NOTE TO USERS

This reproduction is the best copy available.

UMI®

Biochemical System Simulation on a Heterogeneous Multicore Processor

By

Sevin Al Assaad

Department of Electrical and Computer Engineering

McGill University, Montreal

"A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Engineering"

Copyright © Sevin Al Assaad 2008

January 2009

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-66921-1
Our file Notre référence
ISBN: 978-0-494-66921-1

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Acknowledgements

I would like to thank my thesis supervisor, Prof. Warren J. Gross, for his guidance and support during my work in the Integrated Microsystems Laboratory. Additionally, I would like to thank Nathaniel Azuelos and David Fernandez Becerra for their continuous and generous input in helping me understand the CBE processor, and port the algorithm onto it. Our weekly meetings have proven invaluable, and I would not be here without you. I would also like to thank Laurier Boulianne for designing the original version of GridCell. My project would not exist without your original idea and design.

I would also like extend my deepest gratitude to l'Association Philippe Jabre for their financial support during my undergraduate and graduate studies. Without your generous scholarships, I would not have been able to come to Canada to pursue bigger and better opportunities. You have marked my life in an invaluable way, and I will be forever grateful.

Last but not least, I would like to thank my parents for their continuous support, and for believing in me even when I did not. Without your unconditional love and support, I would not be the person nor the professional I am today.

Thank you all for your immense impact on my life and education. I dedicate this thesis to you.

Abstract

Biological system simulation is an increasingly popular field of study that provides biologists with the tools necessary to simulate biochemical systems in order to obtain quantitative models. The purpose of this thesis is to describe an accelerated version of GridCell, a stochastic biological system simulator. GridCell tracks each individual particle's location in the system, as well as the time evolution of the concentration of each species involved. It simulates molecular diffusion via Brownian movements, and particle interactions are dependent on their locations. We present here a parallel adaptation of the algorithm, implemented on a heterogeneous multicore processor, i.e. IBM Cell Broadband Engine (CBE). We introduce the CBE architecture and outline its advantages, as well as describe the original algorithm. Subsequently, we detail the parallel implementation and the algorithm modifications. Finally, we perform timing analysis to show that the parallel version provides improved performance over the original serial version.

Résumé

L'étude de systèmes biologiques vise à mieux comprendre leur comportement sous différentes conditions biochimiques. La simulation de ces systèmes aide à la création de modèles quantitatifs, ainsi facilitant cette étude. Le but de cette thèse est de présenter une version accélérée de GridCell, un simulateur stochastique de systèmes biologiques. GridCell a l'avantage de suivre la progression de toutes les particules présentes dans le volume simulé, tout en calculant la concentration de chaque espèce impliquée dans la simulation. GridCell permet de simuler les interactions entre les différentes particules, ainsi que leurs mouvements stochastiques. Une adaptation parallèle de GridCell, implémentée sur le Cell Broadband Engine (CBE) de IBM, est ainsi présentée. Tout d'abord, l'architecture du processeur ainsi que ses nombreux avantages sont exposés. Ensuite, les deux versions de GridCell, originaire et parallèle, seront introduites de par la description de l'algorithme qui les gouverne. Cette thèse conclue par une analyse des temps d'exécution démontrant une amélioration de la performance vis-à-vis la version originaire non-parallèle.

Glossary

B:	Byte
b:	bit
CAM:	Computation Acceleration Model
CBE:	Cell Broadband Engine
CBE version:	Version of GridCell implemented on the CBE
DMA:	Direct Memory Access
DMAC:	Direct Memory Access Controller
DPFP:	Double Precision Floating Point
EIB:	Element Interconnect Bus
GUI:	Graphical User Interface
LS:	Local Store
MFC:	Memory Flow Controller
PPE:	PowerPC Processing Element
PPSS:	PowerPC Processor Storage Subsystem
PPU:	PowerPC Processing Unit
SBML:	Systems Biology Mark-up Language
SDFP:	Single Precision Floating Point
Serial-Intel:	Serial implementation, run on an Intel processor
Serial-PPU:	Serial implementation, run on the CBE, using only the PPU
SPE:	Synergic Processing Element
SPU:	Synergic Processing Unit

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction and Background

Computational cell biology is a cross-disciplinary area of research joining together computer simulation technology with cellular biology [1]. The long-term goal of this field is to simulate a biological cell in order to understand the different molecular interactions occurring within it. For example, quantitative models of cancerous stem cells have been found to be very useful in the understanding of cancer dynamics [2]. One system that is often simulated is the Michaelis-Menten system [3], described by the equation: $E + S \leftrightarrow ES \rightarrow E + P$.

S is a substrate that binds to enzyme E to yield ES, which can decompose into the product P and the enzyme E, or into its original form E + S. Usually, the enzyme is the limiting factor, since its concentration is much lower than that of S. The results of the simulation (Figure 1-1) give insights into the time evolution of the concentration of the different species involved.

A large number of computer simulators have been developed in order to study various biochemical systems and determine the time evolution of particle concentrations in those systems [4-24].

1.1 Introduction

GridCell is a stochastic simulator of biochemical reactions in which the volume under test is represented by a 3D grid, and each particle in the system is an independent object that can move and react [25, 26]. GridCell supports both molecular diffusion and particle interactions, and the simulation runs over discrete time-steps.

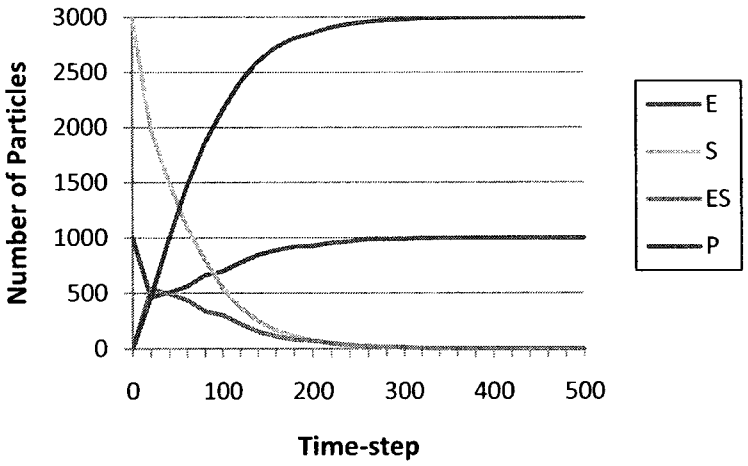


Figure 1-1: Michaelis-Menten System

In this thesis, we present a parallel version of the GridCell algorithm, originally described in [25, 26]. Stochastic simulators are linearly implemented, whereas biological processes occur randomly and concurrently [27], where a particle is dependent only on its immediate neighbourhood, and is not affected by the movement and interactions of particles further away. Therefore, it is possible to divide the simulation environment over multiple processors, and implement GridCell on a multicore system, such as the Cell Broadband Engine (CBE). In the following section, we provide some background information about common biochemical simulators.

1.2 Background

There are two different types of biochemical simulators: deterministic and stochastic. Deterministic simulators solve a system of mathematical equations which describe the biochemical process to be tested as well as the different particle interactions. The equations also describe the evolution of species concentrations. For the same problem, deterministic simulators generate the same solution every time. On the other hand, stochastic simulators generate probabilities and random numbers in an attempt to recreate molecular behaviour. In stochastic simulators, the time evolution to reach a system's steady state is different for each simulation, but the results are comparable.

Some of the most popular simulators are discussed here. Virtual Cell [17, 18, 23], a deterministic simulator, is presented first. Next, Gillespie's Stochastic Simulation Algorithm is introduced because of its importance in the development of other stochastic simulators [11]. Subsequently, seven different stochastic simulators are presented: StochSim [15, 16, 28], MCell [29], SmartCell [4], ChemCell [19], Cell++ [22], MesoRD [12] and the original GridCell program [25, 26].

1.2.1 Virtual Cell

Virtual Cell [17, 18, 23], developed at the University of Connecticut Health Center, is a deterministic simulator based on solving a set of mathematical equations. The model under test is represented by a system of equations in Virtual Cell Mathematics Description Language or VCMDL. VCMDL is a declarative language that was specifically created for Virtual Cell, and used to describe classes of mathematical equations that need to be solved during the simulation [17, 18, 23]. Once the VCMDL model description is obtained, it is then translated into programming code which is sent to the numerical solvers which resolve the mathematical equations.

Virtual Cell is accessible through a graphical interface (GUI) that allows users to specify the characteristics of the biological processes to be tested, such as the type and number of the molecules in the system, the reactions that characterize the interactions between them, as well as their location and spatial topology within cellular compartments. This quantitative data is put together to construct a complex spatial model depicted by mathematical equations, thus obtaining the VCMDL model description. For biologists untrained in physics and mathematics, the GUI provides a simple way of inputting information into the system, and requires no knowledge of the underlying equations. On the other hand, for experienced users with a mathematical background (e.g.: bioengineers and mathematical biologists), the model can also be manually created by directly entering the equations describing the biological process in VCMDL format. In

either case, the model definition is finalized when initial and boundary conditions have been specified. The simulation produces spatial and temporal results comparable to experimental data, such as particle concentrations. Virtual Cell is not suitable for processes with a variable 3D structure such as mitosis. More information on Virtual Cell can be found in [17, 18, 23].

In the next section, we introduce Gillespie's Stochastic Simulation Algorithm.

1.2.2 The Stochastic Simulation Algorithm (SSA)

Gillespie's SSA was created because deterministic approaches, which are based on solving equations, are not suitable in all situations, such as nonlinear or unstable systems. In such cases, the number of equations to solve increases significantly and the equations become too complex. The size of the system also increases with the number of species in the system. Consequently, the computation time and resource utilization considerably increase such that the simulation cannot occur in a reasonable amount of time [11]. Additionally, the mathematical representation of the system does not always include important information regarding the biochemical processes, such as the spatial organization of the volume, or the distribution of the particles within that volume [22].

The SSA was the first stochastic simulator to be created. Stochastic approaches are based on random number generation. Those random numbers are used to determine the way particles interact, based on a list of reactions that describe the process under test. The basic assumption for the SSA is that the system is well-mixed or "spatially homogeneous" [11]: a particle can react with any other particle in the system. In the simulation, there are N_i particles of type S_i, and M reactions R_u with reaction parameters c_u. The SSA algorithm is probability-based, and attempts to simulate the Joint Probability Density Function (PDF) for each reaction, at specific times.

The SSA process is divided into four consecutive steps, as follows:

- The first stage is the initialization stage in which the initial values of all the parameters, including Ni and cu, and the list of chemical reactions, are specified and stored.
- The second stage is the Monte Carlo phase, which consists of generating two random numbers that satisfy the PDF, using Monte Carlo techniques [30] : an index u of the reaction to perform next, and a time increment τ.
- The third step is the update stage where the time is incremented by τ, and the values of the reactants and products involved in R_u are modified as if reaction R_u just executed. For instance, if R_u is the reaction $S_1 + S_2 \rightarrow 2S_3$, the update stage involves decreasing the concentration of S_1 and S_2 by 1, and increasing the copy number of S_3 by 2.
- Finally, in the publish stage, the updated values are output.

The process, starting from the Monte Carlo stage, iterates until the simulation has completed, or there are no more reactants in the system. This algorithm computes the time evolution of all the species in the biological system under test, and does not take into account the spatial localization of the particles or the characteristics of the volume in which the process takes place. In the SSA, molecular diffusion is not supported, and particle localization is not tracked. The SSA is fully described in [11].

1.2.3 SmartCell

SmartCell [4] is a stochastic simulator developed at the European Molecular Biology Laboratory in Heidelberg. It is based on a modified SSA algorithm: the basic SSA hypotheses (e.g.: well-mixed system) are maintained, but molecular diffusion is implemented as an individual event. Intended for the modelling of biological processes, SmartCell supports various cellular geometries and compartments, allowing the user to localize certain species within

membranes. Because of the spatial distribution of molecules, the volume is no longer necessarily well-mixed, which does not adhere to Gillespie's hypothesis for the SSA algorithm. This situation is remedied by dividing the volume into smaller sections in which the molecules are well-mixed; and molecular events take place within those sections.

The model is described using SBML (or Systems Biology Markup Language [31]]) which is an XML language used to create biological models. The SBML file includes all the information regarding the number of molecules in the system as well as the characteristics of the reactions between them (e.g.: constant rate). The molecular entities are defined also defined in the SBML file, as well as the different processes (reaction or diffusion). Each process element is an event which involves at least one entity. Before the simulation begins, a list of entities and processes is generated. Complex reactions (with three or more reactants) are simplified into many simpler reactions with one or two reactants.

For each event (reaction and diffusion) in the system, a probability is calculated. Events are then added into the events queue, and an associated reaction-time is computed for each. The event with the smallest reaction-time is executed, a new probability is calculated for it, and it is added to the queue. If the queue is not empty, the next event to occur is the one with the smallest reaction-time. It is important to note that reaction-times of events in the queue can change when the current event has executed. Thus, after each event has occurred, reaction-times are updated in the queue. Additionally, the reaction time-step varies during the simulation as it is dependent on the fasted reaction at the time.

One drawback of SmartCell is that the computation is very slow for larger systems. Additionally, localization of individual particles is not inherently supported as the algorithm is based on the SSA which represents particles in bulk. Details about SmartCell can be found in [4].

1.2.4 StochSim

StochSim [15, 16], developed at the University of Cambridge for a study on bacterial chemotaxis [32], represents each individual molecule in the system as

an "independent software object". Additionally, StochSim supports a 2D structure in which processes are simulated. Three different structures are available: a square, a triangle, and a hexagon.

Interactions between molecules are simulated based on probabilities pre-computed from the molecular concentration of the reactants and the reaction rates. Complex reactions are achieved through multiple uni- and bi-molecular reactions (one or two reactants). The simulation is done in two-dimensions, and reactions can occur between randomly chosen molecules. One set of random numbers is used to select the reactants; another is compared to the reaction probability to verify if the reaction will occur. It is possible to define reactions that are dependent on neighbouring particles. In this case, the reaction rates depend not only on the state of the selected particle, but also on the states of its neighbours.

During the initialization process, the user must specify the rate of the reactions in the system. In addition, the simulation time-step (defined as the length of time of one iteration) and the reaction probabilities are computed. The simulation then proceeds by executing a small subroutine at every time-step.

The algorithm iterates over discrete time-steps, and the simulation occurs between randomly chosen particles. Once molecules have been arbitrarily chosen to react, a random number R_n is generated and compared to the probability of the reaction. The reaction occurs if R_n is less than the probability of reaction. The fact that the specific location of reacting molecules does not play a role in the reaction is a major drawback in the StochSim algorithm. More information on StochSim can be found in [15, 16].

1.2.5 MCell

In Gillespie's algorithm as well as SmartCell, stochastic simulation is achieved by representing molecules in bulk [4, 11]. The spatial location of the particles is irrelevant to the time evolution of the species concentrations. However, recent research indicates that the spatial localization of particles in a

system plays an important role in the molecular interactions within that system [33]. MCell [29] was created to resolve this situation.

MCell is a simulator of cellular microphysiology, based on Monte Carlo methods. This stochastic tool solves reaction-diffusion problems and analyses molecular events (movements and reactions) within a 3D volume of arbitrary shape. Molecular diffusion is accomplished through random walk movements which reproduce Brownian motion [34] without tracking the absolute molecular displacements. The location of a particle in the 3D volume is not required to be known. However, one must know its relative position with respect to boundaries (e.g. membranes and compartments). This way, molecular diffusion and collision are both simulated without the need for absolute spatial localization. Particles move independently from each other, but are restricted by boundaries.

Reactions are based on individual molecule selection, and the realistic representation of molecular interactions is achieved through probability calculation and random number generation. A molecule is selected, and a random number is generated and compared with known Monte Carlo probabilities to determine if the reaction will occur for the chosen molecule.

The model under test is defined in a simulation file using MDL, MCell's Model Description Language. These MDL files represent the tool's program interface; they include all the necessary information required for model creation and are user-created. Input and output parameters are also specified in the MDL file. The input parameters are divided into two subsets: (1) those that define the biological process to be simulated (e.g.: number of molecules in the system), and (2) those that characterize the simulation (e.g.: time-step).

The simulation starts with an initialization stage in which the simulation environment is set up. Random numbers are pre-generated in this stage as well. The simulation is then run in discrete time-steps during which the stochastic molecular events take place. The program is started using command line calls.

Relative positioning is tracked, and collision only occurs for molecular diffusion. The MCell tool is described in [29].

1.2.6 Cell++

Cell++ [22] is a stochastic simulation environment developed at the University of Toronto. It allows the user to model a variety of different biological processes while accurately simulating small and large molecules with variable molecular concentrations. The volume is represented by a 3D cubic lattice, and cellular compartments are supported with the spatial implementation of membranes. The basic design of the simulator combines Brownian dynamics to accurately represent larger particles, with a cellular automata approach that describes the behaviour of smaller molecules. Discrete components diffuse via random walks restricted by the spatial localization of particles and boundaries (collision), and interactions are handled using a probability-based Monte Carlo approach in which the probability of reaction depends on the components involved.

Although Cell++ is a stochastic simulator, a deterministic set of equations must be solved in order to handle the diffusion of smaller molecules, thus allowing molecules to move into adjacent sites. Additionally, the concentration changes are described by another set of equations.

Cell++ is based on an iterative algorithm that advances through discrete time-steps. The simulator provides the user with an interactive graphical interface to manipulate and control the simulation. Modifying the input files and the source code allows for new systems to be simulated. More information about Cell++ is found in [22].

