reaction A.3 says that a molecule of species S_2 converts into a molecule of species S_3 with constant c_4.

Reaction A.2 is reversible. It says that two S_1 molecules combine to form one S_2 molecule with constant c_2 and that a S_2 molecule decays into two S_1 molecules with constant c_3.

$$S_1 \xrightarrow{c_1} \emptyset \tag{A.1}$$

$$2S_1 \underset{c_3}{\overset{c_2}{\rightleftharpoons}} S_2 \tag{A.2}$$

$$S_2 \xrightarrow{c_4} S_3 \tag{A.3}$$

In conventional macroscopic chemistry, we are used to the amount of the reactants and products being interpreted in terms of concentrations, and the constants being rate constants; in mesoscopic kinetics, however, the amounts are molecular copy numbers (the number of molecules of that species in the system) and the constants are *probability rate constants*, which, when multiplied by the number of possible collisions of the reactant molecules, yield reaction propensities.

A common convention is to denote the copy number of species S_1 by s_1, so the propensity of ReactionA.1 may be written $c_1 s_1$, while the propensity of the "forward" (top) Reaction A.2 is $c_2 s_1(s_1 - 1)$, since if two molecules of S_1 collide, they have only $s_1 - 1$ other molecules to collide with.

In many cases, what are described above as propensity "constants" can, in fact, be complex functions of any species in the model, since not all reactions (consider catalysis, for example) are dependent purely on the number of molecules (or, in the macroscopic case, concentrations) of reactants present. See Appendix A.3 for an example of a system with explicit, more complex, propensity functions.

[1] The opposite end from the head, the "tail".

A.2 Decay Dimerisation Reaction

The decay dimerisation (DD) model is non-stiff, and the simplest model system used, with 3 species and 4 reaction channels.

It describes a monomer, S_1 that is both rapidly decaying and dimerising reversibly into an unstable dimer, S_2, which, in turn, slowly isomerises into a stable dimer, S_3.

From [19] and [10].

A.2.1 Reaction Equations

$$S_1 \xrightarrow{c_1} \emptyset$$
$$2S_1 \underset{c_3}{\overset{\frac{c2}{2}}{\rightleftharpoons}} S_2$$
$$S_2 \xrightarrow{c_4} S_3$$

A.2.2 Model Parameters

Parameter	Value
c_1	1.0
c_2	0.002
c_3	0.5
c_4	0.04

Table A.1: Model Parameters for Decay Dimerisation [19, p.15]

A.2.3 Initial Copy Numbers

These are the usual copy numbers used in numerical experiments, but note that s_1 is 10^5 in Section 6.1.1.

Species	Copy Number
s_1	10000
s_2	0
s_3	0

Table A.2: Model Parameters for Decay Dimerisation [19, p.15]

A.3 Metabolite-Enzyme

The metabolite-enzyme (ME) model is a simple, slightly stiff, generic model of 4 species — two metabolites, X and Y, whose production is controlled by corresponding enzymes, E_X and E_Y — with 9 reaction channels.

A.3.1 Reaction Equations

$$\begin{array}{rcl} \emptyset & \xrightarrow{\frac{k_1 e_x}{1+\frac{x}{k_i}}} & X \\ X+Y & \xrightarrow{kxy} & \emptyset \\ Y & \xrightarrow{\mu y} & \emptyset \\ \emptyset & \xrightarrow{\frac{k_4}{1+\frac{y}{k_r}}} & E_Y \\ E_Y & \xrightarrow{\mu e_y} & \emptyset \end{array} \qquad \begin{array}{rcl} \emptyset & \xrightarrow{\frac{k_2 e_y}{1+\frac{y}{k_i}}} & Y \\ X & \xrightarrow{\mu x} & \emptyset \\ \emptyset & \xrightarrow{\frac{k_3}{1+\frac{x}{k_r}}} & E_X \\ E_X & \xrightarrow{\mu e_x} & \emptyset \end{array}$$

A.3.2 Model Parameters

Parameter	Value
k	0.001
k_i	0.001
k_r	30
k_1	0.3
k_2	0.3
k_3	0.02
k_4	0.02
μ	0.002

Table A.3: Model Parameters for Metabolite-Enzyme

A.3.3 Initial Copy Numbers

Species	Copy Number	Description
X	10	Metabolite X
Y	10	Metabolite Y
e_x	10	Enzyme X
e_y	10	Enzyme Y

Table A.4: Initial Conditions for Metabolite-Enzyme

A.4 *E. Coli* Heat-shock Reaction

The heat-shock reaction of *E. coli* (HSR) is a very stiff system of 28 species and 61 reaction channels which models the response of *E. coli* to heat stress. At elevated temperatures, proteins begin to denature; the response is the activation of several genes that produce chaperone enzymes, some of which help to refold denaturing proteins, whilst others help to degrade denatured proteins [24, 2].

