THE UNIVERSITY
OF BIRMINGHAM

Acceleration of the Discrete Element Method on a Reconfigurable Co-processor

by

Benjamin Carrión Schäfer

A thesis submitted to the School of Engineering of
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Engineering
Department of Electronic, Electrical and Computer Engineering
The University of Birmingham
February 2002

Abstract

Granular materials are important for many different disciplines, e.g. geomechanics, civil engineering and chemical engineering. Many approaches have been used to model their behaviour, but one of the best and most important is the Discrete Element Method (DEM). The DEM was first developed during the 70's, but its widespread use has been hampered by its extremely computationally demanding nature.

The DEM can be run on a parallel computer by farming out different sub-domains onto different processors. However, particles transiting from one sub-domain to another create communication and synchronisation overheads which limit the speed-up achieved by parallel processing. Also, if some cells become much more heavily populated than others, then there will be inefficiencies due to load imbalance between the processors. As a result of these effects, the speed-up achieved by running the DEM on parallel processor computers is far less than linear.

This thesis describes work on the acceleration of the DEM using reconfigurable computing. A custom hardware architecture for the DEM has been designed and implemented on a Field Programmable Gate Array (FPGA) mounted on a reconfigurable computing card. The design exploits the low level parallelism of the DEM by using long, wide computational pipelines that compute many arithmetic operations concurrently. It also exploits the high level parallelism by overlapping the main computational tasks using domain decomposition techniques. Speed-ups of a factor of at least 30 per FPGA have been achieved for simulations involving 25,000 to 200,000 particles. A multi-FPGA system has been implemented that allows the full overlap of computation with communication, so that an almost linear speed-up can be achieved as the number of FPGAs is increased. The effect of the short wordlength arithmetic used in the FPGA has been investigated, and the accuracy of the simulations has been found to be acceptable.

To my brother Norbert,

Thank you very much for everything

Acknowledgments

This research has been supported financially by the University of Birmingham's interdisciplinary research fund, the Department of Electronic, Electrical and Computer Engineering and the Department of Civil Engineering.

I would first of all like to thanks my two supervisors Dr. Andrew H.C. Chan and Dr. Steven F. Quigley for all their support. Thank you very much for everything, which is quite a lot.

I would also want to thank Dr. Alonso Corona and Dr. Ignacio Llamas for their friendship. This would have not been the same without them.

To all the colleagues in room 439. Thank you very much for making the time spent in Birmingham so enjoyable.

Last but far from least, thank you very much to my family for their support from the far distance.

Contents

CHAPTER 1 INTRODUCTION **1**

1.1 INTRODUCTION 1

1.2 CONTRIBUTION OF THIS THESIS 3

1.3 THESIS ORGANISATION 4

1.4 REFERENCES 7

CHAPTER 2 THE DISCRETE ELEMENT METHOD **8**

2.1 INTRODUCTION 8

2.2 ANALYSIS OF THE BEHAVIOUR OF GRANULAR MEDIA 10

2.3 SIMULATIONS 10

2.4 ADVENT OF THE DISCRETE ELEMENT METHOD (DEM) 12

2.5 CONCEPTS OF THE DISCRETE ELEMENT MEHTOD 13

2.6 DEM ANALYSIS 17

2.7 PARALLEL ANALYSIS 24

2.8 SUMMARY AND CONCLUSIONS 28

2.9 REFERENCES 30

CHAPTER 3 SOFTWARE IMPLEMENTATION OF THE DEM **32**

3.1 INTRODUCTION 32

3.2 INITIALISATION OF THE SIMULATION 34

3.3 SIMULATOR FEATURES 42

3.4 RUNTIME COMPARISON BETWEEN THE FORTRAN AND THE C SIMULATOR 45

3.5 VALIDATION OF THE SIMULATOR 47

3.6 DISCUSSION 49

3.7 SUMMARY AND CONCLUSIONS 50

3.8 REFERENCES 50

CHAPTER 4 REVIEW AND ANALYSIS OF PARALLEL DEM IMPLEMENTATIONS 52

4.1 INTRODUCTION 52
4.2 BASIC IDEAS ABOUT PARALLELISM 53
4.3 PARALLEL DEM IMPLEMENTATIONS 57
4.4 MODELLING OF A MULTIPROCESSOR SYSTEM 69
4.5 SUMMARY OF THE PARALLEL DEM IMPLEMENTATIONS 79
4.6 USE OF FIELD PROGRAMMABLE GATE ARRAYS FOR THE DEM 80
4.7 SUMMARY AND CONCLUSIONS 81
4.8 REFERENCE 82

CHAPTER 5 HARDWARE IMPLEMENTATION OF THE DEM ON A FPGA 84

5.1 INTRODUCTION 84
5.2 MOTIVATION 85
5.3 FIELD PROGRAMMABLE GATE ARRAYS (FPGAS) 85
5.4 RECONFIGURABLE COMPUTING PLATFORM 89
5.5 HARDWARE IMPLEMENTATIONS 91
5.6 DATA FORMAT 93
5.7 IMPLEMENTATION CLASSIFICATIONS 95
5.8 LOW LEVEL PARALLELISM IMPLEMENTATION 95
5.9 HIGH AND LOW LEVEL IMPLEMENTATION 104
5.10 VALIDATION OF THE HARDWARE DESIGNS 120
5.11 DISCUSSION 124
5.12 SUMMARY AND CONCLUSIONS 125
5.13 REFERENCES 127

CHAPTER 6 SOFTWARE AND HARDWARE ANALYSIS 128

6.1 INTRODUCTION 128
6.2 SPEED-UP 129
6.3 DATA PRECISION 133
6.4 ERRORS IN ARITHMETIC OPERATIONS 137
6.5 ERROR PROPAGATION IN COMPUTER ARITHMETIC 140

6.6 ARITHMETIC ERROR ANALYSIS OF TE HARDWARE IMPLEMENTATION 145
6.7 COMPARISON OF BULK ERRORS IN SW AND HW 155
6.8 DISCUSSION 166
6.9 SUMMARY AND CONCLUSIONS 168
6.10 REFERENCES 168

CHAPTER 7 SUITABILITY OF THE HW DESIGN FOR A MORE COMPLEX DEM 170

7.1 INTRODUCTION 170
7.2 INSERTION OF WALLS 171
7.3 MULTIPLE RADII 182
7.4 3-DIMENSIONS 185
7.5 SUMMARY AND CONCLUSIONS 200
7.6 REFERENCES 201

CHAPTER 8 SCALABLE AND ALTERNATIVE IMPLEMENTATIONS OF THE DEM 202

8.1 INTRODUCTION 202
8.2 MULTI-FPGA DISTRIBUTED MEMORY SYSTEM 203
8.3 SHARED MEMORY SYSTEM 2012
8.4 ALTERNATIVE SINGLE FPGA IMPLEMENTATION 215
8.5 SUMMARY AND CONCLUSIONS 227
8.6 REFERENCES 228

CHAPTER 9 CONCLUSIONS AND FUTURE WORK 229

9.1 CONCLUSIONS 229
9.2 FUTURE WORK 233

Appendix A.1: PUBLICATION LIST

Table of Figures

CHAPTER 2THE DISCRETE ELEMENT METHOD (DEM) 8

Figure 2–1 Examples of Storage and transportation of granular media 9

Figure 2–2 Dem Flow chart 13

Figure 2–3 Balls in contact 14

Figure 2–4 Screenshots of two system (without and with grid) 17

Figure 2–5 Modified DEM flow chart with domain decomposition 18

Figure 2–6 Two cases if different number of balls in contact 19

Figure 2–7 Geometrical deduction of the maximum number of balls in contact for balls of same radius 22

Figure 2–8 Graphical representation of the force update equations 25

Figure 2–9 Example of a serial and parallel graphical representation of an algorithm 25

Figure 2–10 Graphical representation of the position update equations 26

Figure 2–11 Examples of domain decomposition in order to make use of the high level parallelism of the DEM 27

CHAPTER 3 SOFTWARE IMPLEMENTATION OF THE DEM 32

Figure 3–1 DEM Flow chart 34

Figure 3–2 Example of the simulator initialisation file 35

Figure 3–3 Screenshots of the simulators initial data after the particles have been generated 38

Figure 3–4 Screenshots of an initial state of the simulator after reading in the data file 39

Figure 3–5 Data structure of the particles 40

Figure 3–6 Data linked list structure 41

Figure 3–7 Simulator’s tool bar 42

Figure 3–8 Run time graph as a function of the number of boxes 43

Figure 3–9 Screenshot of the simulator with different grid sizes 44

Figure 3–10 Re-draw option window 45

Figure 3–11 Runtime comparison between the FORTRAN and the C simulator 46

Figure 3–12 Sequence of the collision of two balls 47

Figure 3–13 Initial and final state of the simulation 48

CHAPTER 4 REVIEW AND ANALYSIS OF PARALLEL DEM IMPLEMENTATIONS 52

Figure 4–1 Transputer Network Configuration 58

Figure 4–2 Pseudo Code for the Multi processor systems 59

Figure 4–3 Measured speed-up 60

Figure 4–4 Speed-up of the parallel implementation as a function of the number of processors 62