1.2.7 ChemCell

ChemCell [19] was developed at Sandia National Laboratories to model the stochastic behaviour of prokaryotic cells such as microbes. The algorithm implements molecular diffusion using Brownian motion and the particles react with each other based on the constant reaction rates. The cell is modelled by a geometric volume, and each molecule is implemented as one "particle" in the system with its own (x,y,z) coordinates and type. Compartments, such as the

nucleus, are inherently supported through internal boundaries, and each compartment has its own external boundary. During the simulation, particles diffuse and react with other neighbouring particles. However, collision is supported such that particle diffusion is limited by compartment boundaries and other particles. Reactions occur by computing probabilities and comparing them to known reaction probabilities stored in the system. Whether the reaction occurs or not is based on the comparison results.

The first part of the simulation consists of setting up the model by reading an input file with a list of commands. In this phase, the geometry of the model (e.g.: compartment creation, topology and size) is defined, as well as the characteristics of the biological process (e.g.: particle species and list of chemical reactions). The next phase is referred to as the "timestepping" stage in which particles move and react with each other. Movement is done following Brownian motion rules. In order for particles to react with each other, they have to be neighbours. To find neighbouring particles, a binning algorithm is used: a reaction can only occur between two particles in the same or neighbouring bins. Once a pair is selected, random numbers are generated to verify if the reaction will occur. At the end of the timestepping stage, simulation outputs can be generated. More information on ChemCell can be found in [19].

1.2.8 MesoRD

MesoRD [12] was developed at the Uppsala University in Sweden. This stochastic simulation tool attempts to solve the Reaction-Diffusion Master Equation (RDME) which describes the Markov Process. In probability theory, the Markov Process illustrates a probability distribution in which the current state is independent from past states [35].

In order to solve the RDME, other simulators such as the Next Subvolume Method [36] have been developed in which the system volume is divided into smaller subvolumes characterized by the concentration of each species found in that subvolume. For 3D geometries, a large number of subvolumes are required

for accurate analysis, thus increasing the state-space dimension of the RDME which raises the execution time. MesoRD was implemented in order to achieve improved performance.

The biological system to be modeled is described in an SBML file which is read at run-time. The SBML input file includes such information as the reaction rates, geometry and diffusion information, as well as the species present in the system and their initial concentrations. Other parameters, such as the simulation settings and visualization options, are also specified during the initialization stage through either the user interface for Windows users, or the command line for UNIX users. Chemical reactions and molecular diffusion are handled within each subvolume, and their rates depend on the amount of each species found in that subvolume. Once the simulation starts, three threads divide the work: (1) a simulation thread handles molecular events (reactions and diffusion), (2) a visualization thread runs the 3D viewer, and (3) a status thread displays the results of the simulation. More information about MesoRD can be found in [12].

1.2.9 Original GridCell

GridCell is a biological system simulator that represents the simulation environment with a 3D cubic grid comprised of discrete voxels [25, 26]. The system under test is modeled in an SBML file, and the program supports both biochemical reactions and molecular diffusion.

Particles move and react based on probabilities computed based on the characteristics of the molecules in the system under test: random numbers are generated that determine if a particle will move and/or react, and which reaction will occur. Additionally, GridCell keeps track of particle localization since each particle is an independent object that has its own 3D coordinates in the simulation volume. Furthermore, since each discrete voxel holds up to one particle at a time, molecular diffusion is constricted by particle locations, thus taking molecular crowding into effect. The algorithm is described in more detail in [25, 26] and in Section 3.1.

1.3 Summary

The basis behind stochastic simulators is that mathematical equations cannot accurately represent all biological processes. For nonlinear systems that are chemically unstable, it is not possible to model the molecular behaviour realistically [11]. In addition, deterministic approaches do not always take into account the spatial localization of the particles in the system, which has been shown to play an important role in molecular interactions [33]. Furthermore, most stochastic simulators model biological systems without including the effects of localization and/or particle collision. All those simulators implement molecular diffusion through random walk movements following Brownian rules. On the other hand, molecular interactions are handled very differently from one simulator to the other. Table 1-1 summarises the various simulators just presented.

1.4 Motivation

By studying similar work done in the field, one can understand the motivation behind GridCell. Firstly, GridCell represents the volume as a 3D grid in which the well-mixed assumption does not necessarily hold. As we will see however, we can still simulate well-mixed systems with GridCell. This allows for the simulation of different types of processes. Secondly, the choice of the reaction to perform is dependent on the particle chosen and its neighbouring molecules. Thus, reactions do not occur between randomly-chosen particles. The spatial localization of particles plays an important role on both diffusion and molecular interactions.

Since biochemical reactions occur randomly and concurrently within a biological system, each particle is dependent only on its immediate neighbourhood. In this thesis, we take advantage of this characteristic of biochemical reactions to demonstrate the use of a multicore parallel architecture, namely the Cell Broadband Engine (CBE), to implement the GridCell algorithm.

Table 1-1: Summary of simulators

Simulator	Type	Summary
Cell++ [22]	Stochastic	- Hybrid approach - Uses equations for small molecule diffusion and concentration changes - Time- and spatial-evolution
ChemCell [19]	Stochastic	- 3D volume - Time- and spatial-evolution - Collision is supported - Reactions based on probability calculations
GridCell [25, 26]	Stochastic	- Model described in SBML - Reaction and diffusion occur in a 3D grid comprised of discrete voxels - Molecular crowding and localization taken into account
MCell [29]	Stochastic	- Uses Monte Carlo methods - Molecular diffusion is implemented - Relative positioning is tracked - Collision only for diffusion
MesoRD [12]	Stochastic	- Model described in SBML - Reaction and diffusion occur in subvolumes - Reactions based on probability calculations
SmartCell [4]	Stochastic	- Model defined in SBML - Modified SSA: diffusion supported - Well-mixed subvolumes (total volume not necessarily well-mixed) where reaction/diffusion occur - Particles represented in bulk
SSA [11]	Stochastic	- Molecules represented in bulk - Time-evolution only (no localization) - Well-mixed volumes only - Reactions/particles randomly selected - Diffusion not supported.
StochSim [15, 16]	Stochastic	- Particles as independent software objects - 2D volume structure - Randomly chosen particles - Time-evolution only
Virtual Cell [17, 18, 23]	Deterministic	- System of equations - Spatial simulation is possible (equations can include info about spatial structure)

In the following chapters, we discuss the GridCell algorithms as well as the serial and CBE implementations, and analyse their performance on various platforms. In Chapter 2, we describe the Cell Broadband Engine (CBE), as well as

the important software and hardware tools that we use in the implementation of GridCell. In Chapter 3, we present the GridCell algorithm, starting with the serial version, and followed by the CBE adaptation. Results verification and performance analysis are carried out in Chapter 4. We conclude this thesis in Chapter 5 with an outline of potential future work.

Chapter 2 The Cell Broadband Engine

The Cell Broadband Engine (CBE) [37] is a heterogeneous multicore processor designed originally for the Sony PlayStation 3© (PS3) [38] gaming platform. A heterogeneous processor is comprised of processing elements with different architectures. It has since been used as a high-performance computing tool because of its highly-parallel structure. In this chapter, we present the architecture of the CBE, as well as the important hardware and software tools that are used in GridCell.

2.1 Architecture

The CBE is comprised of two different types of processing elements connected together and to main memory through the Element Interconnect Bus (EIB). Figure 2-1 displays a high-level block-diagram of a cell processor. The PowerPC Processing Element (PPE) is a PowerPC processor with a standard PowerPC instruction set, and is described in Section 2.1.1.

Currently, each CBE has eight Synergic Processor Elements (SPE) which are specialized highly-parallel processors (see Section 2.1.2). However, on the PS3, either one or two SPEs are disabled to improve yield [38], and thus cannot be used.

2.1.1 PowerPC Processing Element

The PPE is comprised of a PowerPC Processing Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS). The PPU is a general-purpose 64-

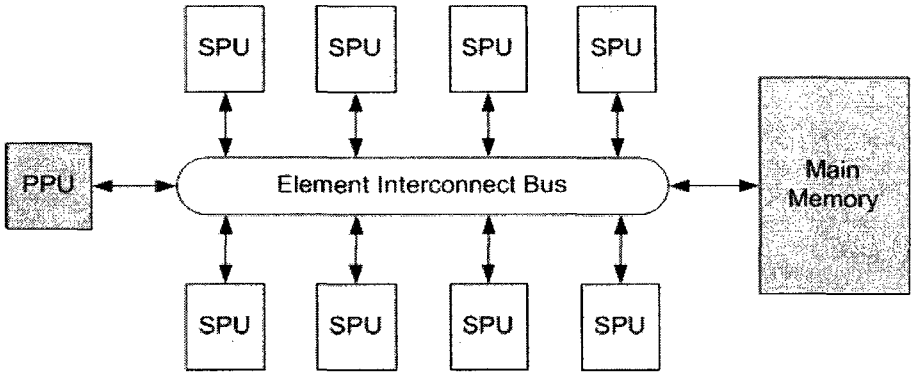


Figure 2-1: CBE High Level Block Diagram [37]

bit (b) processor. It is a multi-threaded core with separate L1 instruction and data caches (128-byte (B) lines). The PPU register file has three types of data registers: (1) 32 64b General Purpose Registers, (2) 32 64b Floating Point Registers, and (3) 32 128b vector Registers for single-instruction-multiple-data (SIMD) processing. Aside from the data registers, the register file also holds control and error registers.

Two execution units, the Fixed-Point Unit and the Vector/Scalar Unit, perform fixed-point (integer) operations, and floating-point and vector (SIMD) operations respectively. SIMD operations are possible on the PPU through the vector/SIMD multimedia extension instructions. Additionally, memory transfers involving the PPU are managed by the Memory Management Unit, which is responsible for all address translation.

The PPSS is an L2 unified cache through which communication with the EIB is possible. Like the L1 caches, L2 cache lines are 128B. The PPSS carries out requests coming from the PPU, or to the PPU from the SPUs and other devices. The PPU communicates with the PPSS through loads (32B) and stores (16B), whereas data transfers between the PPSS and the EIB occur on 16B load and store buses. Figure 2-2 shows a high-lever block diagram of the PPE.

2.1.2 Synergistic Processor Elements

Each SPE is a combination of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). The SPU is a SIMD core processor with a

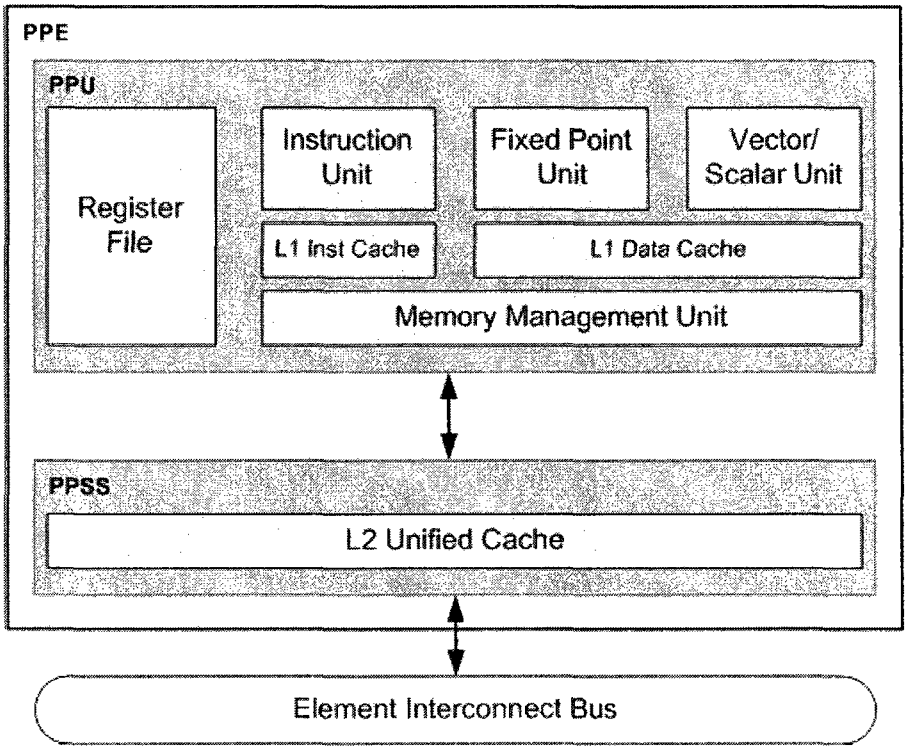


Figure 2-2: PowerPC Processor Element [37]

unified vector register file holding 128 128b vector registers. Those are General Purpose Registers that can be used for fixed or floating-point storage.

The SPU stores both instructions and data in its Local Store (LS), which is only 256KB, making memory management on the CBE a crucial task. The SPE communicates with main memory, the PPE and other SPEs through its MFC (see Section 2.2).

Unlike the PPU, the SPU does no memory address translation for LS addresses. However, through effective-addressing, software on the PPE and other SPEs can access the LS in main memory. The SPE must use Direct Memory Access (DMA) transfers to move data from main memory to the LS, and vice versa, in order to access the data.

2.1.2.1 Limitations

Floating-Point (FP) operations are supported on the SPU. A 128b vector holds four single-precision (SP) FP values, and SPFP operations are executed in SIMD, i.e.: four operations at a time. However double-precision (DP) instructions

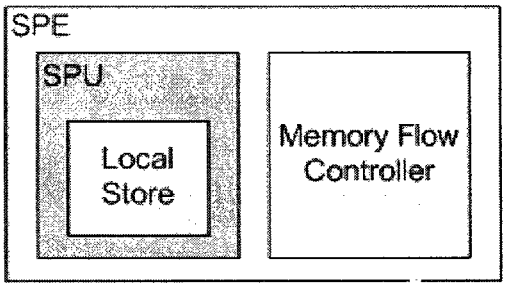


Figure 2-3: Synergic Processor Element [37]

are not executed in parallel. Vectors are broken down into scalar values prior to the execution, increasing the execution time making DPFP operations inefficient. A 128b vector holds two DPFP values, and DPFP instructions execute on one DPFP value at a time. SPFP operations are much more efficient on the current version of the CBE because the PS3 gaming platform did not require double-precision execution.

Since the SPU is a SIMD processor, all instructions are executed on 128b vectors. Scalar code can be written for the SPU; however, scalar values are stored in and must be extracted from vectors prior to execution since there are no scalar registers. This extraction increases the execution time, and reduces the efficiency of scalar code. It is preferable to store scalar values in vectors, and perform vector operations instead.

The SPU's performance is also constrained by branches and conditional statements because of the linear instruction flow of the SPU. A branch instruction can disrupt the sequential execution, and can have a very high penalty. When a branch is not taken, it causes an 18 or 19-cycle penalty, which is over twice the average SPU latency (maximum 7 cycles), resulting in degraded performance.

Although it is not always possible to eliminate all conditional statements in a piece of code, there are means to reduce their effect. Function-inlining can be used to eliminate the branch caused by an instruction call, but excessive inlining increases the size of the code, thus reducing the effective space for data. Secondly, loop branches can be reduced or eliminated by unrolling the loops, although loop branches are highly predictable, and for a large index, the penalty becomes negligible. Another approach is to use program-based branch prediction. Program-based prediction is based on program constructs where heuristics determine how

Table 2-1: Sample Target Applications

Some Accelerated Functions	Sample Target Applications
Signal Processing	Medical Imagine/Visualization
Image Processing	Drug Discovery
Matrix Mathematics	Training Simulations
Vector Mathematics	Secure Communications
Physics Simulations	Digital Content Creation/Distribution
Encryption/Decryption	Computational Chemistry
Pattern Matching	Voice and Patter Recognition
Parallel Processing	Network Processing
Real Time Processing	Climate Modeling

to predict the branch associated with those constructs. For example, unconditional branches should always be predicted as taken. The advantage of this technique is that a misprediction penalty is small compared to the penalty incurred without prediction, or to the computation time [37].

The SPU's limitations dictate what type of applications can be run on the CBE. The CBE is targeted towards applications that are highly-parallel and highly-repetitive, with very little branching, such as biochemical simulations. Applications such as signal and image processing can be very easily and effectively accelerated on the CBE. Table 2-1 list some target applications that can be run on the CBE [39].

2.1.3 Element Interconnect Bus

The Element Interconnect Bus (EIB) connects all the elements on the CBE: main memory, the PPE and all the SPEs. It consists of four unidirectional 16B data rings (two clockwise, and two counter-clockwise). Each ring accommodates 128B transfers (equivalent to one PPE cache line). The EIB's maximum internal bandwidth is 96B per cycle and can handle over 100 outstanding memory transfers.

The EIB supports memory-coherence, and thus the CBE is designed to be fully coherent with other CBEs in a system.

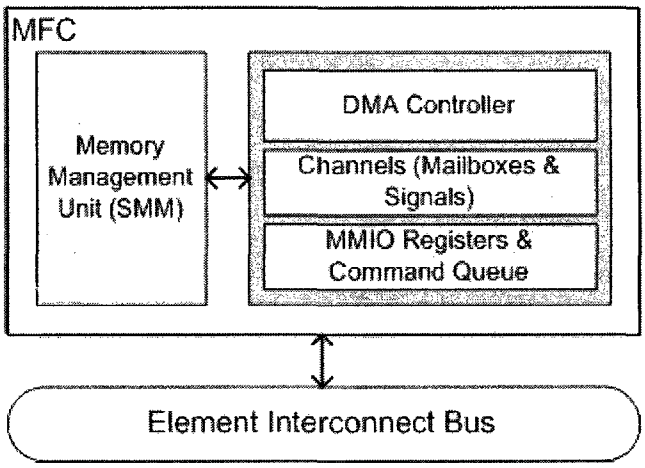


Figure 2-4: Memory-Flow Controller [37]

2.2 Memory Flow Controller

The MFC is the SPU's main interface to main memory and other processors and devices, through the EIB. The MFC's most important role is to allow communication between the SPU's LS and main memory. This is done through DMA transfers by means of the DMA controller. The MFC also provides other functionalities to the SPU, such as synchronization between the LS and main memory, as well as other communication features such as Mailboxes and Signals. The MFC block diagram is shown in Figure 2-4. The figure shows the different elements of the MFC that are discussed in the following Sections.