A.4.1 Reaction Equations

$$
\begin{aligned}
S_1 + S_2 &\underset{c_2}{\overset{c_1}{\rightleftharpoons}} S_3 \\
S_1 + S_4 &\underset{c_4}{\overset{c_3}{\rightleftharpoons}} S_5 \\
S_1 + S_6 &\underset{c_6}{\overset{c_5}{\rightleftharpoons}} S_7 \\
S_4 + S_{14} &\underset{c_8}{\overset{c_7}{\rightleftharpoons}} S_{15} \\
S_{14} + S_{16} &\underset{c_{10}}{\overset{c_9}{\rightleftharpoons}} S_{17} \\
S_3 + S_6 &\underset{c_{12}}{\overset{c_{11}}{\rightleftharpoons}} S_8 \\
S_5 + S_6 &\underset{c_{14}}{\overset{c_{13}}{\rightleftharpoons}} S_9 \\
S_3 + S_{10} &\underset{c_{16}}{\overset{c_{15}}{\rightleftharpoons}} S_{12} \\
S_5 + S_{11} &\underset{c_{18}}{\overset{c_{17}}{\rightleftharpoons}} S_{13} \\
S_{15} + S_{18} &\underset{c_{20}}{\overset{c_{19}}{\rightleftharpoons}} S_{19} \\
\emptyset &\underset{c_{22}}{\overset{c_{21}s_{13}}{\rightleftharpoons}} S_{22} \\
\emptyset &\underset{c_{24}}{\overset{c_{23}s_{22}}{\rightleftharpoons}} S_{14} \\
S_{17} &\overset{c_{25}}{\rightarrow} S_{16} \\
S_{15} &\overset{c_{26}}{\rightarrow} S_4 \\
S_{19} &\overset{c_{27}}{\rightarrow} S_4 + S_{18} \\
S_{28} &\overset{c_{28}}{\rightarrow} S_4 + S_{27} \\
\emptyset &\underset{c_{30}}{\overset{c_{29}s_{13}}{\rightleftharpoons}} S_{23} \\
\emptyset &\underset{c_{32}}{\overset{c_{31}s_{23}}{\rightleftharpoons}} S_{18} \\
S_{19} &\overset{c_{33}}{\rightarrow} S_{15}
\end{aligned}
$$

$$\emptyset \underset{c_{35}}{\overset{c_{34}s_{12}}{\rightleftharpoons}} S_{25}$$
$$\emptyset \underset{c_{37}}{\overset{c_{36}s_{25}}{\rightleftharpoons}} S_4$$
$$S_{19} \overset{c_{38}}{\rightarrow} S_{14} + S_{18}$$
$$S_{21} \overset{c_{39}}{\rightarrow} S_{20}$$
$$S_{28} \overset{c_{40}}{\rightarrow} S_{14} + S_{27}$$
$$\emptyset \underset{c_{42}}{\overset{c_{41}s_{13}}{\rightleftharpoons}} S_{24}$$
$$\emptyset \underset{c_{44}}{\overset{c_{43}s_{24}}{\rightleftharpoons}} S_{20}$$
$$S_{21} \overset{c_{45}}{\rightarrow} S_4$$
$$S_4 + S_{20} \underset{c_{47}}{\overset{c_{46}}{\rightleftharpoons}} S_{21}$$
$$\emptyset \underset{c_{49}}{\overset{c_{48}s_{13}}{\rightleftharpoons}} S_{26}$$
$$\emptyset \underset{c_{51}}{\overset{c_{50}s_{26}}{\rightleftharpoons}} S_{27}$$
$$S_{28} \overset{c_{52}}{\rightarrow} S_{15}$$
$$S_{15} + S_{27} \underset{c_{54}}{\overset{c_{53}}{\rightleftharpoons}} S_{28}$$
$$S_5 \overset{c_{55}}{\rightarrow} S_1$$
$$S_{13} \overset{c_{56}}{\rightarrow} S_1 + S_{11}$$
$$S_9 \overset{c_{57}}{\rightarrow} S_7$$
$$S_{15} \overset{c_{58}}{\rightarrow} S_{14}$$
$$S_{19} \overset{c_{59}}{\rightarrow} S_{14} + S_{18}$$
$$S_{20} \overset{c_{60}}{\rightarrow} S_{14} + S_{27}$$
$$S_{21} \overset{c_{61}}{\rightarrow} S_{20}$$