Figure 4–5 Domain decomposition for the Hopper discharge 63

Figure 4–6 Single-port and dual-port hopper 64

Figure 4–7 Computation time required for the individual tasks of the DEM simulation of the single-port hopper 65

Figure 4–8 Computation time required for the individual tasks of the DEM simulation of the dual-port hopper 65

Figure 4–9 Measured speed-up for the DEM simulations of the single-port hopper 66

Figure 4–10 Measured speed-up for the DEM simulations for the dual-port hopper 67

Figure 4–11 Example of the different regular domain decomposition types 70

Figure 4–12 Domain decomposition types in the multi-procession modelling SW 73

Figure 4–13 Simulation time for different number of processor systems 74

Figure 4–14 Simulation time for different number of processor showing the time spent by each unit(domain split into cells) 75

Figure 4–15 Initial and final conditions for a simulation decomposing the domain in columns and cells 76

Figure 4–16 Time needed to perform the last cycle in a simulation with 8 processors decomposition the domain in columns 77

Figure 4–17 Time needed to perform the last cycle in a simulation with 8 processors decomposition the domain in cells 77

Figure 4–16 Speed-up for different initial velocities 79

CHAPTER 5 HARDWARE IMPLEMENTATION OF THE DEM ON A FPGA 84

Figure 5–1 Three FPL waves, PLDs, FPGAs/CPLD and CSoC 87

Figure 5–2 Field Programmable Gate Array (FPGA) internal structure 89

Figure 5–3 RC100-PP Picture 90

Figure 5–4 RC100-PP Block Diagram 91

Figure 5–5 System layout 92

Figure 5–6 Hardware Software selection 93

Figure 5–7 Data format 94

Figure 5–8 Low level parallelism FPGA Implementation block diagram 95

Figure 5–9 Balls in contact 96

Figure 5–10 Neighbour check model 97

Figure 5–11 Forces update unit internal structure 99

Figure 5–12 Velocity and Position update unit internal structure 100

Figure 5–13 Memory map for the low level parallelism implementation 102

Figure 5–14 FPGA's internal memory map 103

Figure 5–15 Domain decomposition 105

Figure 5–16 High and low level parallelism FPGA implementation block diagram 106

Figure 5–17 Scheduling of the computation 107

Figure 5–18 Simulation example of the adaptive cell boundaries 110

Figure 5–19 Memory map for the high and low level parallelism implementation 111

Figure 5–20 High and low level parallelism scheduling 113

Figure 5–21 Time needed for each task 113

Figure 5–22 Graph of clock cycles needed to compute t(forces) t (pos)+t(interface) and t(cc) for a different number of contact check units. 116

Figure 5–23 Comparisons of experimental and analytical values to compute the contact checking for different number of contact check units. 117

Figure 5–24 Screen shot of the initial state of the hardware debugger 121

Figure 5–25 Screen shot of a debugged system of 50 balls after 20 cycles 122

Figure 5–26 pseudo code of under and overflow registration 123

Figure 5–27 Underflows in the forces and position update units as a function of the number of bits 124

CHAPTER 6 SOFTWARE AND HARDWARE ANALYSIS 128

Figure 6–1 Initial state of the 500 domain assembly for the SW (a) and for the HW (b) 130

Figure 6–2 Initial state of the system of 50,000 particles 131

Figure 6–3 Graphical representation of the measured and ideal speed-up 132

Figure 6–4 16-bit data format 133

Figure 6–5 Chopping example 136

Figure 6–6 Rounding example and comparison with chopping 136

Figure 6–7 Rounding carry example 137

Figure 6–8 Truncation/Round off propagation model 140

Figure Error! No text of specified style in document.**-1** Noise distribution model 140

Figure Error! No text of specified style in document.**-10** Normal distribution function 142

Figure 6–11 Velocity and Position update unit internal structure 146

Figure 6–12 Forces update unit internal structure 147

Figure 6–13 Special cases for cosine and sin 148

Figure Error! No text of specified style in document.**-2** Expanded flow graph for the F_x force pipeline in the forces update unit 149 151

Figure Error! No text of specified style in document.**-3** Expanded flow graph for the x coordinate pipeline in the position update unit 154

Figure 6–16 SW window to measure the difference between the SW and the HW simulation 155

Figure 6–17 Error accumulation in the hardware implementation compared to the software implementation for the x and y coordinates 157 158

Figure 6–18 Initialisation file for the simulations 159

Figure 6–19 Balls' contact model 159

Figure 6–20 Energy window once the software and hardware system have been generated

Figure 6–21 System's Energy progression with damping for the software and the hardware implementation 160

Figure 6–22 Systems Energy progression without damping for the software and hardware implementation 162

Figure 6–23 Behaviour of the centroids difference between the software and the hardware implementation 163 164

Figure 6–24 Periodic domain example 165

Figure 6–25 Average velocity progression
Figure 6–26 System energy for float and double SW simulation

CHAPTER 7 **SUITABILITY OF THE HW DESIGN FOR A MORE COMPLEX DEM** **170**

Figure 7–1 Dataflow diagram for the DEM with walls 171
Figure 7–2 Wall description 172
Figure 7–3 Ball-Wall contact detection 174
Figure 7–4 Ball-Wall contact for Vertical/Horizontal walls 175
Figure 7–5 Equations to compute the forces between a wall and a ball 176
Figure 7–6 Equations to compute the new position of the wall 177
Figure 7–7 Balls in contact with different radii 182
Figure 7–8 Contact detection for particles of different radius 183
Figure 7–9 Suggested Contact balls' data structure for systems with balls of different radii in the FPGA 184
Figure 7–10 3-D Contact checks 187
Figure 7–11 Equations to compute the forces between two particles in 3-D 190
Figure 7–12 Equations for the 3-D Position Update 191
Figure 7–13 Graphical Representations of the 3-D position update equations 193
Figure 7–14 Screen-shot of the 3-D software simulator 197
Figure 7–15 Graphical representation of the Simulation time of 2-D and 3-D system 198

CHAPTER 8 **SCALABLE AND ALTERNATIVE IMPLEMENTATIONS OF THE DEM** **202**

Figure 8–1 Distributed Memory Multi-FPGA system 204
Figure 8–2 Two RC1000-PP system 204
Figure 8–3 Board selection for the Multiple FPGA design 205
Figure 8–4 Screenshot of the software program for the multiple FPGA design. The domain is split in two equally loaded parts 206
Figure 8–5 Columns that need to be transferred from one board to another after every cycle 207
Figure 8–6 Domain mapping to the 4 memory units of RC1000-PP board. 207
Figure 8–7 Influence of a finer and coarse grained board memory 210
Figure 8–8 Example of a shared memory system 2012

Figure 8–9 FPGAs' memory accesses 214
Figure 8–10 Runtime reconfigurable architecture 217
Figure 8–11 Reconfigurable sequence 218
Figure 8–12 Number of columns to be cached into the FPGA 219
Figure 8–13 Internal FPGA structure with microprocessors 222
Figure 8–14 Hardware implementation using an FPGA with embedded microprocessors 223
Figure 8–15 Computation time of the three main task, replacing the forces update unit with 4 microprocessors 225

CHAPTER 9 CONCLUSIONS AND FUTURE WORK 229

Figure 9–1 Mapping of the domain decomposition on the FPGA's internal memory 230

List of Tables

CHAPTER 2 THE DISCRETE ELEMENT METHOD (DEM) 8

Table 2–1 Timing analysis for a system with 500 particles ran for 1000 time steps with and without grid 18

Table 2–2 Arithmetic operations needed for the contact check 19

Table 2–3 Arithmetic operations of the force update functions 22

Table 2–4 Arithmetic operations of the coordinate and velocity update function 23

CHAPTER 3 SOFTWARE IMPLEMENTATION OF THE DEM 32

Table 3–1 Comparison of the simulation time needed for an assembly of 500 particles ran for 1000 cycles depending on the amount of times the assembly is re-drawn. 61

Table 3–2 Runtime comparison between between the FORTRAN and the C simulator 46

Table 3–3 Energy comparison between the original FORTRAN and the new C simulator 49

CHAPTER 4 REVIEW AND ANALYSIS OF PARALLEL DEM IMPLEMENTATIONS 52

Table 4–1 Listing of the low and high implementations of the DEM on different HW platforms 69