The MFC provides the SPU with different mechanisms that allow it to communicate explicitly with other processors and devices: (1) DMA Transfers, (2) Mailbox messages, and (3) Signal Notification messages.

2.2.1 Direct Memory Access Controller

The Direct Memory Access Controller (DMAC) allows transfers between main memory and the SPU's LS. The DMAC executes the memory transfer commands in parallel with the SPU execution, thus preventing the SPU from being bottlenecked by the DMA commands. It also allows the SPU to pre-fetch data while executing, therefore allowing multi-buffering on the SPU. The controller can issue up to 16 independent transfers to and from the LS.

There are two types of DMA transfers: (1) single transfers, and (2) list transfers. Single transfers allow the DMAC to fetch one block of consecutive data from main memory to LS. On the other hand, to fetch data in non-consecutive blocks in main memory, DMA list transfers can be used. A DMA list is a list of transfer elements where each element consisting of a transfer size and an effective address in main memory. This allows fetching non-consecutive blocks in main memory and storing them consecutively on the LS, using a single DMA list command.

The size of one DMA transfer, whether it is a single transfer or as part of a DMA list, cannot exceed 16KB. In addition, a DMA list can hold up to 2048 transfer elements. Currently, 2048 transfer elements of 16KB each exceed the size of the SPU LS, but this flexibility allows the size of the LS to increase in future versions of the CBE, without modifying the DMA list construct. Another constraint of DMA transfers relates to the size and alignment in main memory of the data being transferred. First, the data size must be 1B, 2B, 4B, 8B or a multiple of 16B. Second, the data must be aligned on a 128B boundary. Failure to do so will result in bus errors.

In addition to the DMAC, the MFC has a channel facility referred to as "Tag-Group Completion Facility". This facility provides memory synchronization between main memory and LS. Each DMA command is tagged with a 5b identifier that allows several commands to be grouped together, and each tag is used to determine when a command or group of commands has finished executing. Additionally, since the DMA commands are executing in parallel with the SPU operation, tags allow for "blocking" transfers such that execution on the SPU halts until the DMA commands have finished executing.

2.2.2 Mailboxes

Each SPU has three 32b mailbox queues: (1) outbound mailbox queue, (2) outbound interrupt mailbox queue, and (3) inbound mailbox queue. The mailbox facility manages communication through three mailbox channels that control the

queues: (1) two outbound one-entry channels: the SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox, and (2) one four-entry inbound channel: the SPU Read Inbound Mailbox. Mailboxes were intended to send short messages, such as program status and flags, between the SPE and the PPE. However, they can be used for sending 32b data of any type.

The inbound mailbox queue is a FIFO queue of four entries, and allows other processors and devices to send messages to the SPE. If messages are being read at a slower rate than they are being received, the fourth entry is always overwritten with the latest message to arrive. On the other hand, the outbound queues can hold up to one entry only.

Mailbox operations (read channel, or write channel) are, by default, blocking operations on the SPE: writing to a full location, or attempting to read an empty location will cause the SPE to stall until the mailbox operation can be executed. This is a very useful tool because it can allow the PPE to control the execution on the SPE, and vice versa. However, it is possible to have non-blocking operations, by setting flags in the mailbox operation. In the CBE implementation of GridCell, the PPE and SPE communicate via mailbox messages to synchronize the execution, and make sure that the data is ready (therefore assuring data integrity on both processors).

2.2.3 SPU signal notification

The Signal-Notification Channel facility in the MFC allows the SPU to send signals (such as buffer completion flags) to other processors and devices in the system. Each SPU has two independent 32b signal notification facilities. Reading from signal channels can be implemented to cause an SPU interrupt, or the SPU can poll (and thus block) when waiting for a signal to appear. Signal sending commands are executed like DMA commands. Signals and mailbox messages can be used to the same effect. Depending on the reason for usage, one or the other might be preferred. Table 2-2 shows the differences between signals and mailboxes.

Table 2-2: Comparative Analysis of Mailboxes and Signals

Attribute	Mailboxes	Signals
Direction:	1 Inbound 2 Outbound	2 Inbound
Interrupts:	2 Mailbox event interrupts	2 signal event interrupts
Message Accumulation	No	Yes
SPU Commands	Channel reads/writes	Sndsig, sndsigf and sndsigb
Destructive Read	Reading consumes entry	Reading channel resets bits
Channel Count	Number of available entries	Number of waiting signals

2.3 Programming Models

There are seven different programming models that can be used to design applications for the CBE: (1) Function-Offload Model, (2) Device-Extension Model, (3) Computation-Acceleration Model, (4) Streaming Model, (5) Shared-Memory Multiprocessor Model, (6) Asymmetric-Thread Runtime Model, and (7) User-Mode Thread Model [13].

GridCell uses the Computation Acceleration Model (CAM). In this model, the PPE acts as a controller and the SPEs are responsible for executing the computation-intensive sections. It does not require a significant rewrite of the application; only the individual sections that need to be accelerated on the SPU are recoded. The work can be partitioned manually by the programmer, or automatically by the compiler, and the SPUs execute the work in parallel. Memory transfers are handles through DMA commands or message passing. This is the simplest and easiest way to take advantage of the functionalities of the CBE, as well as the highly-parallel structure offered by the multiprocessor system. Table 2-3 briefly summarises the different programming models

Table 2-3: Programming Models Comparison [37]

Programming Model:	Brief Description:
Asymmetric-Thread Runtime Model	Threads can run on either the PPE or the SPEs. The PPE is multithreaded, whereas each SPE can only run one thread at a time.
Computation-Acceleration Model	The PPE acts as a controller, and the SPEs execute the most computationally-intensive functions. Each SPE runs one thread to its completion.
Function-Offload Model	The SPEs are used to run specific procedures. The bulk of the code runs on the PPE.
Device-Extension Model	Special Case of the Function-Offload model: the SPEs act as I/O devices
Shared-Memory Multiprocessor Model	The PPE and the SPEs fully interoperate; such that the CBE is a shared memory multiprocessor with 2 instruction sets.
Streaming Model	The SPEs work on data that streams though, either in serial or parallel. The PPE acts as a controller.
User-Mode Thread Model	Tasks are processed by available SPEs. Each SPE thread manages different functions. At the completion of an SPE thread, the SPE starts processing another thread.

2.4 CBE Performance Analysis

We implemented GridCell on the CBE because of its various advantages. First, the CBE is a multiprocessor system in which the processors are specifically designed to handle such tasks as the ones GridCell requires. It is a highly-parallel structure, and GridCell's algorithm is suited for such hardware since the work is very repetitive and computationally intensive. In addition, in biological systems, particles are only dependent on their immediate neighbourhood, and are independent of the movement and interactions of particles further away. Therefore, it is possible to divide the simulation volume over multiple processors, and performing the operations in parallel.

Additionally, the different functions of the SPE and the PPE allow for better power usage. The shared-memory system with three levels of memory

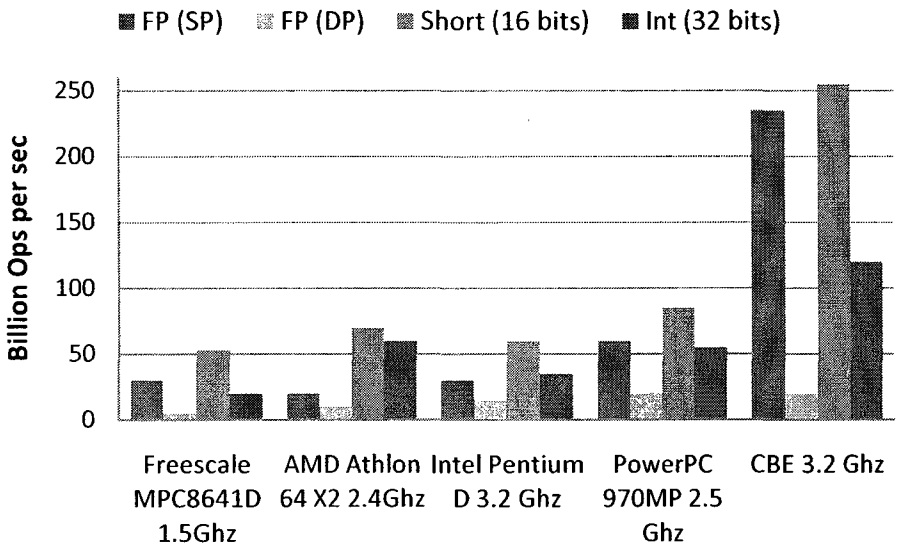


Figure 2-5: Peak performance comparison [39]

hierarchy and the asynchronous DMA functionality provide superior memory management over other multiprocessor systems. Finally, the larger register files, simpler hardware and multithreaded environment allows for increased frequency, thus breaking the frequency barrier set by other systems. These hardware capabilities of the CBE give it a competitive advantage over other processors used in high-performance computing.

Figure 2-5 outlines a comparison of peak performance of five different processors, including the CBE, in integer (16 and 32b) and floating point (SP and DP) operations [39]. One can see from the figure that SPFP and Integer operations are significantly faster on the CBE. However, no performance improvement is seen for DPFP because of their inefficiency of execution.

2.5 The Sony PlayStation 3©

GridCell was run on the Sony PlayStation 3© which has certain usage restrictions because of its use for gaming. First, only six SPEs can be used for high-performance computing (two have been turned off in order to improve yield [38]). Second, the size of main memory is limited to 250MB.

Chapter 3 The Algorithm

In this chapter, we describe the algorithms governing the original serial version, as well as the CBE implementation of GridCell, highlighting the differences between the two.

3.1 Original Implementation

The original serial GridCell algorithm, published in [25, 26], is described in the flow chart in Figure 3-1. The algorithm starts with a pre-processing stage where all the variables are set up, the 3D grid is generated, and the reactions created. Afterwards, at each time-step, GridCell loops over all the particles in the system, once for the reaction stage, and once for the movement stage. The simulation ends when the simulation time is reached.

3.1.1 Pre-processing stage

The pre-processing stage involves the initialization of the 3D grid, as well as the reactions list. When GridCell is started, a Systems Biology Mark-up Language (SBML) file must be specified, which contains the characteristics of the biological model to be simulated. Based on this information, the size of the 3D grid is obtained, as well as the initial concentration of each species involved in the simulation. Additionally, during the pre-processing stage, the list of reactions to be simulated is generated. This process is described in greater detail in Section 3.1.2.

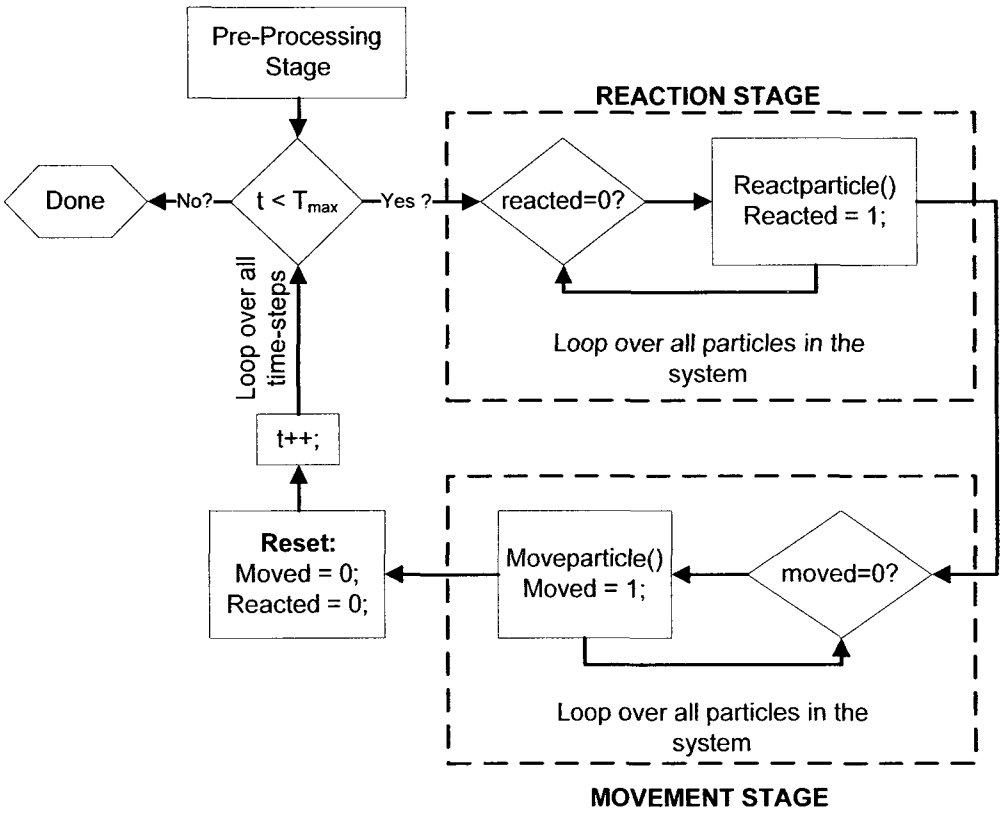


Figure 3-1: Serial algorithm flow chart

Compartments can be specified in the SBML file. However, in this version of GridCell, only one SBML compartment is supported. Additional compartments can be manually created by inserting immobile particles at specific locations in the grid, using an optional input file, referred to as the structure file. By default, the particles will be randomly placed within a cube of the volume specified in the SBML file. The structure file gives the user some flexibility in setting up the 3D structure of the grid. The characteristics of the structure file are as follows:

- The first line specifies the shape and size of the simulation space. Three positive numbers describing the relation between the three axis lengths must be chosen. The set {a b c} is equivalent to the relation $x = \frac{a}{b}y = \frac{a}{c}z$.

  These numbers must be selected based on the following rules:

  - To obtain the same volume as the one specified in the SBML file, you should choose three numbers whose product is equal to '1'.

For example, the set {1 1 1} will generate a cube, and the set {1 ½ 2} will generate a 3D rectangle with $x = 2y = \frac{1}{2}z$.

- If the product of the three values is not '1', the volume generated will be scaled by the product of the three values. For example, the set {4 2 1} will generate a rectangular volume 6 times (4*2*1) larger than the one specified in the SBML file where $x = 2y = 4z$.

- The subsequent lines are species-related, and characterize the initial location and quantity of the particles. The number of lines is not limited, and it is possible to have more than one line associated with the same species in which case particles of the same species are located in different areas of the grid. The total quantity of each species cannot exceed the value specified in the SBML file. For example, the line **3 7245 1 5 0 10 8 11 0** is interpreted as:

  - The first number (**3**) is the species. In the SBML file, species are labelled with their name. A species number is generated based on the order in which the species are specified in the SBML file. Species 'i' is the i^{th} particle type entered in the SBML file.
  - The second value (**7245**) is the number of particles that will be placed in the specified volume. This value cannot exceed the value specified in the SBML file. If '-1' is entered instead of a positive number, a homogeneous solid block of that species is created, thus forming a boundary or membrane. This allows for the manual creation of compartments.
  - The next six numbers (**1 5 0 10 8 11**) represent the coordinate ranges of the subvolume where these particles will be randomly placed (x_{min} x_{max} y_{min} y_{max} z_{min} z_{max}).
  - The last value indicates whether the particles are immobile (**0**) or mobile (**1**). Immobile particles are given a species number, they does not participate in any reactions, and have a moving ratio of 0 (they do not move).

For example, consider the following structure file:

```
1 1 4
1 25000 0 0.4 0 1 0 1 0
2 1250 2 3 4 10 11 12 0
3 1000 0 1 0 1 0 1 1
```

- The volume is scaled by 4 (1*1*4)
- 25000 mobile particles of species 1 are placed in a rectangle delimited by the two voxels of coordinates (0,0,0) and (0.4,1,1).
- 1250 mobile particles of species 2 are placed in a rectangle delimited by the two voxels of coordinates (2,3,4) and (10,11,12).
- 1000 immobile particles of species 3 are placed in the square delimited by the two voxels of coordinates (0,0,0) and (1,1,1).

In the next sections, we describe the algorithms used for the particle reactions (Section 3.1.2) and movement (Section 3.1.3). In GridCell, each particle has access to its immediate neighbours only, and is independent from particles further away. The immediate neighbours of a particle are those voxels that are within one coordinate displacement away from the current voxel; for a 3D grid, there are 26 neighbours.

3.1.2 Reactions

In the pre-processing stage, the list of reactions is generated by reading the SBML file. GridCell supports only unidirectional reactions of three possible types: (1) transformation reactions ($A \rightarrow B$), (2) split reactions ($A \rightarrow B + C$), and (3) merge reactions ($A + B \rightarrow C$). Reversible reactions are implemented as two independent unidirectional reactions where the reactants and products are reversed. More complicated reactions are reduced into multiple reactions of type 1, 2, or 3, thus creating temporary particles and intermediate reactions. However, temporary particles have a limited lifespan: if they do not react in the two time-steps following their creation, they must revert back to their original state. Additionally, the intermediate reactions created have a probability of reaction of 1

(they will always occur, as long as the necessary conditions are satisfied). Reactions are associated with the species of their first reactant.

GridCell also supports reactants or products of stoichiometry higher than one. In this case, the number of reactants and/or products increases to reflect the stoichiometry, and all the elements are treated as different. For example: $A + 2B \rightarrow 3C$ is equivalent to $A + B + B \rightarrow C + C + C$, which is interpreted in the system as having three reactants, and three products.