A.4.2 Model Parameters

Parameter	Value	Parameter	Value
c_1	2.54	c_2	1
c_3	0.254	c_4	1
c_5	0.0254	c_6	10
c_7	254	c_8	10000
c_9	0.000254	c_{10}	0.01
c_{11}	0.000254	c_{12}	1
c_{13}	0.000254	c_{14}	1
c_{15}	2.54	c_{16}	1
c_{17}	2540	c_{18}	1000
c_{19}	0.0254	c_{20}	1
c_{21}	6.62	c_{22}	0.5
c_{23}	20	c_{24}	0.03
c_{25}	0.03	c_{26}	0.03
c_{27}	0.03	c_{28}	0.03
c_{29}	1.67	c_{30}	0.5
c_{31}	20	c_{32}	0.03
c_{33}	0.03	c_{34}	0.00625
c_{35}	0.5	c_{36}	7
c_{37}	0.03	c_{38}	3
c_{39}	0.7	c_{40}	0.5
c_{41}	1	c_{42}	0.5
c_{43}	20	c_{44}	0.03
c_{45}	0.03	c_{46}	2.54
c_{47}	10000	c_{48}	0.43333
c_{49}	0.5	c_{50}	20
c_{51}	0.03	c_{52}	0.03
c_{53}	2.54	c_{54}	10000
c_{55}	0.03	c_{56}	0.03
c_{57}	0.03	c_{58}	0.03
c_{59}	0.03	c_{60}	0.03
c_{61}	0.03		

Table A.6: Model Parameters for *E. Coli* Heat-shock Reaction

A.4.3 Initial Copy Numbers

Species	Copy Number
s_1	0
s_2	0
s_3	0
s_4	0
s_5	1
s_6	4645670
s_7	1324
s_8	80
s_9	16
s_{10}	3413
s_{11}	29
s_{12}	584
s_{13}	1
s_{14}	22
s_{15}	0
s_{16}	171440
s_{17}	9150
s_{18}	2280
s_{19}	6
s_{20}	596
s_{21}	0
s_{22}	13
s_{23}	3
s_{24}	3
s_{25}	7
s_{26}	0
s_{27}	260
s_{28}	0

Table A.8: Initial Conditions for *E. Coli* Heat-shock Reaction

A.5 Circadian Rhythm

The circadian rhythm is a well-known cellular phenomenon, also known as the "biological clock", and modeled by the Vilar oscillator [27][1]. The version of the Vilar oscillator used here it is a system of 9 species and 16 reaction channels.

A.5.1 Reaction Equations

$$
\begin{array}{rcl@{\qquad}rcl}
A + R & \xrightarrow{ar\gamma_C} & C & A & \xrightarrow{a\delta_A} & \emptyset \\
C & \xrightarrow{c\delta_A} & R & R & \xrightarrow{r\delta_R} & \emptyset \\
A + D_A & \xrightarrow{ad_A\gamma_A} & D'_A & D'_A & \xrightarrow{d'_A\theta_A} & A + D_A \\
\emptyset & \xrightarrow{d_A\alpha_A} & M_A & \emptyset & \xrightarrow{d'_A\alpha'_A} & M_A \\
M_A & \xrightarrow{m_A\delta_{M_A}} & \emptyset & \emptyset & \xrightarrow{m_A\beta_A} & A \\
A + D_R & \xrightarrow{ad_R\gamma_R} & D'_R & D'_R & \xrightarrow{d'_R\theta_R} & A + D_R \\
\emptyset & \xrightarrow{d_R\alpha_R} & M_R & \emptyset & \xrightarrow{d'_R\alpha'_R} & M_R \\
M_R & \xrightarrow{m_R\delta_{M_R}} & \emptyset & \emptyset & \xrightarrow{m_R\beta_R} & R
\end{array}
$$

A.5.2 Model Parameters

Parameter	Value
γ_C	2.0
δ_A	1
δ_R	0.2
γ_A	1
θ_A	50
α_A	50
α'_A	500
δ_{M_A}	10
β_A	50
γ_R	1
θ_R	100
α_R	0.01
α'_R	50
δ_{M_R}	0.5
β_R	5

Table A.9: Model Parameters for Circadian Rhythm

A.5.3 Initial Copy Numbers

Species	Copy Number	Description
D_a	10	Gene A
D'_a	10	Gene A with bound activator
M_a	10	mRNA A
D_r	10	Gene R
D'_r	10	Gene R with bound activator
M_r	10	mRNA R
C	10	Complex C
A	10	Activator A
R	10	Repressor R

Table A.10: Initial Conditions for Circadian Rhythm

B. CellMC User Guide

B.1 Overview

CellMC is a set of *XSL-T* stylesheets, `C` source code, and a wrapper around *gcc* which converts an SSA model, expressed as an SBML file, into optimised `C` code and thence into an executable for the host platform. *CellMC* does not support cross-compilation.