CHAPTER 5 HARDWARE IMPLEMENTATION OF THE DEM ON A FPGA 84

Table 5–1 Hardware requirements for the low level parallelism units 101

Table 5–2 Hardware resources used for by this implementation 117

Table 5–3 Growth of ideal number of balls/column as a function of the number of contact check units to make t(cc) = t(position)+t(r/w). 119

Table 5–4 Relation of number of balls allowed in the system to make t(cc)=t(pos)+ t(interface) and its memory requirements 120

CHAPTER 6 SOFTWARE AND HARDWARE ANALYSIS **128**

Table 6–1 Simulation time for the SW and the HW implementation for 500 particles 130

Table 6–2 Run time for the software and hardware simulation and speed-up results 131

Table 6–3 Worst case analysis for the multiplication as a function of the input values 138

Table 6–4 Worst case analysis for the division as a function of the input values 138

Table 6–5 Worst case analysis for the subtraction as a function of the input values 139

Table 6–6 Simulation time for 10,000 and 20,000 particles for 1000 cycles in single and double precision floating-point arithmetic 166

CHAPTER 7 SUITABILITY OF THE HW DESIGN FOR A MORE COMPLEX DEM **170**

Table 7–1 Minimum set of parameters to describe a wall 172

Table 7–2 Rest of parameters needed once the wall is interacting will the balls 173

Table 7–3 Arithmetic operations needed the check for contacts between walls and balls 174

Table 7–4 Number and types of arithmetic operations to check for contacts between balls and walls, only for vertical and horizontal walls. 175

Table 7–5 Additional arithmetic operations needed to calculate the forces between a wall and a ball, compared to the arithmetic operations needed forces between two balls. 177

Table 7–6 Arithmetic operations needed to compute the new position of a wall 178

Table 7–7 Number of Xilinx slices needed to perform different arithmetic operations 178

Table 7–8 Number of Xilinx slices needed for the additional arithmetic operations to include walls 179

Table 7–9 Variables to describe a 3-D particle 185

Table 7–10 Number of arithmetic operations needed for the contact check in 3-D 187

Table 7–11 Number of Arithmetic operations involved in the3-D forces update units for particles of the same radius. 190

Table 7–12 Number of Arithmetic operations involved in the3-D position update units for particles of the same radius 192

Table 7–13 Comparison between the arithmetic operations for the 2-D and the 3-D case in each task 194

Table 7–14 Slices needed to accommodate arithmetic operations for 3D 195

Table 7–15 Runtime simulation results for 2-D and 3-D assemblies with the same 198

properties
Table 7–16 Theoretical speed-up of the 3-D SW simulator and the theoretical HW design 1 199

CHAPTER 8 SCALABLE AND ALTERNATIVE IMPLEMENTATIONS OF THE DEM 202

Table 8–1 Example of the number of boards that can work in parallel without having to stall any operation in any of the FPGAs. 209
Table 8–2 Example of the number of boards that can work in parallel without having to stall the operation of any FPGA for 8 memory units instead of 210
Table 8–3 Comparisons of speed-up obtained by hardware DEM for a single FPGA and two FPGAs compared to an optimised software version. 211
Table 8–4 Number of units that can be implemented in the reconfigurable area 216
Table 8–5 Values of the time needed to compute the cc, forces and position update versus the time needed to reconfigure the 50% of the Xilinx XCV2000E. 220
Table 8–6 Number of arithmetic operations involved in the forces update unit 223
Table 8–7 Number of operations needed for each arithmetic operation. 224
Table 8–8 List with the number of units implemented in parallel. 224

Glossary of Abbreviations

2-D	Two **D**imensional
3-D	Three **D**imensional
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
CC	**C**ontact **C**heck
CISC	**C**omplex **I**nstruction **S**et **C**omputer
CLB	**C**onfigurable **L**ogic **B**lock
DEM	**D**iscrete **E**lement **M**ethod
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GUI	**G**raphical **U**ser **I**nterface
HDL	**H**ardware **D**escription **L**anguage
HW	**H**ard**w**are
IC	**I**ntegrated **C**ircuit
IP	**I**ntellectual **P**roperty
I/O	**I**nput/**O**utput
KCM	Constant **C**oefficient **M**ultiplier
LAB	**L**ogic **A**rray **B**lock
LC	**L**ogic **C**ell
LE	**L**ogic **E**lement
LUT	**L**ook **U**p **T**able
MISD	**M**ultiple **I**nstructions **S**ingle **D**ata
MIMD	**M**ultiple **I**nstructions **M**ultiple **D**ata
PAL	**P**rogrammable **A**rray **L**ogic
PC	**P**ersonal **C**omputer
PCI	**P**eripheral **C**omponent **I**nterconnect
PLD	**P**rogrammable **L**ogic **D**evice
Ulp	**U**nit of **L**east **P**recision
RISC	**R**educed **I**nstruction **S**et **C**omputer
RTR	**R**un **T**ime **R**econfiguration
SIMD	**S**ingle **I**nstructions **M**ultiple **D**ata
SISD	**S**ingle **I**nstructions **S**ingle **D**ata
SW	**S**oft**w**are

VHDL **V**ery High Speed Integrated Circuit **H**ardware Description **L**anguage

CHAPTER 1

INTRODUCTION

1.1 Introduction

The number of transistors that can be integrated onto a single silicon die tends to double approximately every 18 months, just as Intel's co-founder Gordon Moore predicted more than 20 years ago [1]. This increase in density is accompanied by a corresponding improvement in speed. This has led to a widespread availability of very powerful computer equipment at low cost, a development that has affected the design approaches used in many branches of engineering. One important consequence is that much wider use is made of simulator programs. Simulators have the great advantage of being able to test a system without having to actually build it. This saves an enormous amount of time and therefore money. Nevertheless, there are still some applications where the available computing power of standard computers is still not sufficient to perform many of the desired simulations. One of these application areas is the use of the Discrete Element Method for modelling the behaviour of granular materials.

Granular materials can be found everywhere in life, and their study is important to many different disciplines, such as geomechanics, civil engineering and chemical engineering. There are many different approaches to model their behaviour (e.g. analytical, physical and

numerical). The numerical techniques are the most powerful, as they have the greatest flexibility, and they also provide full visibility of the internal behaviour of the medium at every stage.

The Discrete Element Method (DEM) is a numerical method to model the behaviour of particle assemblies. The DEM was first developed in the 1970's, but its widespread use has been hampered by its extremely computationally demanding nature. The DEM considers every particle as an individual body and computes the total force applied to each particle. From this, using Newton's second law, the acceleration of each particle is established; this can be integrated to give each particle's velocity; this in turn can be integrated to provide an updated position. Each particle's force interaction, acceleration and position are calculated individually at each time step. The assumptions underlying the method are only correct if no disturbances can travel beyond the immediate neighbours of a particle within one time step. This generally means that the time step must be limited to a very small value, thus making the DEM *extremely* computationally expensive.

The particles may be bonded together to represent, for example, rock, or they may remain unbonded to represent, for example, soil. Bonded together they can represent entire structures, such as dams or bridges. It is has even been suggested [2] that the DEM may in future replace the more popular continuum methods such as the Finite Element Method and the Finite Difference Method, as these have two main drawbacks. Firstly a suitable stress-strain law may not exist; secondly, localised features, such as cracks, are difficult to model with the continuum approaches. However in order to use the DEM to simulate entire engineering structures, which may involve millions of particles, enormous computing power is required.

Although the DEM is extremely computationally expensive, it exhibits an extraordinarily high degree of parallelism. Many attempts have therefore been made to run the DEM on multiprocessor computer systems. Ideally, one would hope to achieve a speed-up of the simulation that is proportional to the number of processors used (linear speed-up). However, synchronization and communication overheads, as well as load balancing problems, mean that these systems underachieve the ideal limit when the number of

processors is large. Other approaches to allow the simulation of realistic particle problems (typically hundreds of thousands of particles) are therefore needed.

The impact of Moore's law has been felt not only by computer processor chips, but also by programmable logic. Originally programmable logic was used only for small-scale glue logic applications. However, the complexity and speed of programmable logic, in particular Field Programmable Gate Arrays (FPGAs), has increased enormously over the last decade. FPGAs have now achieved sufficient logic density that they can be used to implement an entire complex system with minimal off-chip resources.

One promising application area for these devices is to form FPGA-based reconfigurable co-processors within standard computers, which can be used for algorithm acceleration. This approach is known as *reconfigurable computing*[1]. FPGA co-processors have much lower cost and greater flexibility than ASIC hardware (albeit with inferior performance). For the right type of application, a reconfigurable computer can rival the expensive parallel computers that are normally used to accelerate computationally expensive algorithms. FPGAs thus open a new window to low cost hardware acceleration.

The DEM has properties that suggest that it may be suitable for acceleration using FPGAs: it exhibits a enormous degree of parallelism, and can be processed using short wordlength arithmetic. It is therefore tempting to examine how well this algorithm would map into an FPGA coprocessor.