In the SBML file, reactions are specified in terms of reactants and products, and the reaction type is not known beforehand. Once the numbers of reactants and products have been determined, the type of reaction can be obtained:

a. **One reactant and one product**: this is a transformation reaction or "isomerisation" [40], and is the simplest type of reaction that can occur. The reaction can require the presence of a reaction modifier (or enzyme) to occur.

b. **One reactant and two or more products**: this is a split reaction or "chemical decomposition" [40]. If there are more than two products, temporary particles and reactions are created. This is best shown through an example.

   For reaction $(*)\ A \rightarrow B + C + D + E$, two temporary particles and reactions must be created: $(**)\ T_1 \rightarrow B + C$, and $(***)\ T_2 \rightarrow D + E$. This means that $(*)$ can be reduced to $(****)\ A \rightarrow T_1 + T_2$. Reaction $(****)$ is added to the list of split reactions for species A (with products T_1 and T_2), and the intermediate reactions $(**)$ and $(***)$ are added to the list of split reactions associated with T_1 and T_2 respectively.

c. **Two or more reactants and two or more products**: this is a merge reaction or "direct combination" [40]. If there are more than two reactants, and/or more than two products, temporary particles and reactions must be created. This is best shown through an example.

   For reaction $(\bullet)\ A + B + C \rightarrow D + E + F$, three temporary particles and reactions must be created: $(\bullet\bullet)\ A + B \rightarrow T_1$, $(\bullet\bullet\bullet)\ T_1 + C \rightarrow T_2$, and

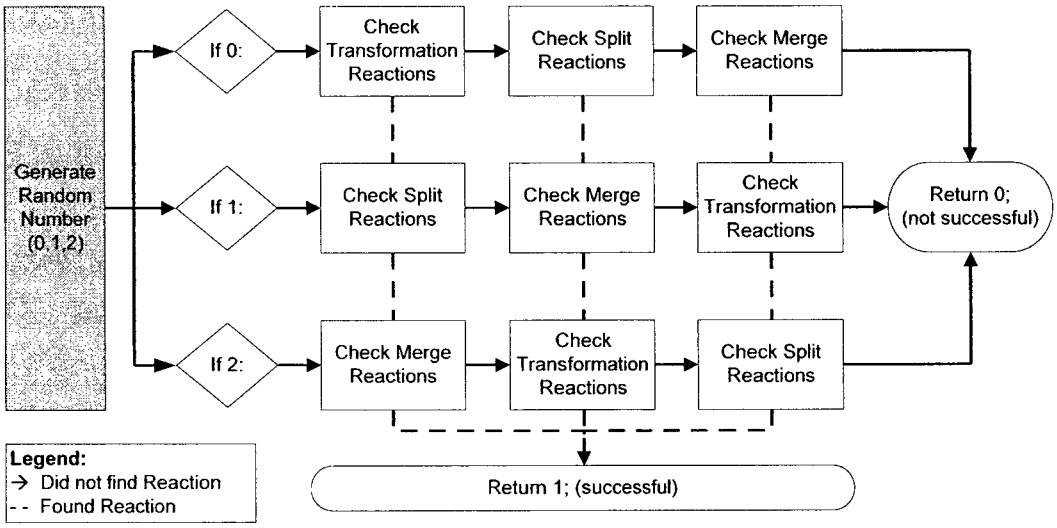


Figure 3-2: Reactions Flow Chart

(••••) $T_3 \rightarrow D + E$. This means that (•) has been reduced to (•••••) $T_2 \rightarrow T_3 + F$. Reactions (••), and (•••) are added to the list of merge reactions for A and T_1 respectively, and (••••) and (•••••) are added to the list of split reactions for T_3 and T_2 respectively.

Reversible reactions are treated as two separate and independent reactions. A reversible transformation reaction $A \leftrightarrow B$ is listed in the system as two transformation reactions: (1) a forward reaction $A \rightarrow B$ associated with species A, and (2) a backward reaction $B \rightarrow A$ associated with species B. Similarly, a reversible split reaction $A \leftrightarrow B + C$ is listed in the system as a forward split reaction ($A \rightarrow B + C$), and a backward merge reaction ($B + C \rightarrow A$). Finally, a reversible merge reaction $A + B \leftrightarrow C$ is listed in the system as a forward merge reaction ($A + B \rightarrow C$), and a backward split reaction ($C \rightarrow A + B$). This also applies to intermediate reactions created from complex reactions. Temporary particles created have an associated state 0 or 1, depending on whether the reaction they are associated with is a forward (0) or backward (1) reaction.

The reactions algorithm follows a round robin process. When a reaction is not successful, other reactions are tested until one succeeds, or all the reactions associated with the current particle's species have been tested (Figure 3-2). First, a random number between 0 and 2 is generated to indicate whether the first

reaction to test is a transformation (0), a split (1) or a merge (2). Once the reaction type has been selected, another random number is computed in order to select which reaction of that type to test first, since each species can be associated with multiple reactions of a specific type. If that reaction is not successful, the other reactions of that type are tested until one is successful, or there are no more reactions. In the latter case, another reaction type is selected based on Figure 3-2, and the process is repeated. The reaction stage stops when a reaction is successful, or when all the reactions associated with the reactant particle's species have been tested. At this point, the particle is marked as "reacted" even if no reaction was successful. The algorithm loops over all the particles in the system.

The algorithms governing the three types of reactions are described next.

3.1.2.1 Transformation Reaction

Transformation reactions are of type $A \rightarrow B$. They are the simplest reaction to execute as they do not require the creation or the consumption of neighbouring particles. However, some transformation reactions require the presence of a reaction modifier (or enzyme) in the immediate neighbourhood of the reactant particle. This information is stored within the reaction characteristics. When GridCell determines that a reaction requires an enzyme, it looks for it in the 26 neighbouring voxels. If the enzyme is not found, the reaction fails.

If the reaction does not require an enzyme, or if the required enzyme is found, a uniform random number α is generated and compared to the reaction's probability of reaction R_n. If $\alpha < R_n$, the reaction is successful, and the transformation occurs. The type of the current voxel is set to B, and the concentrations of A and B are updated in the system: that of A is decremented, while that of B is incremented. If $\alpha \geq R_n$, the reaction fails.

3.1.2.2 Split Reaction

Split reactions are of type $A \rightarrow B + C$. Because they consist of the creation of two particles, split reactions require the presence of at least one empty

voxel in the immediate neighbourhood of the reactant particle in order to house the second product. In addition, split and merge reactions could involve temporary particles, and must check the state of the first reactant particle with the direction of the selected reaction (forward or backward). The condition requires one of three situations to be true: (1) the state is -1, meaning that the particle is not a temporary one; (2) the state is 0, and the reaction is forward; or (3) the state is 1 and the reaction is backward. If this condition is not satisfied, the reaction fails.

On the other hand, if the condition is satisfied, GridCell looks for an empty location amongst the surrounding voxels. If no voxel is unoccupied, the reaction fails.

If an empty voxel is located, a uniform random number α is generated and compared to the reaction's probability of reaction R_n. If $\alpha < R_n$, the reaction is successful, and the split occurs; the type of the current voxel is set to B, and type C is set into the empty voxel. Finally, the concentrations of A, B and C are updated accordingly. If $\alpha \geq R_n$, the reaction fails

3.1.2.3 Merge Reaction

Merge reactions are of type $A + B \rightarrow C$. They are inherently similar to split reactions, with one major difference: merge reactions require the presence of the second reactant B in the immediate neighbourhood of the current particle. Similarly to the split reaction, the temporary particles condition is evaluated, and, if not satisfied, the reaction fails.

If the condition is satisfied, GridCell searches for the second reactant in the immediate surroundings of the current particle. If it is not found, the reaction fails. Otherwise, a uniform random number α is generated and compared to the reaction's probability of reaction R_n. If $\alpha < R_n$, the reaction is successful, and the merge occurs; the type of the current voxel is set to C, and the second voxel is marked as empty. Finally, the concentrations of A, B and C are updated accordingly. If $\alpha \geq R_n$, the reaction fails.

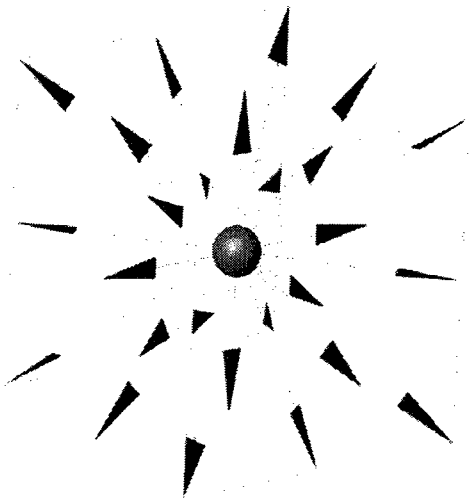


Figure 3-3: D3Q27 Grid [25]

Once all the particles in the system have reacted, the algorithm loops again over all the particles to see if they can diffuse. The movement algorithm is described in the following section.

3.1.3 Movement

In GridCell, particle movement is implemented as a Brownian random walk, where each particle can move only once per time-step. The algorithm starts by determining if the particle can move, in which case its moving ratio is higher than 0. Immobile particles can form compartment boundaries, and provide a movement constraint for the diffusing particles. If the particle cannot move, the algorithm skips to the next particle. Otherwise, a random number α is generated, and compared to the particle's moving ratio M. If $\alpha \geq M$, the movement does not occur.

If $\alpha < M$, the movement can occur and the destination voxel to which the particle will move is determined. A random number between 1 and 27 is generated that indicates one of the possible 27 locations (D3Q27 grid [25], Figure 3-3). If this destination voxel is occupied, collision occurs, and the particle cannot move. If the voxel is empty, the particle is moved to the new location, its moved flag is set to 1, and the original voxel is marked as empty. The special case where

the destination and source voxels are the same ($(X, Y, Z) = (X_{new}, Y_{new}, Z_{new})$) is treated as any other collision case.

The CBE implementation of GridCell is described in Section 3.2.

3.2 CBE Implementation

The CBE algorithm is displayed in Figure 3-4. Execution is shared between the PPU and the SPU. The PPU is responsible for the pre-processing and initialization stages in which the 3D grid is set up. Because SBML is not used for this version of GridCell, the 3D grid and reactions are set-up manually, based on the same rules and conditions described earlier.

3.2.1 Overview of the CBE implementation

The strategy to parallelize GridCell is based on the fact that biochemical systems are inherently parallel systems where local events are independent from events further away. Therefore, it is possible to divide the simulation space over the different SPUs to take advantage of the parallel structure of the CBE. Each SPU executes the simulation on a smaller portion of the volume, and then communicates the results back to the PPU. Additionally, it is possible to obtain a second level of parallelism due to the SIMD nature of the SPU, where four voxels are processed at one time.

Furthermore, because of the highly-parallel structure of the CBE, we expect increased performance of the CBE version of GridCell, compared to the serial version.

3.2.2 PPU pre-processing stage

The PPU pre-processing stage involves generating the DMA lists that will be used by the SPUs to fetch all their blocks. Given the sizes of the 3D grid, the SPU partition, and each SPU block in LS, it is possible to determine the number of blocks that will be fetched from main memory by each SPU, at each time-step.

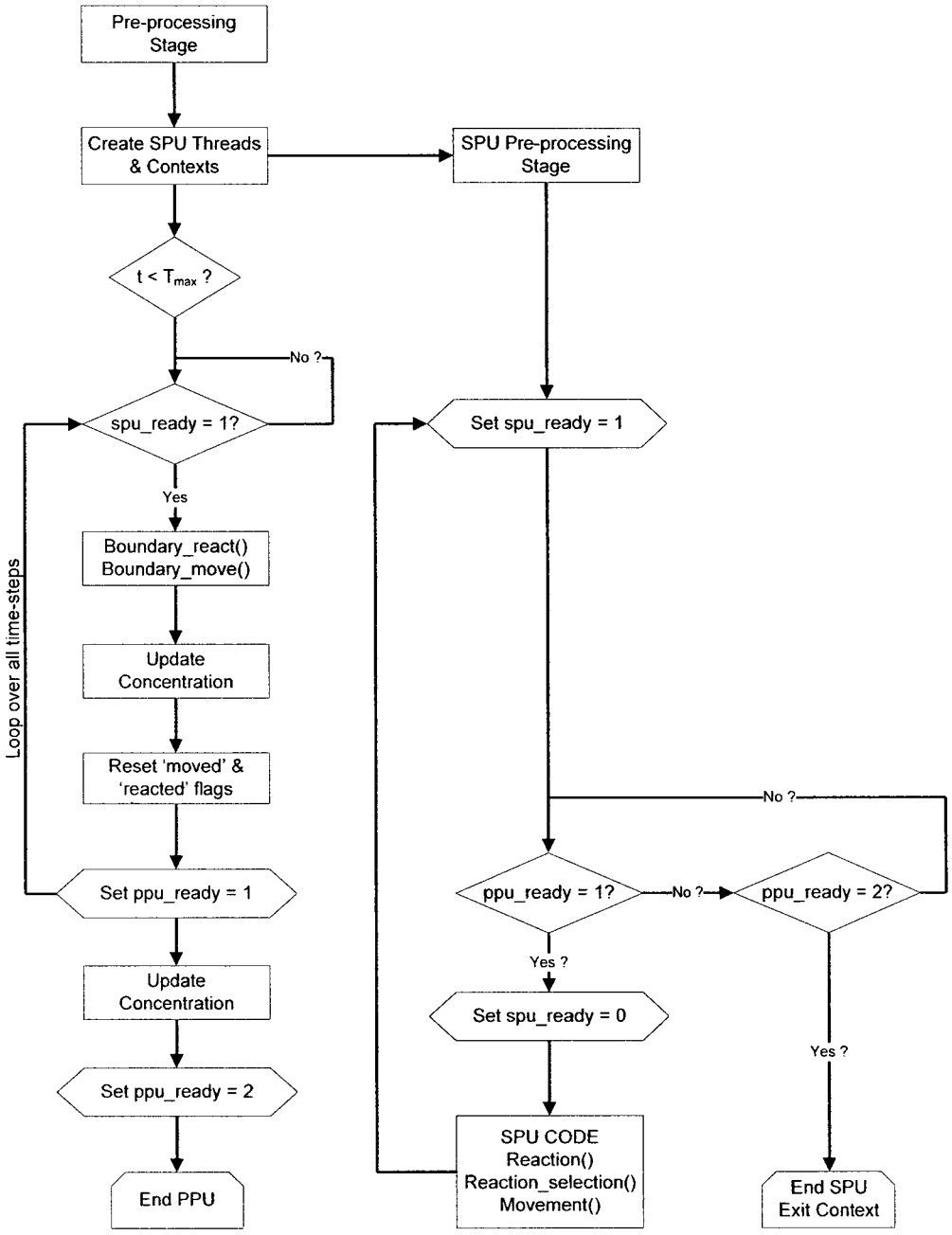


Figure 3-4: Flow chart of the CBE algorithm

Table 3-1: Number of blocks for n SPUs

3D Grid (main memory)	SPU Block (Local Store)	Number of Blocks
X	spu_c	$block_x = \frac{X-2}{n * spu_c}$
Y	spu_r	$block_y = \frac{Y-2}{spu_r}$
Z	spu_d	$block_z = \frac{Z-2}{spu_d}$
Number of blocks per SPU = $numOfDMALists = block_x * block_y * block_z$		

There is one DMA list per SPU block, so the total number of blocks per SPU is equivalent to the number of DMA lists per SPU. Additionally, the DMA lists are generated in the same order as the blocks will be read by the SPU (see Section 3.2.5). Table 3-1 summarizes how to obtain the number of blocks in each direction, for a simulation that uses 'n' SPUs.

Once the number of blocks per SPU is obtained, the DMA lists must be generated. It is necessary to use DMA lists because the 3D grid is stored in consecutive locations in main memory, but the SPUs operate on non-consecutive data since the grid is partitioned. A 3D grid is stored in main memory starting with the elements in the Z direction (for example: 000 001 002 ... 010 011 012 .. 100 111 112 ...). Each DMA command must transfer σ particles, where $\sigma = spu_d + 2$. The effective address of each transfer for block (spu_c, spu_r, spu_d) is determined by the equation: $ea_{low} = \&grid[n + i + ix[j + ij[ik]$.

Figure 3-5 shows various effective address examples. The equation parameters are:

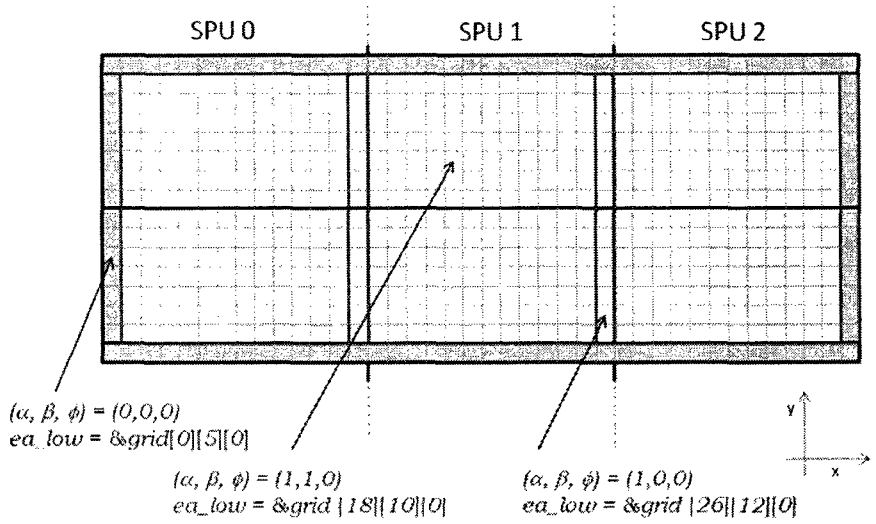
(1) n is the SPU number (starting from 0)

(2) i and j are the x and y coordinates in the SPU block respectively

(3) ix, iy and ik are given by the following equations:

   a. $ix = \propto * spu_c$; α is the number of the block in the x direction.

   b. $iy = \beta * spu_r$; β is the number of the block in the y direction.



$$ea\_low = \&grid[n+i+ix][j+ij][ik]$$

Figure 3-5: DMA List Addresses

c. $ik = \emptyset * spu_d$; ϕ is the number of the block in the z direction.