For each platform, the default behaviour is that which was found to be fastest in testing, although some alternatives exist for debugging and tuning.

B.2 Command-line Options, Switches, and Flags

The syntax below follows the common convention that optional arguments to an option are shown in braces, `[...]`, and mandatory arguments in angle-brackets, `<...>`; within these arguments, free-form strings are indicated by a suggestive label in italics; and strict alternatives are shown between vertical bars, read as "or", with the default shown in italics. If a command-line option lacks such an indication of the format of its argument, then it is a flag taking no argument.

Command-line arguments have, at least, a long form with a leading double-hyphen, e.g. `--long-option-name`, and, often, also a short form with a single hyphen, e.g. `-l`. Short option flags may be globbed, e.g. `-mv` is equivalent to `-m -v`. It is an error to begin a long option with a single hyphen. Options marked with an asterisk are experimental or developmental features which may produce unexpected results on some or all platforms.

Options that modify the runtime behaviour of *CellMC* itself without affecting the generated program (all platforms):

`-h|--help` print usage summary and exit.

`-V|--version` print *CellMC* version and exit.

`-o|--output <`*filename*`>` default `a.out` (as *gcc*).

`-v|--verbose` make *CellMC* more verbose; also passed to *gcc*, which produces *a lot* of output.

`--save-temps` save temporary files; also passed to *gcc*.

`--xslfile <`*filename*`>` override internal choice of XSL-T file ∗

`--no-valid` don't validate SBML model; useful for some old versions of *libxml2* that have error-prone validation.

Options that affect the generated program:

`-p|--profile` generate a profiling (Pass 1) binary.

`-d|--double` use double-precision (default single) ∗

`-g|--gcc-debug [`*label*`]` pass `-g` flag (include debugging symbols) to *gcc* (implies `--no-strip`)

`-O|--gcc-optim [0|1|2|3|s]` specify `-O` flag (optimisation level) to *gcc*.

`--no-strip` don't strip executable; by default, executables are stripped to reduce size.

IA32-specific options (all affect generated program):

`-m|--multicore` generate *pthreads* code for multi-core PCs.

`-l|--log <`*asm*`|lib|fpu>` select `log()` implementation. ∗

`-p|--lpr <none|`*semi*`|full>` select LPR method. ∗

`-r|--rng <`*rsmt*`|stdlib>` select PRNG implementation. ∗

`--march <`*gcc-march-label*`>` pass machine architecture label to *gcc*.

***Cell/BE* options (all affect generated program):**

`-M|--mpi` generate MPI code for a cluster (requires MPI to be installed).

`-s|--sso` turn SIMD slot optimization on (default off) ∗

B.3 Output Metadata Description

It was found during development that it was extremely easy to lose track, or simply forget, exactly what version of models and what version of, and compilations flags for, *CellMC* had been used. Accordingly, *CellMC* records extensive metadata about the simulation including the version information, options, flags, and command-lines used to invoke both *CellMC* and the executable model. To facilitate speed comparisons, data about timing and trajectories is also included. Every line of metadata begins with an octothorpe[1]; raw data can be extracted with a simple shell command, e.g.:

```
grep -v '^#' foo.in > foo.out
```

or

[1] Also known as the "hash", "sharp", or "pound" symbol.

```
sed -n '/^[^#]/,$p' foo.in > foo.out
```

The metadata is intended to be self-explanatory, so only a few fields require explanation. Line numbers refer to the example header following the descriptions.

The model label (line 7) is extracted from the SBML file compiled to produce the program. If the model has an `annotation` element including a `<svn:id>` element (see the examples in the distribution), its contents are used after '$' characters are converted to '%'s (to prevent the information being lost if the results are themselves stored in Subversion); otherwise, the `name` attribute of the `model` element is used.

Often, the number of trajectories calculated is more than the number requested because the number is rounded up (to 4 times the number of cores) for simpler load division. If the number of trajectories calculated differs from the number requested, that is recorded in an additional line (after 19).