1.2 Contribution of this Thesis

This thesis presents a study of the use of reconfigurable computing using FPGAs to accelerate the DEM. The major contributions made by this work are as follows:

[1] Some authors use the term *reconfigurable computing* to refer only to approaches which use run-time reconfiguration of the FPGA, whereas others apply the term to any use of an FPGA co-processor. For most of this work, the second definition is used. However, in chapter 8 an investigation is presented into the application of run time reconfiguration to the DEM problem.

1. A novel approach was taken to accelerate the 2-D DEM by designing a dedicated hardware architecture on an FPGA. A simpler architecture was developed first, which only made use of the low level parallelism of the DEM. Subsequently, a more sophisticated architecture was designed, which exploits not only the low level, but also the high level parallelism, and involves decomposing the domain into different regions. This decomposition is adaptively optimised in order to provide good load balancing.
2. In order to achieve even greater speed-ups, a multi-FPGA design has been implemented. This shows that communication and computation by the FPGAs can be completely overlapped, thus achieving good scalability of the speed-up.
3. Predictions are presented as to how well the hardware architecture would map onto more sophisticated FPGAs. The effects of additional resources, such as embedded multipliers and embedded microprocessors are considered.
4. The effects of short wordlength arithmetic on the DEM results has been investigated, and performance of hardware based on 16-bit fixed point arithmetic has been found to be acceptable. For DEM simulations the result of interest is the behaviour of the bulk, not the behaviour of the individual particles. For the bulk behaviour, the DEM is to a large extent a self-correcting algorithm, thus making low numerical precision tolerable. A relatively low wordlength was necessitated by the limitations of the FPGA hardware available.
5. The prototypes built for the hardware implementations were limited by the available FPGA resources, and therefore a relatively simple DEM problem was implemented (using a 2-D domain containing no walls, with all particles having the same radius, and using a simple interaction law). An analysis has been carried out that demonstrates the resource requirements that would be needed to extend the architecture to a more complicated DEM problem.

1.3 Thesis Organisation

Chapter 2 introduces the basic concepts of the Discrete Element Method (DEM). The principle stages of the algorithm are formulated, and their asymptotic complexity is discussed. The most time consuming task involves a search of the domain for contacts between particles, which has a complexity of $O(N^2)$, where N is the number of particles in

the domain. If the one large domain is decomposed into multiple smaller domains, the expense of the $O(N^2)$ search is greatly reduced. Domain decomposition also provides an obvious and natural way to parallelise the DEM, by allocating different domains to different processors. However, processing the different domains on different processors leads to a communication overhead as particles transition from one domain to another, which slows down the speed-up achievable by parallel computers. Chapter 2 discusses approaches to domain decomposition, and illustrates the effectiveness of the method. It also provides an assessment of the expenses associated with domain decomposition.

Chapter 3 presents the development of the software DEM simulator used in this work. This is based on a standard public domain FORTRAN code for the DEM, but is re-written in C, and contains numerous enhancements and improvements. These include the use of C's superior data structures to accelerate the simulation, a visual interface, an interface to the hardware version, and a debug mode in which the hardware and software versions can be run in synchrony and their results compared. Also, the software written for this project has the ability to emulate the performance of the code on parallel machines of different capabilities (e.g. processor speed, and inter-processor communications bandwidth). The simulator tracks the time taken and the amount of communication generated for the processing of each sub-domain, and uses this to produce reports on how well the simulation would speed up on various types of parallel machine.

Chapter 4 presents a brief review of issues influencing speed-up in parallel processing. It then surveys the various attempts at parallelisation of the DEM that can be found in the literature. Most of these attempts have chosen problems that are anomalously favourable for parallel processing, e.g. domains decomposed into vertical strips in which particles fall under gravity. This gives rise to very few transitions between subdomains, which means that communications overhead is very low, and good load balance is always achieved. In these "nice" problems, high speed-ups can be achieved, but for more realistic problems the speed-up is far less than linear. At the end of chapter 4, the simulator developed in chapter 3 is used to assess the "niceness" of a variety of problem types, domain decompositions, and initial conditions. The dependence of the communication and synchronisation overheads on the choice of problem and domain decomposition is quantitatively assessed.

Chapter 5 describes two hardware designs that were implemented in a reconfigurable computing board. The first is a simpler implementation, which only makes use of the low level parallelism of the algorithm. The second is a more complex implementation, which exploits not only the low level parallelism, but also the higher level by performing the major tasks in parallel using domain decomposition techniques. In order to allow the hardware implementation to fit on the available FPGA boards, a number of compromises had to be made. A limited formulation of the DEM was implemented, which performed only 2-D simulations of assemblies whose constituent particles all have the same radius and whose domains contain no walls. Also, 16 bit fixed point arithmetic was used.

In chapter 6 both hardware implementations are compared with the software implementation in terms of speed-up and numerical precision. The complex hardware version gives a factor of 30 speed-up compared to the software version, for a simulation scenario that was deliberately chosen to be as favourable as possible for the software. For other scenarios, the speed-up achieved by the hardware is much greater. Chapter 6 also provides a review of issues underlying error propagation in finite precision arithmetic, and demonstrates that the hardware is free of pathological cases that cause catastrophic loss of accuracy. The actual loss of precision caused by the 16-bit arithmetic is estimated, and the estimate is confirmed by measurement of simulations running on the hardware.

Chapter 7 describes how the design can be modified to relax some of the restrictions that were used for the design in chapter 5. The use of 3-D, variable particle radius, and walls within the domain are considered. Projections are made as to how much hardware is required, and what speed-up can be expected.

Chapter 8 presents a design of a system using multiple FPGA boards to achieve a higher speed-up. A system using two FPGAs was actually implemented, and its results are used to project how well the design could scale to a system using many FPGA boards. Chapter 8 also discusses how well the design could be adapted to take advantage of the properties of more sophisticated FPGAs. Application of run-time reconfiguration and embedded microprocessors and multipliers is considered.

Chapter 9 discusses the conclusions of the study, and offers some directions for possible future work.

1.4 References

[1] Moore, G. *"Cramming more components onto integrated circuits",* Electronics, Volume 38 Number 8, April 19, 1965

[2] Cundall, P.A. *"A discontinuous future for numerical modelling in geomechanics?"* Proceedings of the Institution of Civil Engineers, Geotechnical Engineering 149, Issue 1 Pages 41-47, January 2001.

Chapter 2

THE DISCRETE ELEMENT METHOD (DEM)

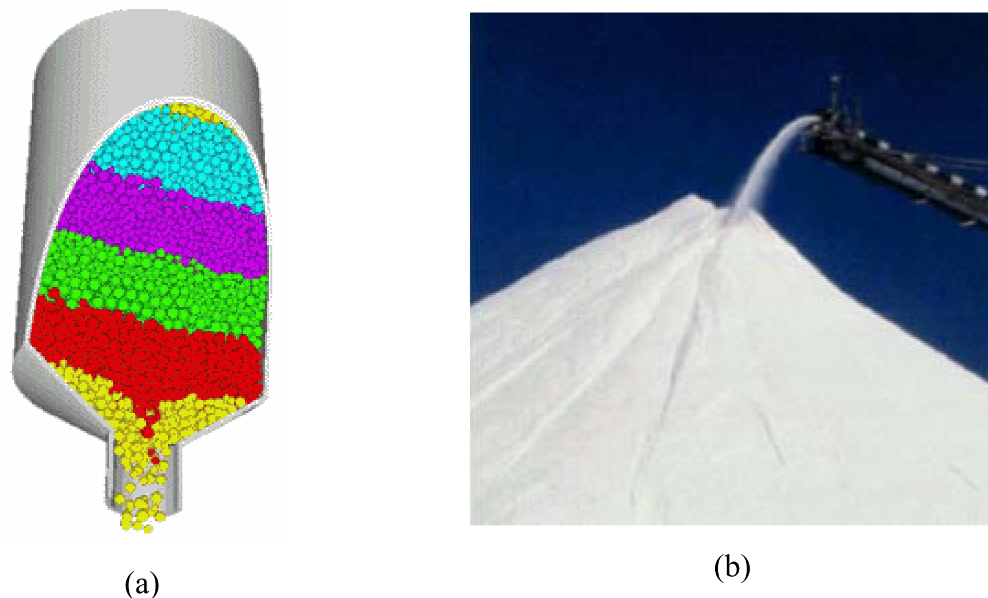
2.1 Introduction

Granular materials can be found everywhere in life. They appear in civil engineering structures in the form of, for example, sand and gravel, in the chemical and pharmaceutical industry, e.g. pills, and in the agricultural industry as all kinds of grain. Their behaviour has always interested human beings, but their study has been hampered because of their distinctive properties. Numerical techniques started to appear in the early 1970's [1], but the lack of computational power made it impossible to simulate real problems in sufficient detail. With the increase of computing power these numerical techniques have become more and more important, but there is still not enough computing power to solve large problems, which involve millions of particles in 2-D and in 3-D.