Once the DMA lists are generated, an additional vector is created which stores the addresses to each DMA list. Since the number of DMA lists is usually too large to fit in LS, the SPU fetches the DMA list corresponding to each block before operating on that block, instead of storing the inventory of DMA lists. Therefore, during the SPU pre-processing stage, the vector of DMA list addresses is fetched first. However, each DMA list address is 32b or 8B, but the transfer size must be a multiple of 16B, so there must be an even number of DMA lists. Furthermore, the transfer size cannot exceed 16KB, resulting in a maximum number of DMA lists/SPU blocks that is allowed:

$$transfer\ size = numOfDMALists * sizeof\ (long) \leq 16KB$$
$$transfer\ size = numOfDMALists * 8B \leq 16KB$$
$$numOFDMALists \leq \frac{16KB}{8B} = 2048$$

Consequently, the total number of DMA lists (and SPU blocks) must be even and less than 2048 so that the memory transfer requirements are satisfied.

3.2.3 PPU Execution

Once the DMA lists and address vector have been generated, the PPU creates the SPU contexts and the SPU threads. At this point, the SPU pre-processing stage begins (see Section 3.2.4). Once the SPU pre-processing has completed, the SPU sends a signal to the PPU's mailbox indicating that the SPU is ready to begin execution. At this point, the time-step loop begins (§), and the execution continuously switches between the PPU and the SPUs.

The PPU carries out the reaction section on the boundary particles (details are outlined in Section 3.2.3.1). It then sends a message to the SPU mailboxes to signal that data is ready and the SPUs can start processing. Once an SPU has finished processing all of the blocks assigned to it, it sends a message to the PPU to signal that it is ready for the next time-step. The PPU waits for all the SPUs to be done, executes the movement code on the boundaries, updates the concentration of all the species, resets the 'moved' and 'reacted' flags to 0 for all particles, and increments the time-step. The process restarts again from (§). When the simulation time is reached, the execution exits the time-step loop, and the PPU sends a message to the SPU mailboxes to signal the end of the simulation, at which point the SPU threads are completed.

3.2.3.1 Boundary Particles

Although the SPUs are responsible for the bulk of the data processing, the PPU does some of the work also, as shown in Figure 3-4. In a 3D cube, each particle has access to 27 voxels, including itself (D3Q27 grid). However, the particles that are on the boundary of the cube (e.g.: x = 0) only have 18 immediate neighbours, and there are six different boundaries corresponding to: (1) $x = 0$, (2) $y = 0$, (3) $z = 0$, (4) $x = x_{max} - 1$, (5) $y = y_{max} - 1$, and (6) $z = z_{max} - 1$.

To determine if a particle is on the boundary, a conditional statement must be used, which is inefficient on the SPU. In order to eliminate this conditional, boundary particles are not evaluated on the SPUs. Instead, the PPU serially performs reaction and movement operations on those particles, whereas the SPU

processes the rest of the grid. Consequently, out-of-bounds assessments are no longer needed on the SPU since no boundary particles are evaluated. For a grid of size (*X*, *Y*, *Z*), the area that is operated on by the SPUs is $(X-2)(Y-2)(Z-2)$. For larger blocks, the area of the boundaries becomes negligible compared to the area operated on by the SPUs. Hence, the effect of the PPU operation becomes negligible overall.

Without boundary particles, the area of the 3D block that is worked on by the SPUs is delimited by (x,y,z) = (1,1,1) to (x,y,z) = ($x_{max}-2$, $y_{max}-2$, $z_{max}-2$). However, if this is the area that is sent to the SPU, the particles at coordinates x = 1 for example become boundary particles since they only have access to 18 neighbours (corresponding to x = 1, and x = 2). Therefore, if the SPU will analyze a block of data of size (spu_c, spu_r, spu_d), it is required to fetch a block of data of size $(spu_c+2, spu_r+2, spu_d+2)$ in order to guarantee that each particle will have access to its 26 neighbours (the D3Q27 grid must be in LS in its entirety). The voxels on the additional six planes will not yet be processed by the SPU. They are read and written to by reaction and movement codes, but those particles do not diffuse nor react. They will be operated on by the PPU, other SPUs, or the same SPU as part of other blocks. This is best shown through an example.

Consider the case outlined in Figure 3-6 where the grid is distributed over three SPUs. For simplicity, 2D is assumed. However, the same concept applies to 3D. The grid is divided into three equally-sized partitions. Since the LS on the SPU is limited, each partition could be divided into smaller blocks that can fit in LS. In this example (Figure 3-6), each SPU will operate on four blocks serially. The particles on the boundaries in gray (arrows a_1, a_2, a_3 & a_4) are operated on by the PPU. For a block of size (spu_c, spu_r), SPU 0 must fetch a block of size (spu_c+2, spu_r+2) as shown in the figure, thus borrowing four boundaries:

(1) One x boundary from the PPU (arrow a_1).

(2) One x boundary from SPU 1 (arrow b).

(3) One y boundary from the PPU (arrow a_2).

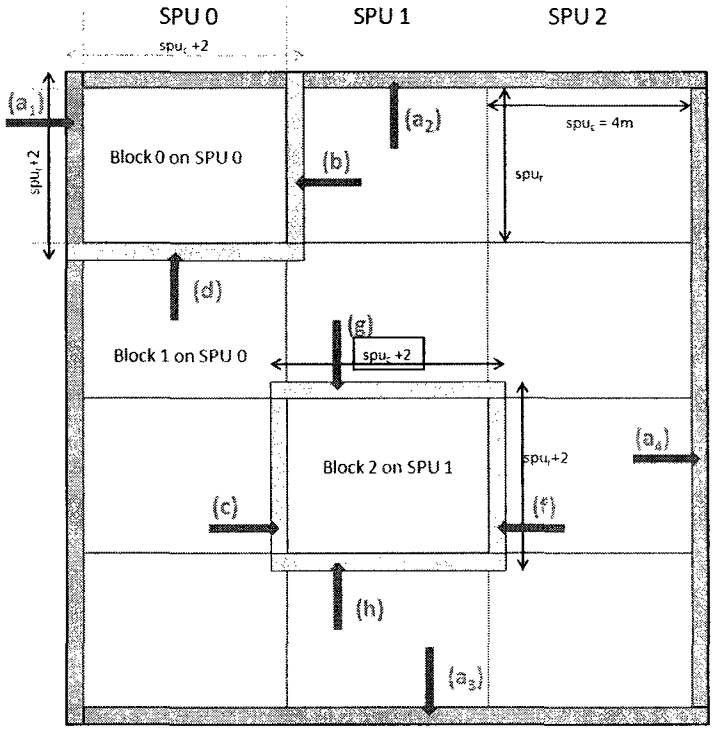


Figure 3-6: Block Distribution over 3 SPUs

(4) One y boundary from the next block on the same SPU (arrow d).

The particles pointed to by arrow 'd' are fetched twice by the same SPU, once as a boundary, and once as data to be processed. Similarly, in order to analyze block 2, SPU 1 must fetch 4 boundaries: arrows (c) and (f) from SPUs 0 and 2 respectively; and arrows (g) and (h) from blocks on SPU 1.

The SPU is a SIMD processor, and requires data to be in 128b registers. Additionally, the code is based on a series of integer or float operations (32b each). The SPU code thus executes the movement and the reaction sections on four voxels at a time. However, since each voxel needs exclusive access to 26 neighbours, it is important that those neighbours not be shared amongst the four voxels to make sure that data integrity is maintained (the same voxel cannot be written to simultaneously by two different particle operations).

Figure 3-7 shows four examples of voxel selection (2D is chosen for simplicity; the same concept applies to 3D). The figure shows that voxels need to be separated by at least two other voxels in order to be operated on

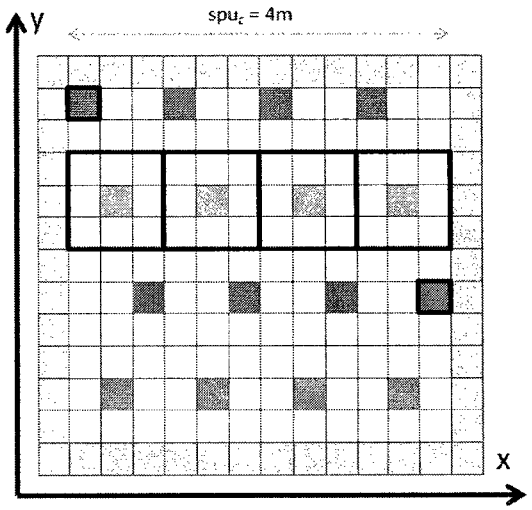


Figure 3-7: Examples of Voxel Selection

simultaneously. In GridCell, simultaneous execution of four particle operations occurs in the x direction. For this reason, for an SPU block size (spu_c, spu_r, spu_d), spu_c must be a multiple of 4, and be at least 12; otherwise, it is not always possible to find four independent particles that can be executed upon simultaneously. There are no such conditions on spu_r and spu_d.

Given the above mentioned requirement and the execution of boundaries by the PPU, there are conditions on the total size $(columns, rows, depth)$ of the 3D grid (in main memory) for accurate execution. For a simulation over n SPUs, and an SPU block size (spu_c, spu_r, spu_d) in LS, without borrowed boundaries, the conditions on the 3D grid and SPU block sizes are outlined in Figure 3-7.

3.2.4 SPU pre-processing stage

As shown in Figure 3-4, the pre-processing stage on the SPU is executed only once per simulation. It involves the memory transfers of (1) the control block, which includes all the information required for the SPU operation, (2) the reaction lists per species, and (3) the list of DMA list addresses in order to fetch the DMA lists. Once this is done, the SPU signals the PPU that it is ready to operate. The DMA transfers of the DMA list and the actual block of data occur within the processing stage. Based on the size of the 3D grid, each SPU could

Table 3-2: Conditions on the grid size and number of SPU blocks

Variable:	Condition:
Size of the 3D grid:	(1) $columns = (spu\_columns * n) * m_1 + 2$ (2) $rows = spu\_rows * m_2 + 2$ (3) $depth = spu\_depth * m_3 + 2$ *where m_1, m_2 and m_3 are non-zero positive integers*
Maximum number of blocks & DMA lists per SPU	$numOfDMALists \% 2 = 0$ $numOfDMALists \leq 2048$
Minimum value for $block_z$	$block_z \geq 6$
Value of spu_c	$spu_c = 4 * m$ $spu_c \geq 12$

process more than one data block per time-step resulting from the limited size of the LS.

The processing stage involves (1) fetching the DMA list from memory, (2) fetching the data block from memory using DMA list commands (Section 3.2.5), (3) executing the reaction and reaction-selection stages (Sections 3.2.6 & 3.2.7), (4) executing the movement stage (Section 3.2.8), and finally, (5) writing the block back to memory.

3.2.5 Multi-buffering

Figure 3-8 shows how multi-buffering is done on the SPU. Since each SPU will operate on multiple blocks, the processing stage involves moving each block from main memory into the SPU, operating on it, and then writing it back to main memory. However, memory transfers are time-consuming, and as the number of blocks and/or the transfer size increase, the amount of time wasted in memory transfers becomes significant.

Multi-buffering is a way of overlapping the execution with memory transfers in order to eliminate their effect on performance. In fact, with multi-buffering, it is possible to eliminate the effect of all but two memory transfers: (1) the initial transfer from main memory into LS, and (2) the final write to main memory.

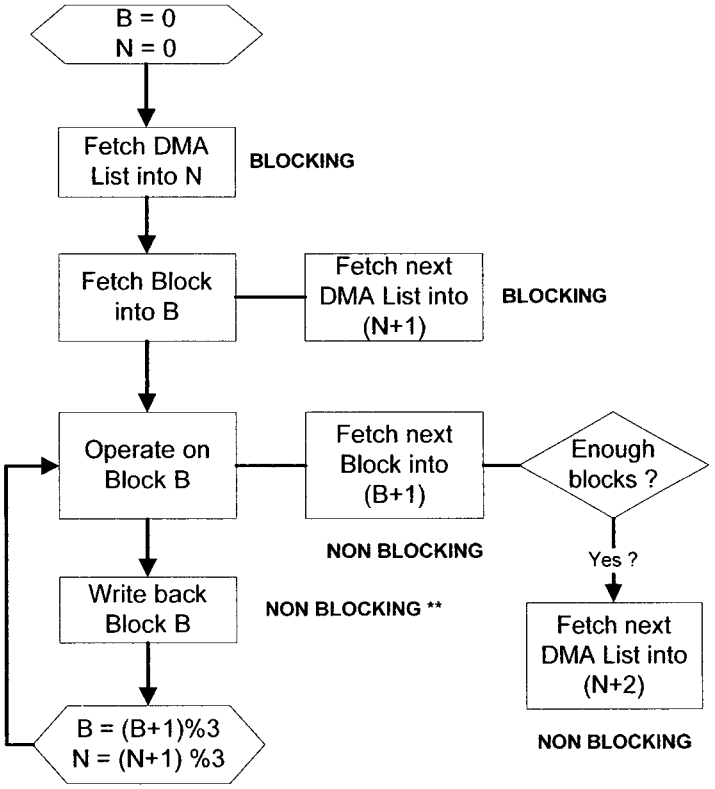


Figure 3-8: Double Buffering Memory Transfers

In GridCell, before the SPU can transfer a block into its LS, it must obtain the corresponding DMA list from memory. Therefore, each block requires two 'get' commands (to fetch data), and one 'put' command (to write data). The execution starts with three consecutive 'get' operations for (1) the first DMA list, (2) the first block, and (3) the second DMA list respectively. These commands are blocking since they are the initial transfers. Afterwards, the execution loops over all the SPU blocks in the system.

During each iteration, the SPU operates on the current block, while fetching the next block and the following DMA list. At any time, the SPU stores three DMA lists, and two SPU blocks, except for the last pair of blocks where it does not fetch a third list (since there are no more blocks to operate on). The DMA lists are therefore triple-buffered, whereas the SPU blocks are double buffered. At the end of each iteration, the SPU writes back the current block into main memory. This is a non-blocking operation which overlaps with the execution on the following block.

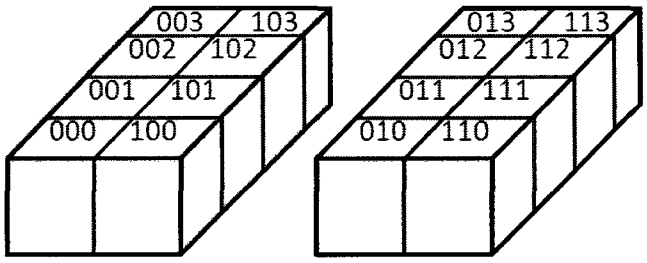


Figure 3-9: Example of Block Selection for Double-Buffering

However, each SPU block contains data shared with other blocks. Thus, adjacent blocks cannot be double-buffered. A pattern of execution must be found such that blocks can be double-buffered without jeopardizing data integrity. The algorithm loops over blocks in the Z direction, for each X and Y pair. It is possible to double-buffer even-numbered blocks, and odd-numbered blocks separately, as shows in the following example.

If there are two blocks in the X direction, two blocks in the Y direction, and four blocks in the Z direction (as outlined in Figure 3-9), the even blocks 000, 002, 010, 012, 100, 102, 110, and 112 can be double-buffered in that order, since they do not share any sides or corners. Similarly, the odd blocks 001, 003, 011, 013, 101, 103, 111, and 113 can be double-buffered in that order. The DMA list for block 001 is fetched from memory while the SPU is executing on block 111, and the block itself is fetched while writing back block 111 to memory (since both are non-blocking operations) However, blocks 111 and 001 share a side. If the execution proceeds as mentioned, block 001 will have incorrect data. This issue occurs as long as there are less than six blocks in the Z direction. As a result, for accurate execution, it is necessary to have at least six blocks in the Z direction on each SPU (Table 3-2) in order to guarantee precise multi-buffering. Since the SPU block is small due to the limited LS, this condition is not unrealistic since simulated volumes (3D grids) have a significantly large number of blocks.

In the next sections, we describe the algorithm for the reactions, reaction selection, and the movement sections. To eliminate unnecessary conditionals, empty voxels are treated like any other voxels. Prior to the beginning of the

Table 3-3: Reaction Results by type (SPU)

Reaction Type	L Vector	Results Vector
Transformation	**Lt** = $\{Lt_1, Lt_2, Lt_3, Lt_4\}$	**TransType**: $\{Pt_1, Pt_2, Pt_3, Pt_4\}$
Split	**Ls** = $\{Ls_1, Ls_2, Ls_3, Ls_4\}$	**Types1**: $\{Ps_1, Ps_2, Ps_3, Ps_4\}_1$ **Types2**: $\{Ps_1, Ps_2, Ps_3, Ps_4\}_2$ **Coordinates of 2nd result**: (Xfs, Yfs, Zfs)
Merge	**Lm** = $\{Lm_1, Lm_2, Lm_3, Lm_4\}$	**Typem1**: $\{Pm_1, Pm_2, Pm_3, Pm_4\}_1$ **Typem3**: $\{Pm_1, Pm_2, Pm_3, Pm_4\}_3$ **Coordinates of 2nd reactant**: Xfm, Yfm, Zfm

execution, if the voxel is empty (type 0), a flag is set to make sure movement and reaction are not possible.