Because of SIMD-isation, there are always more reactions executed (absolute total, line 23) than actually contribute to a trajectory (contributing total, line 25) because execution continues in all 4 SIMD slots even when some have reached their final times. The absolute and effective speeds (lines 24 and 26) are based on the absolute and contributing reaction counts respectively. The number of reactions per trajectory (27), a simple average included as a check, is based on the number of contributing reactions and the number of trajectories computed. The reaction simulation overhead (28) is the percentage of the absolute total reactions that are non-contributing.

```
# Simulation start time             = 2009-04-15 07:53:08
# Simulation end time               = 2009-04-15 07:53:09
# Executable name                   = a.out
# Executable invoked with command   = ./foo -o hsr-opt.out 1000 1000
# Built with CellMC version         = 0.2.5
# Executable built with command     = ./cellmc me-opt.xml
# Model label                       = %Id: me-opt.xml 39 ... %
# FP precision                      = single
# log() implementation              = asm
# PRNG                              = rsmt
# Propensity recalculation limiting = semi
# SIMD slot optimization            = on
# Profiling                         = off
# Built with multi-threading        = off
# Built with MPI support            = (not available)
# PRNG master seed                  = 0xedaec6b1
# Number of reactions in model      = 9
# Number of species in model        = 4
# Number of trajectories requested  = 1000
# Simulated time                    = 1000.000000
# Elapsed simulation time (seconds) = 1.042
# Effective time per trajectory (s) = 0.001s
# Absolute total reactions executed = 3984032
# Absolute simulation speed (Rps)   = 3824912
# Contributing total reactions      = 3979412
# Effective simulation speed (Rps)  = 3820476
# Reactions per trajectory (RpT)    = 3979
# Reaction sim. overhead (%)        = 0.12
```

B.4 Example Operation

Compiling from an SBML model in `me-opt.xml` with *CellMC*:

```
emmet@esprit:~/WIP/cellmc/src$ ./cellmc -o me me-opt.xml
emmet@esprit:~/WIP/cellmc/src$
```

Compiling an unordered model in `hsr.xml` with *CellMC*:

```
emmet@esprit:~/WIP/cellmc/src$ ./cellmc -po phsr hsr.xml
emmet@esprit:~/WIP/cellmc/src$ ./phsr -o hsr.xsl 4 50
emmet@esprit:~/WIP/cellmc/src$ xsltproc -o hsr-opt.xml hsr.xsl hsr.xml
emmet@esprit:~/WIP/cellmc/src$ ./cellmc -mo hsr hsr-opt.xml
emmet@esprit:~/WIP/cellmc/src$
```

The executables produced by *CellMC* also produce help when invoked:

```
emmet@esprit:~/WIP/cellmc/src$ ./hsr
USAGE:
    hsr [options] <number of trajectories> <stop time>
OPTIONS:
    -h           shows this help text.
    -o <file>    print output to file (instead of stdout).
    -s <value>   sets PRNG seed to 'value'.
    -c <n>       use 'n' compute threads (mnemonic: 'c'='cores').
    -i           suppress info header in results (mnemonic: '-i'='info ...

emmet@esprit:~/WIP/cellmc/src$
```

CellMC invoked without arguments produces help:

```
emmet@esprit:~/WIP/cellmc/src$ ./cellmc
USAGE:
  cellmc [options] sbmlfile

OPTIONS:
  Options that affect cellmc behaviour, but don't affect the
  generated program
    -h|--help                     show this help text
    -V|--version                  show version information and exit
    -v|--verbose                  make cellmc more verbose
    -o|--output <filename>        set output filename (default: 'a.out')
    --xslfile <filename>          override internal choice of XSL-T file
    --save-temps                  save temporary files (as gcc)
    --no-validation               don't validate SBML model

  Options that affect the generated program:
    -d|--double                   use double-precision (default: single)
    -p|--profile                  generate profiling code
    -g|--gcc-debug [lbl]          pass -g flag (debugging symbols) to gcc
                                      (implies --no-strip)
    -O|--gcc-optim [0|1|2|3|s]    pass -O flag (optimization level) to gcc
    --no-strip                    don't strip executable

  Options specific to this platform (IA32)
    -m|--multicore                generate pthreads code for multiproce ...
    -l|--log <asm|lib|fpu>        select log() implementation (default: ...
    -r|--lpr <none|semi|full>     select LPR method (default: 'semi')
    -n|--rng <rsmt|stdlib>        select PRNG implementation (default:  ...
    --march <gcc-march-label>     pass machine architecture label to gcc

emmet@esprit:~/WIP/cellmc/src$
```