Granular materials can be defined as large conglomerations of discrete non-biological macroscopic particles. (For biological entities a number of difficulties arise. Firstly, for collections of animals or plants, individual entities may be capable of autonomous self-directed motion. For macromolecules, interactions forces are non-local, e.g. Van der Waal forces). Typically the radius of such particles has to be at least 1 μm. Granular materials

behave differently from any other familiar form of matter. Like a liquid, they can flow and assume the shape of the container, and like a solid, they can support weight; some can support a tensile stress; others cannot [2]. They can therefore be considered as a state of matter in their own right.

Many of the raw materials used in the food, chemical and pharmaceutical industries are granular media (illustrated in Figure 2–1). With the advent of modern machinery, the speed at which granular raw materials are processed has increased dramatically. This increase in speed has greatly increased the chance of damage to the fine particles during processing. In order to optimise the speed of production, and to reduce the amount of damage caused to



(a) (b)

Figure 2–1 *Examples of storage and transportation of granular media*

(a) Particles in a hopper (b) Transporting particles in conveyer belt

the particles, the effect of these mechanical interactions needs to be known. In many cases, for the purpose of processing speed-up, water and other fluid may be added which would alter the surface energy and adhesion between particles. Other manufacturing processes, e.g. in the automotive industry, rely on casting large metal parts in carefully packed beds of sand. Yet the technology for handling and controlling granular materials is poorly developed. Estimates show that 60% of the capacity of many industrial plants is wasted due to problems related to the transport of these materials from one part of the factory floor

to another [3]. Hence even a small improvement in our understanding of how granular media behave should have a profound impact for industry. All these require an understanding of the microscopic mechanical properties and the behaviour of their interactions so that the macroscopic behaviour of the bulk can be understood.

2.2 Analysis of the Behaviour of Granular Media

Granular materials are formed of distinct particles, which displace independently from one another and interact with each other only at the contact points [5]. The discrete nature of the granular materials leads to a complex behaviour under conditions of loading and unloading.

Real physical tests on granular materials have the advantage of getting precise results, but have some serious drawbacks. Primarily, the internal stresses cannot be measured accurately and must be estimated from the boundary conditions. Secondly it is almost impossible to repeat two completely identical experiments. These drawbacks led to the development of theoretical models in order to study the behaviour of these materials. These models consist of assemblies of discs or polygons (in 2 dimensions) or spheres or polyhedra (in 3 dimensions). These models are simulated using physical, analytical or numerical means [4].

The numerical modelling approach is the most powerful of the modelling techniques as it is more flexible than analytical modelling and has the advantage over physical modelling that any data can be accessed at any stage of the experiment. The major drawback of this method is that it is computationally very expensive and therefore very time consuming.

2.3 Simulations

In order to simulate the behaviour of a granular material, a suitable model has to be developed first. A model is a mathematical representation of a physical problem in a certain context. Models vary in their accuracy, but no model is perfect: the only perfect model would be the real system itself. Once the model is generated it has to be validated with real experiments in order to check for its correctness and robustness in the area of interest. A simulation can therefore be defined as the implementation of a particular model

in a computer. The solution of a typical model will take a predictable period of time before it delivers a result. It is therefore necessary to distinguish between:

- The real time that a physical process requires to complete a given action
- The computation time, which is the time needed by the computer to simulate the same physical action.

A process that happens in nature in a few seconds may take hours or even days to be simulated on a computer. The simulation time is of course software and hardware dependent. The same software simulator will run much faster on a faster workstation. However, an optimised software version can run faster on a slower workstation in terms of real time, than an un-optimised software version running on a fast workstation. In order to have a fast simulation, software and hardware have to be matched as much as possible.

The first numerical simulations of granular materials appeared at the end of the 1970's [5]. Two approaches were taken:

- The continuum approach, which considered the granular assembly as a continuum. (The success of the continuum approach for civil engineering problems reflects the fact that problems involving soils are of a large scale, for which the discrete nature of the soil does not seem to play an important role). This approach is only valid for certain types of problem.
- The discrete approach, which considered each individual particle as an individual entity.

The disadvantage of the continuum approach is that the discrete nature of the particles is not captured, and that cracks and rupture surfaces are not well captured by this approach [6]. On the other hand, discontinuous models, while treating these issues much better, are computationally very expensive, with simulations of thousands of particles taking hours and even days to finish. In the discrete approach every single particle is considered as an entity by itself, which moves following the physical laws of the domain.

2.4 Advent of the Discrete Element Method (DEM)

Cundall introduced the Discrete Element Method (DEM) in 1971 [7]. This numerical method considers every particle as a separate entity. The interaction force, acceleration and movement of each particle are calculated individually at each time step. The assumptions underlying the method are only correct if no disturbance can travel beyond the immediate neighbours of a particle within one time step. This is due to the explicit nature of the method. This generally means that the time step must be limited to a very small value. There are two main types of numerical time integration scheme:

- Explicit
- Implicit

An explicit method does not require the solution of the global equation. Therefore the information is transmitted from one point to another one time step at a time. If the time step is too large, excessive extrapolation would result, and the method can become unstable. There is therefore a critical maximum time step. Implicit methods, by contrast, involve the solution of the global equation, but have superior numerical stability.

As computing power increases, so does the number of applications that can be modelled reasonably using the DEM. An even higher growth is expected during this decade as computing power keeps growing, and as the method starts to be used in order to model entire engineering structures (such as dams and tunnels), built of particles bonded together to represent solid material. It is further suggested that continuum methods will be replaced by particle approaches in the future [6], as these capture the behaviour of localized cracks much better than the continuum approach, and a suitable stress-strain law for the material may not exist or the law may be excessively complicated with many obscure parameters.

The main drawback to the application of particle methods to large-scale problems is that their very high computational demands limit the size of system that can be simulated within a feasible timescale. Also, time must be spent calibrating the laws by which the micro-structure affects the overall macro-structure behaviour.

2.5 Concepts of the Discrete Element Method

Cundall and his co-worker Strack [1][5][7] developed the Discrete Element Method (DEM) in the seventies to model the behaviour of granular materials. The method is based on the assumption that particles only exert forces on one another when they are in contact. A simulation starts by assuming some initial configuration of particle positions, and then computes which of the particles are touching. The simulation then proceeds by stepping in time, applying the sequence of operations in Figure 2–2 at each step. The force between two particles can be calculated from the strength of the contact between them. The resultant force on a particle is the vector sum of the forces exerted by each of its neighbours. Once the resultant force on each particle has been computed, it is simple to compute the acceleration, the velocity and the position increment for each particle. Finally, the list of which particles are in contact must be re-computed. The force interaction, acceleration and movement of each particle are calculated individually at each time step. The assumptions underlying the method are only correct if no disturbance can travel beyond the immediate neighbours of a particle within one time step. This generally means that the time step must be limited to a very small value (of the order of milliseconds for the stiffness and density of a typical material, though using scaled stiffness or density can change its value). This restriction is due to the explicit nature of the method and it makes the DEM extremely computationally expensive, since many time steps are needed if the dynamic behaviour of the system is required to be modelled accurately.

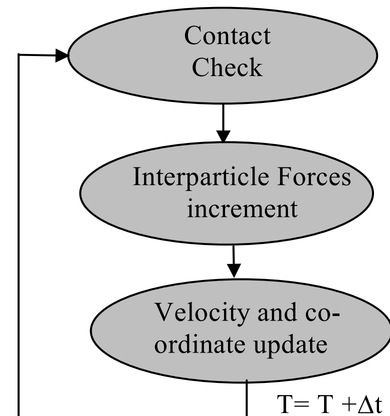


Figure 2–2 *DEM Flow chart*

This method has been widely used in many applications, such as silo flows [8], rock fracture and the collapse of buildings [9]. A detailed description of the three main steps involved in the DEM is given in the next section for a two-dimensional case. For the purpose of this explanation, the domain is assumed to be two dimensional, and the particles are assumed to circular discs. The extension to three dimensions is discussed in chapter 7.

2.5.1 Contact Check

In order to detect if two particles are in contact the following equation has to be solved for circular discs in 2-D:

$$\Delta n = R_1 + R_2 - \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \geq 0 \qquad \textbf{\textit{Eq. 2–1}}$$

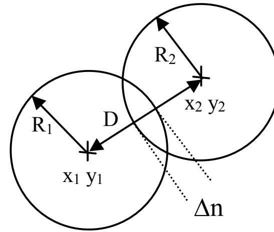


Figure 2–3 *Balls in contact*

Here x_i and y_i are the co-ordinates of each particle centre and R_1 and R_2 are the respective radii, D is the distance between the centres and Δn the separation or overlap of the two particles. If Δn is positive or zero, then the balls are in contact, whereas a negative value of Δn indicates that the balls are not in contact.