3.2.6 Reactions

In the serial code, the reactions are executed one type at a time. However, this involves a lot of inefficient conditional statements. For this purpose, GridCell attempts to find one successful reaction per type for each particle. The results of each are saved in temporary variables (Table 3-3, Section 3.2.7).

Prior to the reaction execution, the type of the particles tested is determined. If the voxel is empty (type 0), or if there are no reactions associated with that species, a flag is set to make sure that no reaction will occur for that voxel.

The SPUs also keeps track of the concentration changes for each species. At the end of every time-step, the SPUs write the concentration changes to memory so that the PPU can update the total amounts, and then reset the concentration vector to 0 for the next time-step. The PPU can therefore track the concentration evolution at each time-step.

In the next sections, we present the algorithms for the (1) transformation, (2) split, and (3) merge reaction respectively, as well as the modifications that are necessary to be able to parallelize the code.

3.2.6.1 Transformation Reaction

Transformation reactions are the simplest reaction type. No new coordinates are required, except to search for reaction modifiers. The order of execution is very similar to that of the serial code, with the only exception being that four reactions are being executed simultaneously.

i. The type at each voxel is determined to make sure empty voxels or particles with no transformation reactions do not react.
ii. Four random numbers are generated to determine which transformation reaction to test first for each particle.
iii. For each reaction that requires a modifier (or enzyme), GridCell searches the immediate surroundings of that particle for the modifier. If no enzyme is found, the reaction fails, and the probability is set to 2 for that particle.
iv. For each reaction that does not require an enzyme, and for each reaction where an enzyme was found, a uniform random number α is generated, and compared to the probability of each reaction (the vector operation $\Gamma < R_N$).
v. If the inequality is satisfied, the flag Lt is set to indicate that a reaction is successful. The expected result of the transformation is stored in a vector called TransType.

The algorithm loops over all the transformation reactions for each of the four voxels until a reaction is successful for each particle or there are no more transformation reactions to test. At the end of the transformation section, two 128b vectors are saved (Table 3-3): (1) Lt which indicates if a reaction was successful (0xffffffff) or not (0) for each voxel, and (2) TransType which stores the transformation result of the successful reactions.

3.2.6.2 Split Reaction

Split reactions involve the creation of a new particle, and thus require the presence of an empty voxel in the immediate neighbourhood of each particle. The order of execution is very similar to the order of the original implementation.

i. The type at each voxel is determined to make sure empty voxels or particles with no split reactions do not react.
ii. Four random numbers are generated to determine which split reaction to test first for each particle.
iii. The first step is to determine the temporary particles condition (see Section 3.1.2.2). If the condition is not satisfied, the reaction fails, and the probability is set to 2 for that voxel.
iv. If the condition is satisfied, the algorithm searches neighbouring voxels for an empty location. If no location is unoccupied, the reaction fails, and the probability is set to 2 for that particle.
v. If an empty voxel is found (at coordinates Xfs, Yfs, Zfs), a uniform random number α is generated, and compared to the probability of each reaction (the vector operation $\Gamma < R_N$).
vi. If the inequality is satisfied, a flag Ls is set to indicate that a reaction is successful. The two results of the split reaction are saved in vectors typse1 and types2.

The algorithm loops over all the split reactions for each of the four voxels until a reaction is successful for each particle or there are no more split reactions to test. At the end of the split section, four 128b vectors are saved (Table 3-3): (1) Ls which indicates if a reaction was successful (0xffffffff) or not (0) for each voxel, (2) the coordinates of the empty voxels (location of the second product) Xfs, Yfs and Zfs, (3) types1 which stores the expected result in the current coordinates, and (4) types2 which stores the expected results at (Xfs, Yfs, Zfs).

3.2.6.3 Merge Reaction

Merge reactions involve the consumption of an existing particle, and thus require the presence of that particle species in the immediate neighbourhood of each particle. The order of execution is very similar to the order of the original implementation.

i. The type at each voxel is determined to make sure empty voxels or particles with no merge reactions do not react.
ii. Four random numbers are generated to determine which merge reaction to test first for each particle.
iii. The first step is to determine the temporary particles condition (see Section 3.1.2.2). If the condition is not satisfied, the reaction fails, and the probability is set to 2 for that particle.
iv. If the condition is satisfied, the algorithm searches the neighbouring voxels for the reaction's second reactant. If it is not found, the reaction fails, and the probability is set to 2 for that particle.
v. If the second reactant is located at coordinates (Xfm, Yfm, Zfm), a uniform random number α is generated, and compared to the probability of each reaction (the vector operation $\Gamma < R_N$).
vi. If the inequality is satisfied, a flag Lm is set to indicate that a reaction is successful. The result of the merge reaction is saved in vector typme1. Keeping in mind that, after the merge, an empty location replaces the second reactant, vector typem3 stores the initial types at (Xfm, Yfm, Zfm).

The algorithm loops over all the merge reactions for each of the four voxels until a reaction is successful for each particle or there are no more merge reactions to test. At the end of the merge section, four 128b vectors are saved (Table 3-3): (1) Lm which indicates if a reaction was successful (0xffffffff) or not (0) for each voxel, (2) the coordinates of the second reactant (Xfm, Yfm Zfm), (3) typem1 which stores the expected result in the current coordinates, and (4) typem3 which stores the pre-merge types at (Xfm, Yfm, Zfm).

3.2.7 Reaction selection

Once all three reactions have been executed, the next step is to determine which reaction's results will be permanently recorded. To determine which reaction's results will be saved, a vector R of four random numbers between 0 and 2 is generated, indicating which reaction will be selected first for each particle (as per Figure 3-2). Three select vectors (select_trans, select_split, and select_merge) are created when R is compared with {0,0,0,0}, {1,1,1,1} or {2,2,2,2}. The values of the select vectors depend on the value of R. When R holds a:

- 0 at a particular location, select_trans will be set to 1 at that location.
- 1 at a particular location, select_split will be set to 1 at that location.
- 2 at a particle location, select_merge will be set to 1 at that location.

For example, select_trans = {0, 0xffffffff, 0, 0} means that a transformation reaction must be selected first for the second particle. A reaction type is chosen if (1) the random number generated corresponds to the reaction type, and (2) a reaction of that type was successful (based on the flag vectors Lt, Ls and Lm). If at least one of those conditions is not satisfied, GridCell checks the following reaction type, based on the flow chart in Figure 3-2.

Based on the values of the three select vectors, and the three L vectors (Lt, Lm, and Ls) (Table 3-3), GridCell determines which reaction will occur, allocates the corresponding results to the SPU block, and updates the concentration displacements of each species involved. Table 3-4 shows an example of reaction selection.

3.2.8 Movement

As previously mentioned, each SPU operates on four particles at a time. The first step is to verify if any of the four voxels are empty or the particles stationary. For any such voxel/particle, a flag is set to make sure the movement will not occur. Secondly, a vector of four random numbers α is generated, and compared to the particle's moving ratio M. Based on the results of the comparison, the movement will either occur (if $\alpha < M$), or not ($\alpha \geq M$). Then,

Table 3-4: Reaction Selection Example

Selection Vector: R = {0,1,0,2}	**Flag vectors:** Lt = {1,0,0,1} Ls = {0,1,1,1} Lm = {1,1,0,0}
Select vectors: Select_trans = {1,0,1,0) Select_split = {0,1,0,0) Select_merge = {0,0,0,1}	Select_trans [0] = 1 → Try transformation reaction first Lt [0] = 1 → **transformation reaction is successful for particle 0**! Select_split [1] = 1 → try split reaction first Ls [1] = 1 → **split reaction is successful for particle 1**! Select_trans [2] = 1 → try transformation reaction first Lt [2] = 0 → try split reaction Ls [2] = 0 → try merge reaction Lm [2] = 0 → **there are no successful reactions for particle 2 !** Select_merge [3] = 1 → try merge reaction first Lm [3] = 0 → try transformation reaction Lt [3] = 1 → **transformation reaction is successful for particle 3 !**
Concentration Update:	The concentration is updated for each of the successful reactions

GridCell establishes the destination voxels. This is done by determining the displacement from the current voxels through the generation of random numbers between 0 and 2 (Rn_x, Rn_y, and Rn_z).

The interpretation of the random numbers is as follow:

- 0 means the new coordinate is equal to the old coordinate: $X = X_{new}$.
- 1 means the old coordinate is incremented by one: $X + 1 = X_{new}$.
- 2 means the old coordinate is decremented by one: $X - 1 = X_{new}$.

There are 27 possible combinations of those three random numbers, thus obtaining one out of 27 possible destination voxels (D3Q27 grid, Figure 3-3). A total of 12 numbers are generated: three coordinates for each of four particles.

Once the displacement is obtained, the coordinates of the destination voxels are computed by comparing the random numbers with {0,0,0,0}, {1,1,1,1} and {2,2,2,2}.

The next step is checking whether the destination voxels are unoccupied. If so, the movement occurs in the same way as in the serial code: the current voxel is marked as empty, and the new voxel is assigned the particle's characteristics (type, lifetime, state, etc). At this stage, particle concentration is not affected as particles are neither created nor consumed.

At the end of the movement code, temporary particles are updated if two or more time-steps have passes since they were created. In this case, their state is reversed so that, at the next time-step, it can revert back to its original form. A temporary particle reverts back to its original form if it cannot react to produce the final product of the original complicated reaction.

For example, the reversible reaction $B \leftrightarrow C + D + E$ is reduced to:

$B \rightarrow C + T_1$ (•)

$T_1 \rightarrow D + E$ (••)

$D + E \rightarrow T_1$ (•••)

$C + T_1 \rightarrow B$ (••••)

If the forward reaction (••) could not occur two time-steps after T_1 was created, the particle T_1's state flag is reversed to indicate a backward reaction. This means that, in the next time-step, the backward reaction (••••) will have a chance to occur such that B is obtained from the temporary particle T_1. Additionally, the forward reaction (••) will not be able to occur since the temporary particles condition will not be satisfied (reaction is forward, but state indicates backward reaction).

At the end of the movement code, the block is written back to main memory, and the execution starts again on the following block. If there are no more blocks to be processed, the final write is blocking, followed by sending the concentration changes back to the PPU. At this point, the PPU will update the concentrations, advance by one time-step, and the process starts again.

Now that the algorithm has been described in detail, the results of the CBE implementation can be validated by undertaking a comparison with the results of the serial version. Additionally, performance and timing analysis can be carried out in order to show that the CBE implementation is better performing than the serial implementation.

Chapter 4 Results and Performance Analysis

In this chapter, we first analyse the results of the CBE implementation in order to establish the accuracy of the algorithm. Since SBML is not used in the CBE implementation, we adjusted the serial code to manually generate the 3D grid and create the reactions. However, the algorithm itself was not modified from the original code. Subsequently, we undertake timing analysis with the purpose of determining if the CBE version yields any performance improvements over the serial version run on the PPU, and on an Intel processor.

4.1 Verification of results

The first step is to determine the validity of the results of the CBE implementation. In order to do so, we compare the concentration results with those of a serial version running on the CBE, utilizing the PPU exclusively (referred to as 'Serial-PPU). We simulate different systems to compare and corroborate results. The first system is a simple reversible reaction $A + B \leftrightarrow C$. The second system represents the Michaelis-Menten kinetics. For the third system, the effects of crowding are analyzed, by adding inert particles to a Michaelis-Menten system, and studying the rate of product creation.

4.1.1 Simple Reversible Reaction

We start by comparing the results of a simple reversible reaction system, between the CBE implementation run on one SPU, and the Serial-PPU implementation.

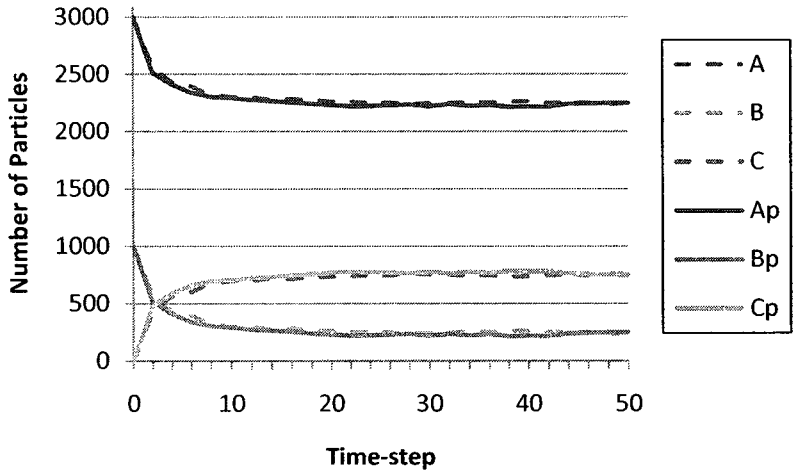


Figure 4-1: Simple reversible reaction: Concentration results of the CBE implementation using only one SPU (Ap, Bp, and Cp) compared to the results of Serial-PPU (A, B and C).

The system is given by the following equation:

$$A + B \leftrightarrow C.$$

For this simulation, the forward probability of reaction is 0.25, and the backward probability is 0.05. This unique reversible reaction is entered in the system as two independent and different reactions:

a) A forward merge reaction $A + B \rightarrow C$ with a probability of 0.25
b) A backward split reaction $C \rightarrow A + B$ with a probability of 0.05.

At the beginning of the simulation, there are 3000 particles of type A, 1000 particles of type B, and no particles of type C. Figure 4-1 shows the results of the simulation with the following parameters:

a) Grid size: 290 x 12 x 20
b) SPU block size: 24 x 2 x 2
c) Number of iterations: 50
d) Number of SPUs: 1
e) Sampling rate: 2 iterations.

As can be seen from Figure 4-1, the results of both implementations are comparable, with the exception of some expected stochastic noise. Additionally,

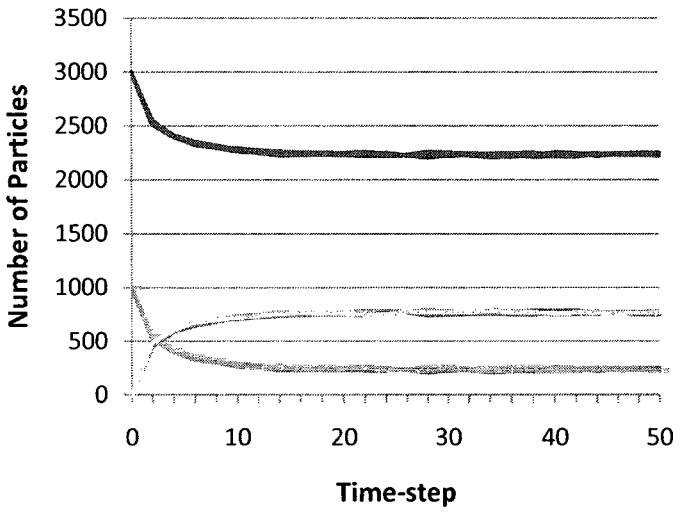


Figure 4-2: Concentration results of the simple reversible reaction on the CBE, over 1, 2, 3 & 6 SPUs

Figure 4-2 shows the results of the simple reversible reaction on the CBE, over one, two, three and six SPUs. As can be seen from the figure, the results are accurate, independently of the number of SPUs used.

4.1.2 Michaelis-Menten System

The Michaelis-Menten system describes the kinetics of many enzymes, and is given by the equation:

$$E + S \leftrightarrow ES \rightarrow E + P.$$

In this equation, S is a substrate that binds to enzyme E to yield ES, which can decompose into the product P and the enzyme E, or into its original form (E+S). Usually, the enzyme is the limiting factor, since its concentration is much lower than that of S. The simulation parameters are:

a) Grid size: 290 x 12 x 20
b) SPU block size: 24 x 2 x 2
c) Number of iterations: 500
d) Number of SPUs: 1, 2, 3 and 6
e) Sampling rate: 20 iterations
f) Initial concentrations: 1000 particles of E and 3000 particles of S.

The above 3D grid size satisfies the conditions set on the size of the grid, with respect to the number of SPUs, and the SPU block size. Additionally, for each of the simulations, the total number of blocks per SPU is even as required (Table 3-2):

For one SPU:

$290 = 2 + 1 * (24 * 12)$,

$12 = 2 + (5 * 2)$ and

$20 = 2 + (9 * 2)$.

➔ 540 blocks per SPU

For two SPUs:

$290 = 2 + 2 * (24 * 6)$,

$12 = 2 + (5 * 2)$ and

$20 = 2 + (9 * 2)$.

➔ 270 blocks per SPU

For three SPUs:

$290 = 2 + 3 * (24 * 4)$,

$12 = 2 + (5 * 2)$ and

$20 = 2 + (9 * 2)$.

➔ 180 blocks per SPU

For six SPUs:

$290 = 2 + 6 * (24 * 2)$,

$12 = 2 + (5 * 2)$ and

$20 = 2 + (9 * 2)$.

➔ 90 blocks per SPU

The system is represented by three different and independent reactions:

a) A forward merge reaction $E + S \rightarrow ES$ probability of reaction 0.25
b) A backward split reaction $ES \rightarrow E + S$ with probability of reaction 0.05
c) A forward split reaction $ES \rightarrow E + P$ with probability of reaction 0.05

Figure 4-3 compares the results of the serial version with those of the parallel version utilizing only one SPU. As expected, the time evolution of the concentration for both implementations is similar, with the exception of some expected stochastic noise. The total number of particles in the system is also similar, with a maximum error margin of 2.43%.

Additionally, we compare the results of the CBE implementation using one, two, three or six SPUs. Figure 4-4 shows the results of the simulation. As can be seen from the figure, the results overlap, and cannot be distinguished. The results are therefore accurate regardless of the number of SPUs used in the simulation.