2.5.2 Inter-particle Forces Increment

Once the contact list for a particle has been established, the total force acting on it can be determined. For every contact identified between two particles, the resulting force can be calculated once the force-displacement law is known. For this study, the simplest possible force-displacement law is adopted: the resulting force between two balls is linearly proportional to the indentation Δn between the balls. (This is not exactly correct in reality, as the contact area will increase with the amount of contact thus rendering the force-displacement law non-linear. Although many advanced interaction laws, such as the Hertzian law, have been proposed [10], they add to the complexity to the calculations, but do not alter substantially the arguments put forward in this thesis). The force displacement law used for each ball is as follows:

$$F_{xi} = k_n \Delta n_{xi} \qquad \textbf{\textit{Eq. 2–2}}$$

$$F_{yi} = k_s \Delta n_{yi} \qquad \textbf{\textit{Eq. 2–3}}$$

$$M_i = F_{si} R \qquad \textbf{Eq. 2–4}$$

where k_i is the stiffness (subscript n for normal and s for shear), n_{xi} and n_{yi} are respectively the x and y components of the current ball indentation against particle i, F_{xi} and Fy_i are the components of the force caused by the interaction with ball i, M_i is the moment acting on the current ball due to the ith ball, F_{si} is the shear force acting on the current ball due to the ith ball and R is the ball's radius. The index i runs from the first to the last ball on the present ball's adjacency list, so the resultant force on a ball is the vector sum of the forces caused by each contact with its neighbours.

$$F_x = \sum_i F_{xi} \qquad \textbf{Eq. 2–5}$$

$$F_y = \sum_i F_{yi} \qquad \textbf{Eq. 2–6}$$

$$M = \sum_i F_{si} R \qquad \textbf{Eq. 2–7}$$

It should be noted that Eq. 2–7 is only correct if the rotation involved is small, since the direction of F_{si} changes with the rotation.

2.5.3 Velocity and Co-ordinate Update

Once the resultant forces of each ball have been calculated by summing the forces of all contacts in vectorial form for every ball, these forces can be used to find the new accelerations using Newton's second law:

$$a_x = F_x / m \qquad \textbf{Eq. 2–8}$$

$$a_y = F_y / m \qquad \textbf{Eq. 2–9}$$

Where F_x is the resultant force in the x-direction, F_y is the resultant force in the y-direction, m is the mass of the particle and a_x and a_y are the acceleration in the x and y-directions respectively.

These accelerations are integrated to obtain the velocities in the x and y directions as well as the rotational velocity using the moment of inertia of the particle I:

$$v_x = v_{x0} + a_x \Delta t$$ ***Eq. 2–10***

$$v_y = v_{y0} + a_y \Delta t$$ ***Eq. 2–11***

$$\dot{\theta} = \dot{\theta}_0 + \left(M / I \right) \Delta t$$ ***Eq. 2–12***

The time step, as is the case for all explicit time integration schemes, has to be limited to a small value in order to retain numerical stability. For the DEM, the constraint is that the time step must be sufficiently small that no disturbances can travel beyond one contact in one time step. The critical time step for each particle can be calculated from its stiffness and mass properties as shown in Eq. 2–13.

$$T_{critical} = 2\sqrt{mass / stiffness}$$ ***Eq. 2–13***

The critical time step of the whole system is limited by the smallest of the critical time steps of the individual particles. The new coordinates can be found by adding the original coordinates to the incremental displacement obtained by integrating the calculated velocities.

$$u_x = u_{x0} + v_x \Delta t$$ ***Eq. 2–14***

$$u_y = u_{y0} + v_y \Delta t$$ ***Eq. 2–15***

$$\theta = \theta_0 + \dot{\theta} \Delta t$$ ***Eq. 2–16***

It should be noted that both displacements and accelerations are defined at the time points which are at the beginning and the end of the time steps, and the velocities are defined at the mid-point of the time intervals.

2.6 DEM Analysis

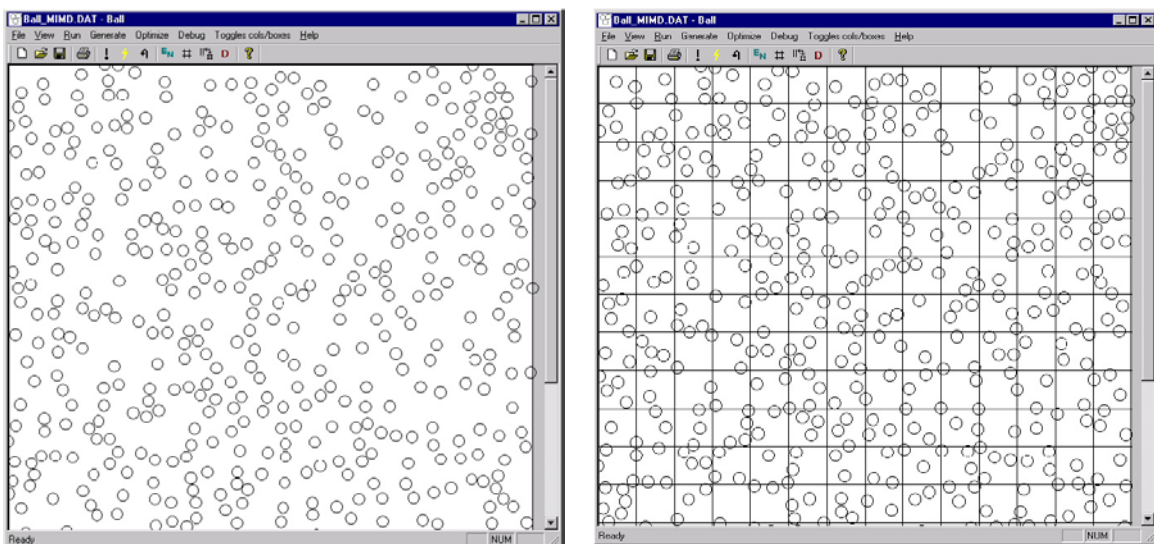
As can be seen from Figure 2–2 the DEM consists of three basic steps:

1. **Contact checking:** to detect the particles that are in contact
2. **Forces update:** to compute the resultant force applied to each particle by other particles in contact with it
3. **Velocity and coordinate update:** in order to recalculate the particles' new velocities and coordinates.

Issues governing the number of arithmetic operations and amount of computer time required for each of these stages are discussed in detail in the following sub-sections.

2.6.1 Contact Checking Analysis

The identification of which particles are in contact with each is the most time consuming operation of the three stages. It requires that each possible pairing of balls be examined, which for N particles requires $O(N^2)$ operations. Thus, for large problem sizes, contact identification dominates the complexity of the problem. Dividing the domain up into cells (see Figure 2–4) using the domain decomposition method can alleviate this. Each particle is tagged as belonging to a particular cell, and it will only be checked for contacts with



(a) (b)

Figure 2–4 *Screenshots of two syste . (a)Without grid (b) with grid*

particles within the same cell and adjacent cells. If the number of particles per cell is c, then the execution time is proportional to $\frac{N}{c}O(c^2)$.

Occasionally a particle may transition from one cell to another, or may straddle the boundary between two cells. A new sub-step has to be included in the data flow of the DEM as *reboxing* of the particles is now necessary whenever a particle moves to an adjacent box (see Figure 2–5).

In order to illustrate the impact of the domain decomposition, a set of experiments was performed using the software described in the next chapter, in order to determine the CPU time spent by the software simulator on each of the steps of the DEM method. An example domain with 500 particles was generated and ran once with domain decomposition and once without. Table 2–1 shows the difference between the two cases. As can be seen, with the new *reboxing* step introduced in the data flow for the case where the domain is decomposed, the contact detection time falls dramatically and the total time needed to perform the contact detection plus reboxing is much smaller than the time needed to produce the contact detection without grid.

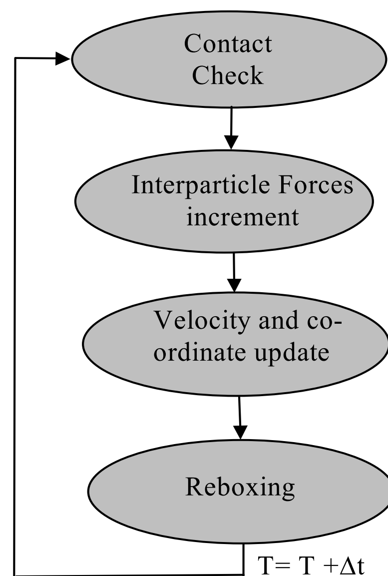


Figure 2–5 *Modified DEM flow chart with domain decomposition*

Table 2–1 *Timing analysis for a system with 500 particles ran for 1000 time steps with and without a grid*

Time	Forces Update	Coordinates Update	Contact check	Rebox
t [No grid]	1.64 s	0.313 s	70.39 s	0
t [Grid]	1.64 s	0.313 s	2.23 s	0.13 s

In addition to analysing the timing characteristics of the contact check stage, it is also necessary to analyse the arithmetic complexity of this stage. The number of arithmetic operations needed to compute Eq. 2–1 is given below in Table 2–2.