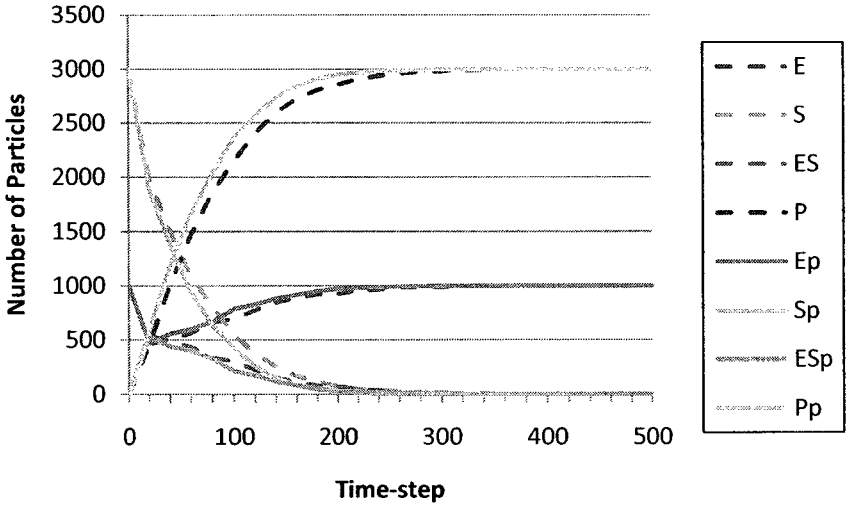


Figure 4-3: Michaelis-Menten System: Concentration results of the CBE implementation using only one SPU (Ep, Sp, ESp and Pp) compared to the results of Serial-PPU (E, S, ES, and P).

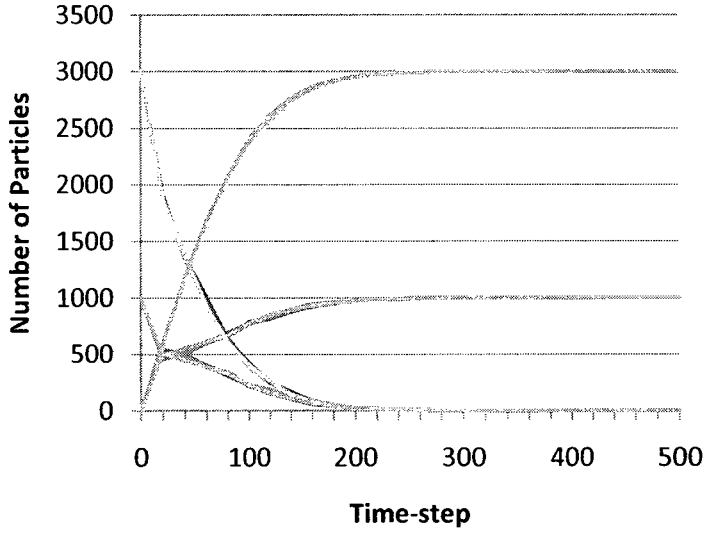


Figure 4-4: Concentration results of the Michaelis-Menten System on the CBE, over 1, 2, 3 & 6 SPUs

4.1.3 Analysis

In Sections 4.1.1 and 4.1.2, we showed that the results of the parallel implementation are comparable to the results of the original serial algorithm. We can conclude that, firstly, the modifications to the algorithm, described in Section

3.2, did not have an impact on the accuracy of the implementation. Secondly, the number of SPUs employed in the implementation does not have any consequences on the algorithm precision.

This result is indicative of the fact that the parallel version of GridCell is algorithmically correct. We can now proceed to analyze the performance of the parallel simulator by measuring and comparing execution times.

4.2 Performance Analysis

In order to show performance improvements over the serial version of GridCell, we compare the execution time of the serial version running on the CBE, utilizing only the PPU (referred to as 'Serial-PPU) with that of the CBE version employing a different number of SPUs, for a variety of test cases. There are four different parameters that can affect the execution time, and consequently the performance of the CBE implementation: (1) the number of SPUs used, (2) the size of each SPU block, (3) the size of the 3D grid that represents the volume simulated, and (4) the portion of the total volume that is initially occupied (the initial particle density, expressed as a percentage of the total volume). We discuss each of these parameters in the following subsections, and analyse their effect on performance.

Moreover, we evaluate the performance of the CBE implementation, over the serial version running on an Intel Pentium IV 3.2 GHz processor (referred to as 'Serial-Intel').

4.2.1 Serial-PPU and CBE implementations

In this section, we compare the performance of the CBE version, with the serial version run on the CBE employing the PPU exclusively (Serial-PPU). We start by determining the effect on performance of the number of SPUs used in the simulation. Afterwards, we establish the effect of different SPU block sizes, followed by that of different 3D grid sizes. Finally, we study the effect of different particle densities.

Table 4-1: Speed-up of the CBE implementation utilizing a different number of SPUs, over the Serial-PPU version.

	Serial-PPU	2 SPUs	3 SPUs	4 SPUs	6 SPUs
Time (s):	147.84	151.21	117.58	100.33	83.50
Speed-up:	1.00	0.98	1.25	1.48	1.76

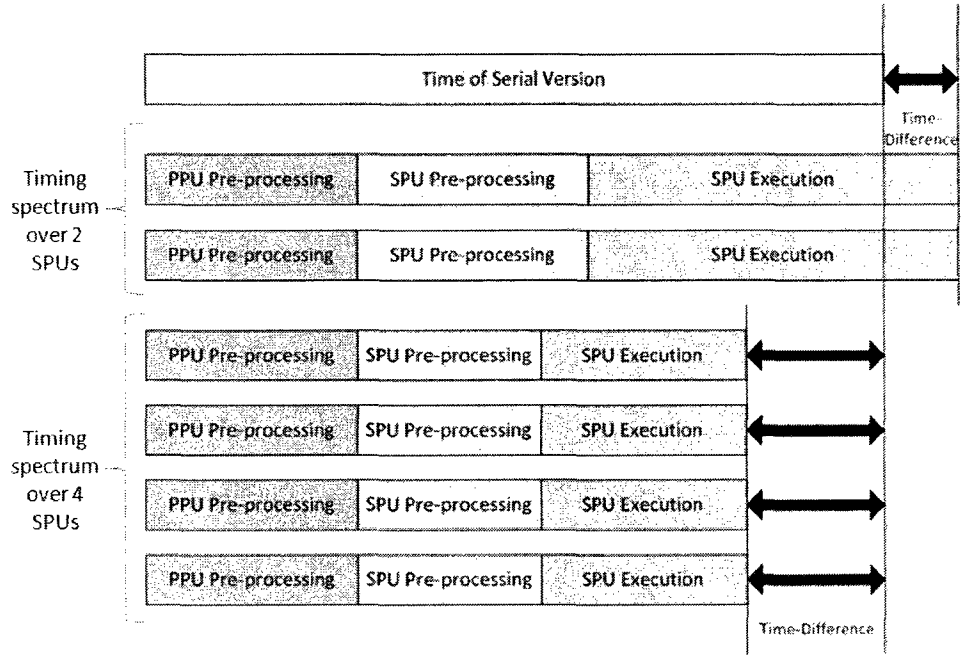


Figure 4-5: Example of timing spectrums

4.2.1.1 Speed-up over the number of SPUs

The number of SPUs used affects the execution time in different ways. For the same volume, particle density, and SPU block size, the number of used SPUs modifies the number of blocks evaluated on each SPU. Additionally, the time spent by the PPU creating the SPU threads and the contexts increases with the number of SPUs. Table 4-1 shows the execution times and the speed-ups obtained when simulating the Michaelis-Menten system with the following parameters:

a) Grid size: 866 X 22 X 14
b) SPU block size: 24 x 2 x 2
c) Number of iterations: 500
d) Number of SPUs: 2, 3, 4 and 6
e) Number of blocks per SPU: 1080, 720, 540 and 360 respectively.

As expected, using more SPUs will decrease the execution time because each SPU operates on a lesser number of blocks (Table 3-1 shows how to obtain the number of blocks per SPU). In this example, using only two SPUs yields a higher execution time because the advantage of dividing the simulation volume over two SPUs is countered by the time wasted setting up all the structures required as well as the SPUs. In order to obtain performance improvements, the time that each SPU spends operating on its blocks must be small enough such that the time spent in the pre-processing stages does not hinder performance. Figure 4-5 shows an example of timing spectrums, for two and four SPUs, compared to the execution time of Serial-PPU, where using two SPUs causes decreased performance. For the same grid size, particle density, and SPU block size, the PPU pre-processing stage takes approximately the same time, but the number of SPUs affects the time spent on the SPU pre-processing, as well as the SPU execution. Therefore, using more SPUs yields better timing results, thus increasing the speed-up over the Serial-PPU version.

4.2.1.2 Timing over SPU block size

The second parameter that may affect the execution time and the speed-ups is the size of the SPU block in LS. This value affects the transfer of each block from main memory into LS since the transfer amount is dependent on the size of the SPU block in the Z direction. Figure 4-6 shows the speed-ups obtained when simulating the Michaelis-Menten system with the following parameters:

a) Grid size: 866 X 22 X 30
b) SPU block size: 22 x 2 x 4 and 12 x 5 x 4
c) Number of iterations: 500
d) Number of SPUs: 2, 3, 4 and 6
e) Particle density: 14%

The figure shows that similar speed-ups are obtained for any SPU block size, as long as all the other parameters are maintained. A larger SPU block size reduces the number of blocks that the SPU will execute on, but it will increase the

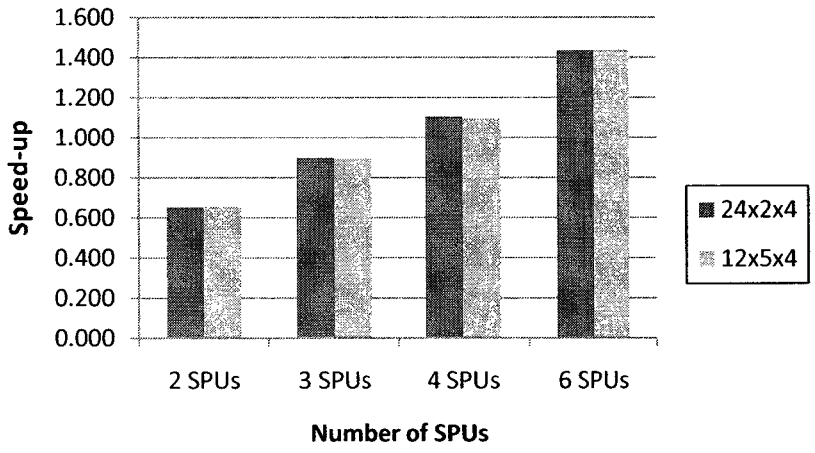


Figure 4-6: Speed-up of the CBE implementation utilizing different SPU block sizes, over the Serial-PPU version, for a different number of SPUs.

Table 4-2: Execution time of the CBE implementation over 3D grid size in seconds, using three and six SPUs

Grid Size	Serial-PPU	Three SPUs	Six SPUs
866 x 22 x 30	227.19	211.81	135.47
578 x 22 x 30	152.03	142.46	91.77
434 x 22 x 30	111.08	106.09	67.84
290 X 22 X 30	71.16	71.51	40.04

amount of data to be transferred when fetching the block into LS from main memory. Additionally, the time the SPU spends operating on the larger block will increase. The combination of all these effects is such that the SPU block size does not influence the execution time of the CBE version, and thus the performance of GridCell on the CBE is not dependent on the SPU block size.

4.2.1.3 Timing over 3D grid size

The third parameter that may influence the execution time is the size of the volume to be simulated (or the 3D grid size). Table 4-2 shows the execution time of the serial version, compared to the CBE implementation using three and six SPUs, obtained when simulating the Michaelis-Menten system for different grid sizes, with the following parameters:

a) 3D grid size: variable.

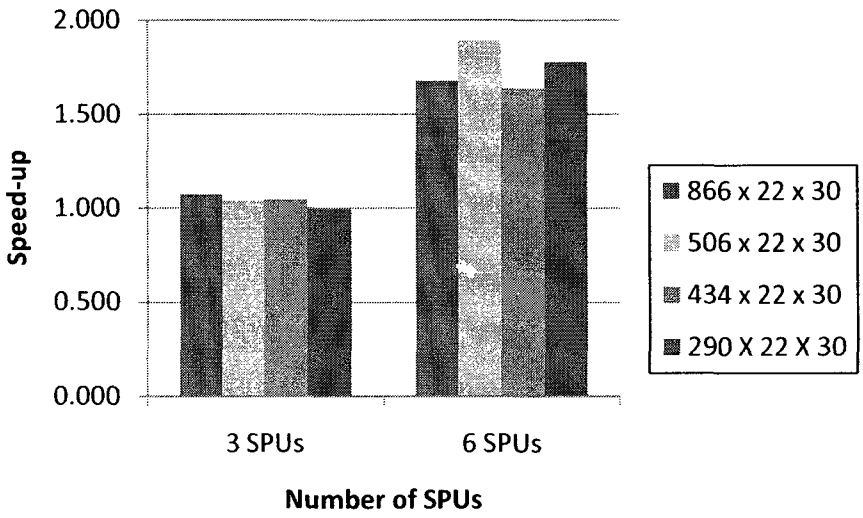


Figure 4-7: Speed-up of the CBE implementation for different 3D grid sizes, over the Serial-PPU version, for 3 and 6 SPUs.

b) SPU block size: 24 x 2 x 2
c) Number of iterations: 500
d) Number of SPUs: 3 and 6
e) Particle density: 20%

As expected, the time increases with the size of the 3D grid representing the volume to be simulated. For larger grids, the number of voxels increases, and consequently, the time to process all those voxels rises. Figure 4-7 shows the speed-ups obtained from this example. As a result of the increase in grid size, the execution time rises comparatively for both the Serial-PPU and the CBE version. Consequently, the acquired speed-ups for the different volumes are relatively close to each other, for both three and six SPUs. We can therefore conclude that, if all the other parameters are maintained, we can obtain analogous performance improvements for variable 3D grid sizes.

4.2.1.4 Timing over initial particle density

The last parameter that may affect execution time and speed-ups is the initial number of particles in the system, or the initial particle density which is expressed as a percentage of the total volume. In order to study the effect of

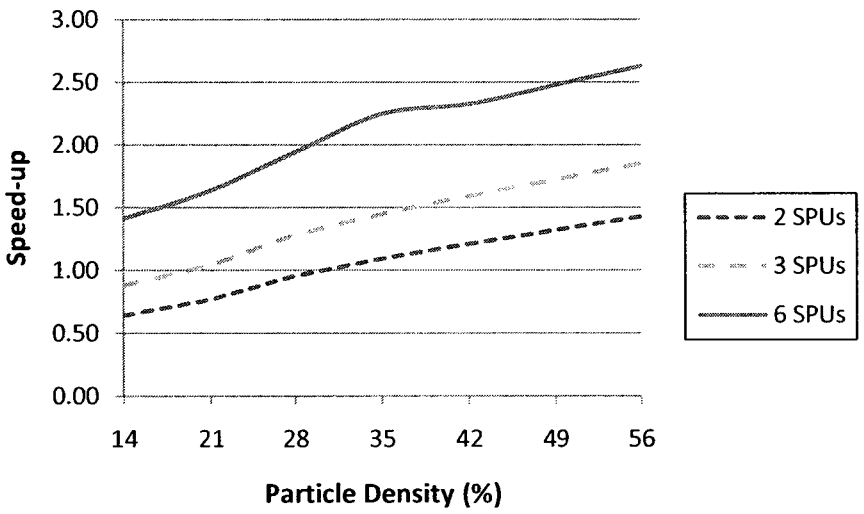


Figure 4-8: Speed-up of the CBE implementation with varying particle density, over the Serial-PPU version, for a different number of SPUs.

particle density on performance, we simulate the Michaelis-Menten system with the following simulation parameters:

a) Grid size: 434 x 22 x 30
b) SPU block size: 22 x 2 x 2
c) Number of iterations: 500
d) Number of SPUs: 2, 3 and 6
e) Particle density: 7% to 56% (7% increments)

The results of the simulation, shown in Figure 4-8, indicate that the speed-ups obtained increase with the initial particle density. In the serial algorithm, the execution over a voxel stops if any of the following occurs: (1) the voxel is empty, (2) the particle is immobile or inert, (3) a successful reaction is found, or (4) no reactions are associated with that particle type. Therefore, for a low particle density, the algorithm spends less time executing since more voxels are empty.

On the other hand, the CBE algorithm operates over four voxels at a time, including empty voxels (a flag is set in this case to prevent reactions or diffusion from occurring). Additionally, the algorithm attempts to find one successful reaction of each type, and then selects one reaction to perform. The CBE version does more work on each voxel, compared to the serial implementation. Even

Table 4-3: Timing analysis, and speed-up of the Serial-Intel version, over the Serial-PPU version, for different 3D grid sizes and particle densities.

Volume & Particle Density	Serial-PPU	Serial-Intel	Speed-up: Serial-Intel over Serial-PPU
866 X 22 X 30 – 20%	84.04	22.43	3.75
866 X 22 X 30 – 35%	132.16	40.21	3.29
866 X 22 X 30 – 50%	178.77	48.36	3.70
866 X 22 X 30 – 75%	254.59	66.83	3.81
578 x 22 x 30 – 20%	57.86	15.53	3.73
578 x 22 x 30 – 35%	89.08	22.78	3.91
578 x 22 x 30 – 50%	118.56	34.87	3.40
578 x 22 x 30 – 75%	165.74	52.94	3.13
434 x 22 x 30 –20%	42.16	12.01	3.51
434 x 22 x 30 – 35%	66.51	19.29	3.45
434 x 22 x 30 – 50%	89.21	25.41	3.51
434 x 22 x 30 – 75%	145.46	43.15	3.37
Average Speed-up			3.55

when the particle density is low, the SPU spends a significant amount of time processing empty voxels.