Table 2–2 *Arithmetic operations needed for the contact check*

	Additions and Subtractions	Multiplications	Square Roots
Number of Arithmetic operations	5	2	1

2.6.2 Forces Update Analysis

For every contact identified between two particles, the contact force has to be calculated. This is assumed to be linearly proportional to the indentation between the balls. The resultant force on a particle is the vector sum of the forces caused by each contact with its neighbours.

The model used throughout this thesis considers a granular medium with all of its particles having identical radius R. This greatly simplifies the hardware implementation of the algorithm, because it means that for a 2-D implementation, a particle can have a maximum of six other particles in contact with itself (see Figure 2–6).

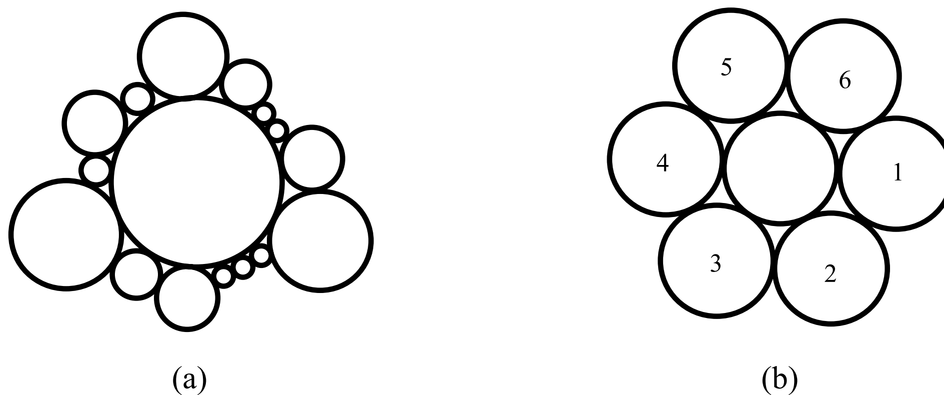


Figure 2–6 *Two cases of different number of balls in contact (a) Balls in contact with different radii (b) Balls in contact with the same radius*

If the particles had varying radii (see Figure 2–6a) there could be in principle be an unlimited number of balls in contact (if $R_{largest} >> R_{average}$ and $R_{smallest} << R_{average}$, where $R_{largest}$ is the larges radius in the system, $R_{smallest}$ is the smallest radius in the system and $R_{average}$ is the radius of the average sized particles). This would necessitate the use of complicated data structures, such as linked lists, which are difficult to handle efficiently in hardware.

That a maximum of six discs can be in contact with a certain disc can be derived mathematically in a very simple manner (see Figure 2–7 and Eq. 2–17 and Eq. 2–18).

$$\alpha = \sin\left(\frac{R}{2 \times R}\right)^{-1} = 30^{\circ} \qquad \textbf{\textit{Eq. 2–17}}$$

$$2 \times \alpha = 60 \Rightarrow \frac{360^{\circ}}{60^{\circ}} = 6 \qquad \textbf{\textit{Eq. 2–18}}$$

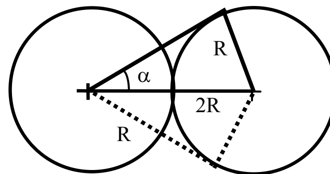


Figure 2–7 *Geometrical deduction of the maximum number of balls in contact for balls of the same radius*

Having all particles of the same radius R means that adjacency information can be represented by a very simple data structure (a 6xN matrix) and that the maximum number of interparticle forces computation required is *6 N*, for the worst case in which all particles have the maximum number of particles in contact.

After determining which particles are in contact, the forces and moments between the particles is calculated. The pseudo code for the calculation of the forces and moments is given below. As mentioned earlier, the forces between the two balls are assumed to be directly proportional to the amount of indentation between the two balls [7].

FOR Ball 1:last

WHILE Ball has balls in contact

$x_{dif} = x_1 - x_2$ (x-coordinate difference)

$y_{dif} = y_1 - y_2$ (y-coordinate difference)

$D = \sqrt{x_{dif}{}^2 + y_{dif}{}^2}$ (distance between particle centroids)

$\sin(\alpha) = \frac{y_{dif}}{D}$

$\cos(\alpha) = \frac{x_{dif}}{D}$

$v_{xdif} = v_{x1} - v_{x2}$ (x-direction relative velocity)

$v_{ydif} = v_{y1} - v_{y2}$ (y-direction relative velocity)

$D_N = [(v_{xdif} \times \cos(\alpha)) + (v_{ydif} \times \sin(\alpha))] \times \Delta t$

$D_S = [(v_{xdif} \times \sin(\alpha)) + (v_{ydif} \times \cos(\alpha)) - (\theta_{s1} \times R_1) - (\theta_{s2} \times R_2)] \times \Delta t$

$D_{FN} = D_N \times {k_N}/{2}$ (Incremental normal force)

$D_{FS} = D_S \times {k_S}/{2}$ (Incremental shear force)

$F_N = F_{Ncontact} + D_{FN}$ (New total normal force in contact)

$F_{NT} = F_N + (D_{FN} \times BDT)$ (BDT is the damping contribution)

$F_S = F_{Scontact} + D_{FS}$ (New total shear force in contact)

(Restoring of normal and shear force in x and y directions)

$F_x = F_{NT} \times \cos(\alpha) + F_{ST} \times \sin(\alpha)$ (Restore normal force to x)

$F_y = F_{NT} \times \sin(\alpha) - F_{ST} \times \cos(\alpha)$ (Restore shear force to y)

$M = F_{ST} \times R$ (Compute moment)

$F_{xnew1} = F_{xold1} - F_x$ (Add force increment in x direction)

$F_{ynew1} = F_{yold1} - F_y$ (Add force increment in y direction)

$M_{new1} = M_{old1} - M$ (Add moment increment)

$F_{xnew2} = F_{xold2} - F_x$ (Same for 2nd particle)

$$F_{ynew2} = F_{yold2} - F_y$$

$$M_{new2} = M_{old2} - M$$

```
    Next Ball in contact
  Next Ball
```

Where Δt is the time step and k_n and k_s are the normal and shear stiffness of the system. Table 2–3 gives an overview of the total number of arithmetic operations needed to compute the force between two balls in contact.

Table 2–3 *Arithmetic operations of the force update function*

	Additions and Subtractions	Multiplications	Divisions	Square Roots
Number of Arithmetic ops	20	18	2	1

In addition to the equations given above, the maximum shear force before the particles start sliding is computed (F_{smax}). In the case that the shear force (F_s) is larger than this maximum shear force (F_{smax}), the particle will slide instead of rotating with the particle in contact, and the shear force will be set equal to the absolute value of F_{smax} preserving the sign of F_s.

2.6.3 Coordinate Update Calculation Analysis

Movement update entails the solution of Newton's second law for each of the N particles. This requires O(*N*) operations. This is the fastest of the three stages that are performed for each time step. The equations that describe how the particles' new velocities (v_{xnew}, v_{ynew}, and θ'_{snew}) and coordinates (x_{new}, y_{new} and θ_{new}) are calculated are given below:

```
  FOR Ball 1 until last
```

$$V_{xnew} = \left\{ \left(V_x \times Con1 \right) + \left[\left(\frac{F_x}{mass} + g_x \right) \times \Delta t \right] \right\} \times Con2$$

$$V_{ynew} = \left\{ \left(V_y \times Con1 \right) + \left[\left(\frac{F_y}{mass} + g_y \right) \times \Delta t \right] \right\} \times Con2$$

$$\dot{\theta}_{snew} = \left\{ \left(\dot{\theta}_s \times Con1 \right) + \left(\frac{M \times \Delta t}{I} \right) \right\} \times Con2$$

$$x_{new} = x + \left(v_x \times \Delta t \right)$$

$$y_{new} = y + \left(v_y \times \Delta t \right)$$

$$\theta_{new} = \theta + \left(\dot{\theta}_s \times \Delta t \right)$$

```
Next Ball
```

Where Con1 and Con2 are constants dependant on the damping of the system, based on the Rayleigh damping constants α and β which can be given in terms of minimum damping ratio λ and the frequency, f, at which the damping ratio is at the minimum, as shown in Eq. 2–19, Eq. 2–20, Eq. 2–21 and. Eq. 2–22.

$$\beta = \frac{\lambda}{2 \pi f}$$ ***Eq. 2–19***

$$\alpha = 2 \pi \lambda f$$ ***Eq. 2–20***

$$Con1 = 1 - \frac{\alpha \Delta t}{2}$$ ***Eq. 2–21***

$$Con2 = \frac{1}{1 + \frac{\alpha \Delta t}{2}}$$ ***Eq. 2–22***

M is the mass of the particles, which is constant for particles of the same radius and density, Δt is the time step, I is the moment of inertia and g is the acceleration due to gravity (in the x and y direction). The number of arithmetic operations involved is given in Table 2–4.