On the other hand, when the particle density is high, the serial algorithm must go through a higher number of non-empty voxels, which increases its execution time. However, the SPU processes these voxels anyway. Hence, when compared to the execution time of Serial-PPU, the CBE implementation's time is lower for high density. Figure 4-8 shows that speed-ups can be obtained for more than 20% initial density for three of more SPUs, and over 35% density for two SPUs. Using six SPUs always yields good speed-ups as can be seen from the multiple examples that we have examined.

4.2.2 Serial-Intel and CBE implementations

In this section, we compare the performance of the CBE version with the serial version run on an Intel Pentium IV 3.2 GHz processor (Serial-Intel, Section 4.2.2.2). To better understand the results of the comparison, we start by evaluating the performance of Serial-PPU with respect to Serial-Intel (Section 4.2.2.1).

4.2.2.1 Serial-Intel compared to Serial-PPU

In order to understand the performance of the CBE implementation when compared to the Serial-Intel version, we start by comparing the Serial-Intel and Serial-PPU versions. The Serial-PPU source code runs on the PPU, which is a PowerPC processor. Table 4-3 shows timing analysis between the Serial-PPU and the Serial-Intel versions of GridCell, for three different volumes, over four different particle concentrations.

Using the tabulated results, we conclude that the Intel version is on average 3.55 times faster than the Serial-PPU version. This is expected since commercial Intel processors are better performing than the PPU. Using this information, we can further study the timing analysis between the CBE implementation, and the Serial-Intel version.

4.2.2.2 Serial-Intel compared to Parallel version

As mentioned in Section 3.2 in the description of the CBE implementation of GridCell, the bulk of the work is processed by the SPUs. However, boundary voxels are traversed by the PPU at each time-step. Additionally, the PPU loops over the 3D grid at each iteration to reset the flags "moved" and "reacted". Consequently, the performance of the CBE implementation is bottlenecked by the PPU execution, given that the Intel implementation is significantly faster than the Serial-PPU version. In order to determine the effect of the PPU execution on overall performance, we simulate the Michaelis-Menten system with the following parameters, twice: (1) to obtain the total execution time, and (2) to gather the execution time of the SPU code only (boundary particles were not evaluated).

a) Grid size: variable
b) SPU block size: 22 x 2 x 2
c) Number of iterations: 200
d) Number of SPUs: 3 and 6
e) Particle density: 20%, 35%, 50% and 75%.

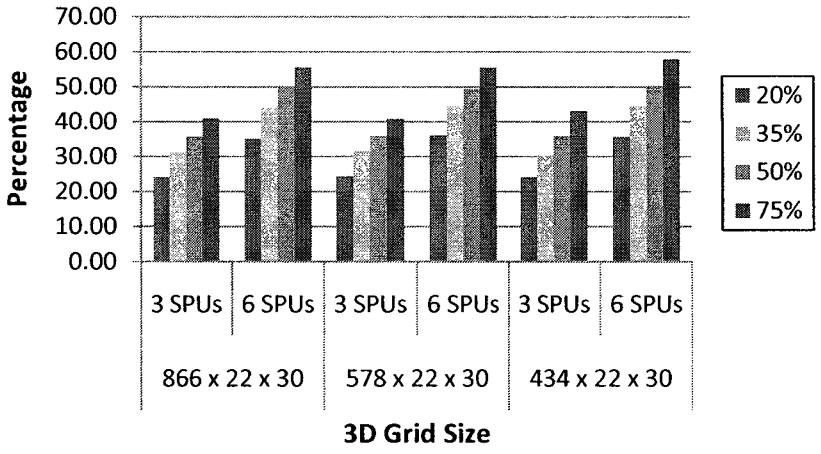


Figure 4-9: PPU Time as a percentage of total time of the CBE implementation, over varying grid sizes, number of SPUs, and particle densities.

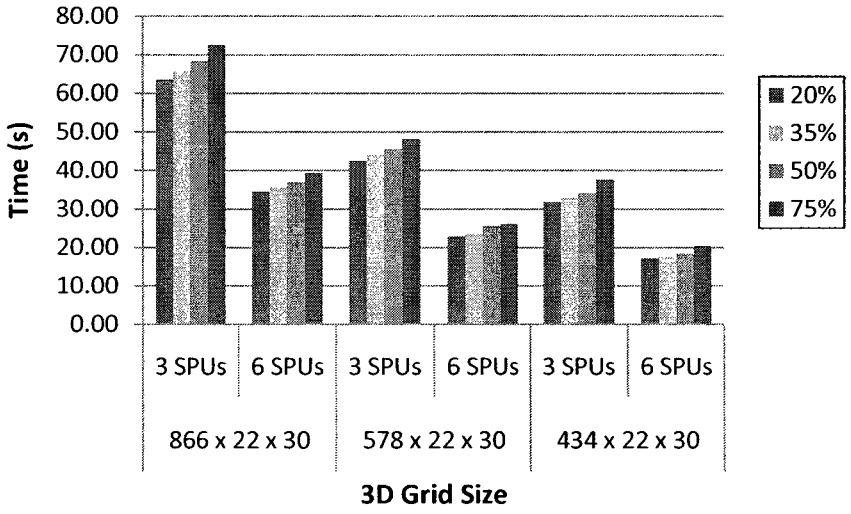


Figure 4-10: SPU Time of the CBE implementation, over varying grid sizes, number of SPUs, and particle densities.

Given the two time values collected, it is possible to estimate the execution time over the boundary particles (PPU Time). The results of the PPU and SPU times are shown in Figure 4-9 and Figure 4-10 respectively. Figure 4-9 shows the PPU time as a percentage of the total execution time of the CBE implementation. We notice that the PPU time's portion of the total time increases only with the particle density. However, as can be seen in Figure 4-10, the SPU

Table 4-4: Speed-up of the CBE version, utilizing a different number of SPUs, over Serial-Intel

Volume & Density	T_3	T_6	T_{in}	T_6/T_3	T_{12}	T_{24}	T_{in}/T_{12}	T_{in}/T_{24}
866 X 22 X 30 – 20%	83.98	53.22	22.43	0.63	33.73	21.37	0.66	**1.05**
866 X 22 X 30 – 35%	95.84	64.08	40.21	0.67	42.84	28.65	0.94	**1.40**
866 X 22 X 30 – 50%	106.54	73.8	48.36	0.69	51.12	35.41	0.95	**1.37**
866 X 22 X 30 – 75%	123.22	88.76	66.83	0.72	63.94	46.06	**1.05**	**1.45**
578 x 22 x 30 – 20%	56.26	35.96	15.53	0.64	22.98	14.69	0.68	**1.06**
578 x 22 x 30 – 35%	64.14	42.99	22.78	0.67	28.81	19.31	0.79	**1.18**
578 x 22 x 30 – 50%	71.07	55.64	34.87	0.78	43.56	34.10	0.80	**1.02**
578 x 22 x 30 – 75%	81.46	58.76	52.94	0.72	42.39	30.57	**1.25**	**1.73**
434 x 22 x 30 –20%	41.95	26.76	12.01	0.64	17.07	10.89	0.70	**1.10**
434 x 22 x 30 – 35%	40.05	32.11	19.29	0.80	25.74	20.64	0.75	0.93
434 x 22 x 30 – 50%	53.46	37.24	25.41	0.70	25.94	18.07	0.98	**1.41**
434 x 22 x 30 – 75%	66.3	48.64	43.15	0.73	35.68	26.18	**1.21**	**1.65**

time increases only slightly over rising particle density, because the SPU operates over all voxels, even if they are empty, unlike the PPU which skips empty voxels.

In order to reduce the effect of the PPU execution on the overall execution time, and thus obtain positive speed-ups, we can utilize a higher number of SPUs. Given that the PS3© only has six active SPUs, we can speculate as to the minimum number of SPUs required to obtain any speed-up. The speed-up is given by the following equation: $Speedup = \frac{T_n}{T_i}$, where T_i is the time of Serial-Intel, and T_n is the CBE version's time over 'n' SPUs. In this case, we have access to three and six SPUs, so we can determine the relationship between the execution time over six (T_6) and three (T_3) SPUs. In this case, $T_6 \approx 0.7T_3$ which we can generalize to $T_{2n} \approx 0.7T_n$. We can therefore estimate the execution time over 12 and 24 SPUs. The results are tabulated in Table 4-4. A ratio $\frac{T_i}{T_n}$ higher than one indicates that a speed-up can be obtained. In this case, 12 SPUs will produce a speed-up in the case of 75% particle density, whereas 24 SPUs will generate better performance over Serial-Intel for all but one case.

Furthermore, given that the PPU execution represents a bottleneck to the overall performance over rising particle density, it is possible to reduce the effect of the PPU execution on the total CBE time if the specific section of the PPU

source code can be parallelized. Reducing the PPU time's sensitivity to particle density by equally processing all voxels, similarly to the SPU execution, will improve the overall performance for dense systems.

Chapter 5 Summary and Conclusion

5.1 Conclusion

Because of its highly-parallel structure, the CBE is an attractive platform for biochemical simulations, as biological events happen locally, and are independent from particles further away. We parallelized GridCell based on the fact that it is possible to divide the simulation space over multiple processors that operate simultaneously. The SIMD structure of the SPUs provides an additional level of parallelism as voxels are processed four at a time, compared to only one at a time for the serial versions.

By studying the performance of the CBE implementation, we conclude that the CBE version of GridCell yields better performance for large dense systems over the Serial-PPU version. Additionally, the size of each SPU block does not affect the execution time of the simulation, since the total number of voxels to be processed overall is the same. The parameter that affects the overall speed-up is the particle density since the serial version processes only non-empty voxels, and for increasing density, the number of non-empty voxels rises, therefore raising the execution time. On the other hand, in the CBE version, empty voxels are processed in the same way as particles, thus density has a limited affect on performance.

Finally, the CBE adaptation of GridCell shows performance improvements over the serial version implemented on an Intel processor when a higher number of SPUs can be used. In fact, it is possible to obtain platforms equipped with more than eight SPUs, such as Cell Blades.

We conclude this thesis with a brief discussion of potential future work that can help enhance the performance and usability of GridCell.

5.2 Future Work

Potential future work that can be done to enhance GridCell's usability can be divided into two categories: (1) those modifications that increase performance, and (2) those that render it more accessible to users unfamiliar with programming techniques. We end this thesis with an outline of prospective work under these two categories.

5.2.1 Increased Performance

Although we are able to achieve improved performance by porting GridCell onto the CBE, superior speed-ups can be achieved by additionally fine-tuning the code. First, it is important that the PPE execution of the boundary particles not hinder performance on the CBE. Since it is possible to write SIMD code for the PPE, rewriting the section of the algorithm that executes on the PPE in SIMD will help achieve increased performance because the execution will be partially parallel. This will also allow for better results compared to the Serial-Intel adaptation of GridCell.

Additionally, the random number generator libraries used produce random numbers serially. Hence, in order to obtain a vector of random numbers on the SPE, the function must be called multiple times. Unfortunately, function calls instigate cycle-delays on the SPE as a consequence of the branch request. In order to minimize this effect, we can rewrite the libraries in SIMD, thus reducing the number of function calls. The libraries can be written in SIMD on the PPE as well since they are called by the portion of the algorithm that operates on the PPE.

With these modifications to the code, we can expect to see increased performance and higher speed-ups without modifying the algorithm itself.

5.2.2 Increased Accessibility

Currently, it is necessary to recompile the code with different parameters in order to simulate various systems. In an effort to make the algorithm more user-friendly, a graphical user interface (GUI) can be designed such that the user can enter the reaction information as well as the simulation and system parameters directly, without the need to modify and recompile the source code. Consequently, users unfamiliar with programming tools will be able to use GridCell.

A GUI will also assist the user in obtaining visual results directly, without having to use other tools, such as MATLAB, to plot the program's output, thus facilitating the application's usage.

Furthermore, adding SBML support to the CBE version of GridCell will allow existing systems to be simulated easily, without the need to manually recreate them. Using SBML files as input will help simulate more complicated systems straightforwardly and at a higher level, since there will be no need to translate the information in the SBML file into source code that GridCell understands.

These modifications will make GridCell a more user-friendly application which, together with the performance improvements acquired, will render the simulator a great instrument to use in the computational biology field.

Bibiography

1. Drummond, A., *Computational Biology and Evolution.* http://bioinf.cs.auckland.ac.nz/.
2. Michor, F., *Mathematical Models of Cancer Stem Cells.* J. Clin. Oncol, 2008. **26**(17): p. 2854-61.
3. Michaelis, L. and M. Menten, *Die Kinetik der Invertinwirkung.* Biochem., 1913. **49**: p. 333-369.
4. Ander, M., et al., *SmartCell, a framework to simulate cellular processes that combines stochastic approximation with diffusion and localisation: analysis of simple networks.* Syst. Biology, 2004. **1**(1): p. 10.
5. Andrews, S., *Smoldyn.* www.smoldyn.org.
6. BPS, *Biological Pathways Simulator.* http://www.brc.dcs.gla.ac.uk/projects/bps/.
7. CI, *Cell Illustrator.* http://www.genomicobject.net/member3/index.html.
8. COPASI, *Complex Pathway Simulator.* http://www.copasi.org/tiki-index.php.
9. E-Cell, *E-Cell.* http://www.e-cell.org/ecell/.
10. Gepasi, *Gepasi.* www.gepasi.org.
11. Gillspie, D.T., *A General method for numerically simulating the stochastic time evolution of coupled chemical reaction.* Journal of Computational Physics 1976. **22**(4): p. 403-434.
12. Hattne, J., D. Fange, and J. Elf, *Stochastic reaction-diffusion simulation with MesoRD.* Bioinformatics, 2005. **21**(12): p. 2923-2924.
13. JigCell, *JigCell.* http://jigcell.biol.vt.edu.
14. Kyoda, K.M., et al., *BioDrive: Simulator for Biochemical and Genetic Networks.*

15. LeNovère, N. and T.S. Shimizu, *StochSim: modelling of stochastic biomolecular processes.* Bioinformatics 2001. **17**(6): p. 575-576.
16. LeNovère, N. and T.S. Shimizu, *StochSim website at University of Cambridge.* http://www.pdn.cam.ac.uk/groups/comp-cell/StochSim.html.
17. Loew, L.M. and J.C. Schaff, *The Virtual Cell: a software environment for computation cell biology.* Trends in biotechnology, 2001. **10**(10): p. 401-406.
18. Loew, L.M. and J.C. Schaff, *The Virtual Cell.* http://cmbi.bjmu.edu.cn/cmbidata/vcell/v-cell/Virtual%20Cell%20Development.htm: p. .
19. Plimpton, S. and A. Slepoy, *Chemcell: a particle-based model of protein chemistry and diffusion in microbial cells.* Sandia Technical Report SAND2003-4509, 2003.
20. Poolman, M.G., *ScrumPy: metabolic modelling with Python.* Systems Biology, IEE Proceedings, 2006. **153**(5): p. 375-378.
21. Poolman, M.G., *ScrumPy.* http://mudshark.brookes.ac.uk/index.php/Software/ScrumPy.
22. Sanford, C., et al., *Cell++ -- simulating biochemical pathways.* Bioinformatics, 2006. **22**(23): p. 2918-2925.
23. Schaff, J., et al., *A general computational framework for modelling cellular structure and function.* Biophysical Journal, 1997. **73**: p. 1135-1146.
24. Snoep, J.L. and B.G. Oliver, *JWS.* http://jjj.biochem.sun.ac.za/index.html.
25. Boulianne, L., et al., *GridCell: a stochastic particle-based biological system simulator.* BMC Systems Biology, 2008. **2**.
26. Boulianne, L., M. Dumontier, and W.J. Gross. *A stochastic particle-based biological system simulator.* in *Proceedings of the Summer Computer Simulation Conference (SCSC07).* 2007. San Diego, California.
27. Lok, L., *The need for speed in stochastic simulation.* Nature Publishing Group, 2008. **22**(8): p. 964-965.
28. Bergmann, F.T. and H.M. Sauro, *Comparing Simulation Results of SBML capable simulators.* Bioinformatics, 2008.

29. Stiles, J.R. and T.M. Bartol, *Monte Carlo Methods for simulating realistic synaptic microphysiology usingMcell (Chapter 4)*, in *Computational Neuroscience: Realistic Modeling for Experimentation*. 2001, E. D. Schutter, Ed. CRC Press, Boca Raton. p. 87-127.
30. Hubbard, D., *How to Measure Anything: Finding the Value of Intangibles in Business*. 2007: John Wiley & Sons.
31. SBML, *Systems Biology Markup Language*. www.sbml.org.
32. Shimizu, T.S., et al., *Molecular model of a lattice of signalling proteins involved in bacterial chemotaxis*. Nat. Cell. Biol., 2000. **2**: p. 792-796.
33. Lemerle, C., B.D. Ventura, and L. Serrano, *Space as the final frontier in stochastic simulations of biological systems*. FEBS Letters, 2005. **528**(8): p. 1789-1794.
34. Brown, R., *A brief account of microscopical observations made in the months of June, July and August, 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies*. Phil. Mag., 1828. **4**: p. 161-173.
35. Weisstein, E., *Markov Process*. MathWorld.
36. Elf, J. and M. Ehrenberg, *Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases*. Syst. Biol. **2**: p. 230-236.
37. IBM, *Cell Broadband Engine Programming Handbook*. 1.1 ed. 2007. 877.
38. Linklater, M., *Optimizing Cell Core*, in *Game Developer Magazine*. 2007. p. 15-18.
39. Perrone, M., *Introduction to the Cell Processor*, in *6.189 IAP*, IBM, Editor. 2007.
40. Chemistry, I.U.o.P.a.A., *Chemical Reaction* Internet edition ed. Compendium of Chemical Terminology.