Table 2–4 *Arithmetic operations of the coordinate and velocity update function*

	Additions and Subtraction	Multiplications	Divisions
Number of ops	8	12	3

2.7 Parallelism Analysis

The previous sections described the DEM in terms of the computational complexity of the tasks and the arithmetic operations they involve. In order to successfully accelerate the DEM with custom hardware, the algorithm needs to be checked for parallelism. If there is no parallelism at all, it is almost impossible to get any speed-up using custom hardware as state-of-the-art serial processors can execute serial code at an enormous speed.

2.7.1 Basic Ideas about Parallelism

Programs that run on serial machines make little or no use of the parallelism of the algorithm they are running. A dedicated hardware architecture can exploit the full extent the parallelism of the algorithm.

Traditionally a measure of speed-up has been used to judge the quality of parallel algorithms running on multiprocessors systems. In the case of designing a dedicated hardware architecture for the DEM the term speed-up can be defined as the time needed to run certain simulations by an optimised software implementation on a single processor machine compared to the time needed by the dedicated hardware design for the same simulation.

2.7.2 DEM Parallelism

The parallelism involved in the Discrete Element Method can be described by a hierarchy of computational structures. The lowest level processes involve the arithmetic operations, while the highest level corresponds to the major tasks namely contact checking, force calculation and position update.

2.7.2.1 Low Level– Fine Grain Parallelism

The low level parallelism is concerned with the concurrent execution of arithmetic operations. Two operations are concurrent if their execution times overlap. One of the easiest ways to represent the parallelism of an algorithm is with the help of graphs [11].

Eq. 2–23 and its graphical representation in Figure 2–8 shows an example of an algorithm in its sequential and a parallel from.

$$z = a + b + c + d \qquad \textbf{Eq. 2–23}$$

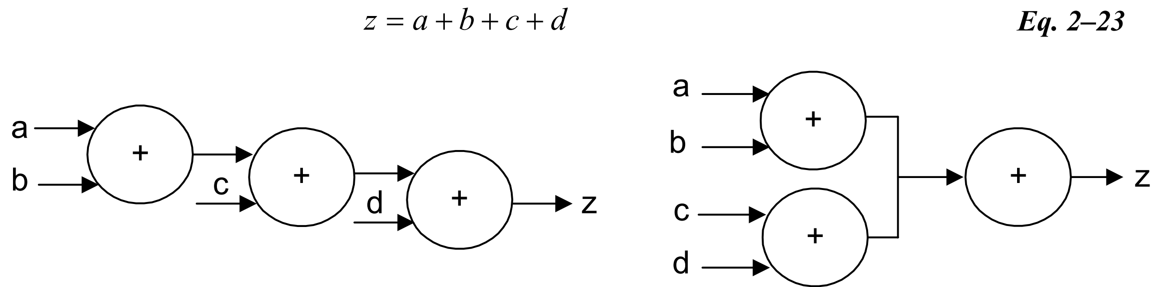


Figure 2–8 *Example of a serial (left) and parallel (right) graphical representation of an algorithm*

This graphical method has to be applied to the equations given in sections 2.6.2 and 2.6.3 in order to have a visual representation of the low level parallelism of the DEM. As can be seen from Figure 2–9 there are two main paths in the forces update equations: one corresponding to the calculations of the normal forces applied to the two balls in contact and a second to compute the shear force between the two balls.

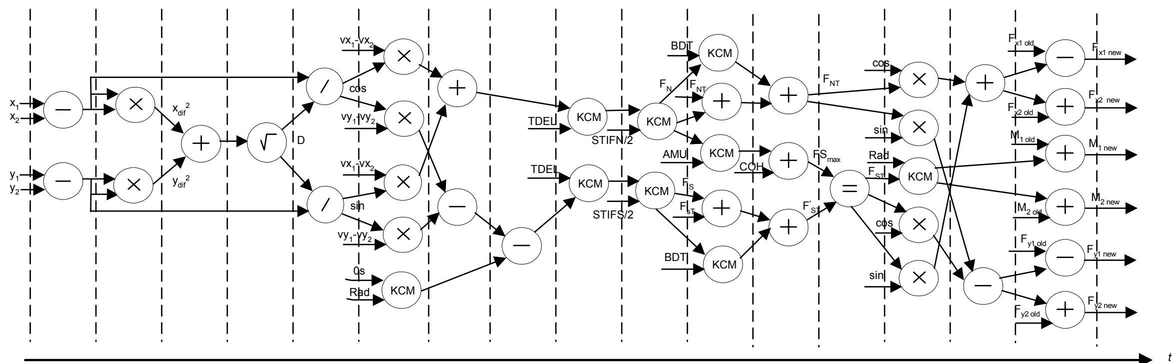


Figure 2–9 *Graphical representation of the force update equations*

In the position update equations there are three independent paths as seen in Figure 2–10. One is for the computation of the velocity and position in the x direction; the second to compute the velocity and new coordinate in the y direction and a third path to compute the angular velocity of the particle and the angular rotation.

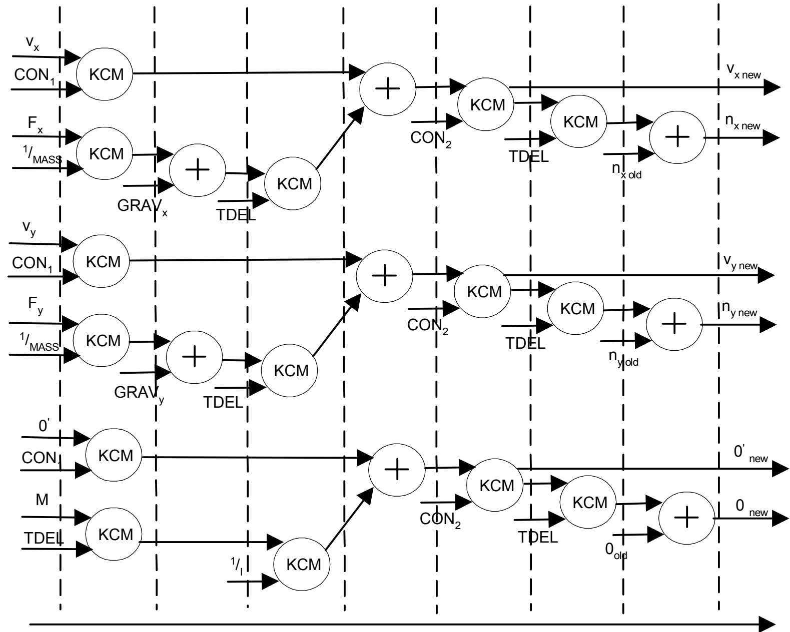


Figure 2–10 *Graphical representation of the position update equations*

The longest path in these two graphs gives the critical path (CP). These critical paths will determine the minimum time needed to compute the algorithm if these are computed concurrently.

2.7.2.2 High Level – Coarse Grain Parallelism

In contrast to the low level parallelism, the DEM also shows some good possibilities for high level parallelism, which is concerned with the larger tasks. If no domain decomposition is used, then the tasks are contact detection, forces calculation and position update. If domain decomposition is used, reboxing must also be carried out. The key question is whether there is a way to compute all these three (or four) tasks simultaneously. The only way to make this feasible is by decomposing the domain into boxes or columns. This allows three possibilities:

1. By dividing the domain into cells the problem can be split among processors i.e. each processor handling part of the domain. This strategy is commonly used in symmetric multiprocessor parallelization of the DEM (see Figure 2–11 (a)).

2. All three (four) tasks are performed at the same time. This will only work if the order in which the tasks have to be performed is followed (contact checking in one cells is performed before the forces update and before position update and in turn the forces update unit in one cell has to be performed in one cell before the position update unit can be performed on that same cell, see Figure 2–11(b)).

3. Combining both methods, dividing the domain into sub-domains and assigning different processors to each sub-domain and performing the DEM tasks in each sub-domain in parallel in each processor.

Figure 2–11 shows two examples of the use of the domain decomposition to use the high level parallelism of the DEM. In Figure 2–11(a), the domain is partitioned into equal sizes in order to split the information between several processors, performing the same task

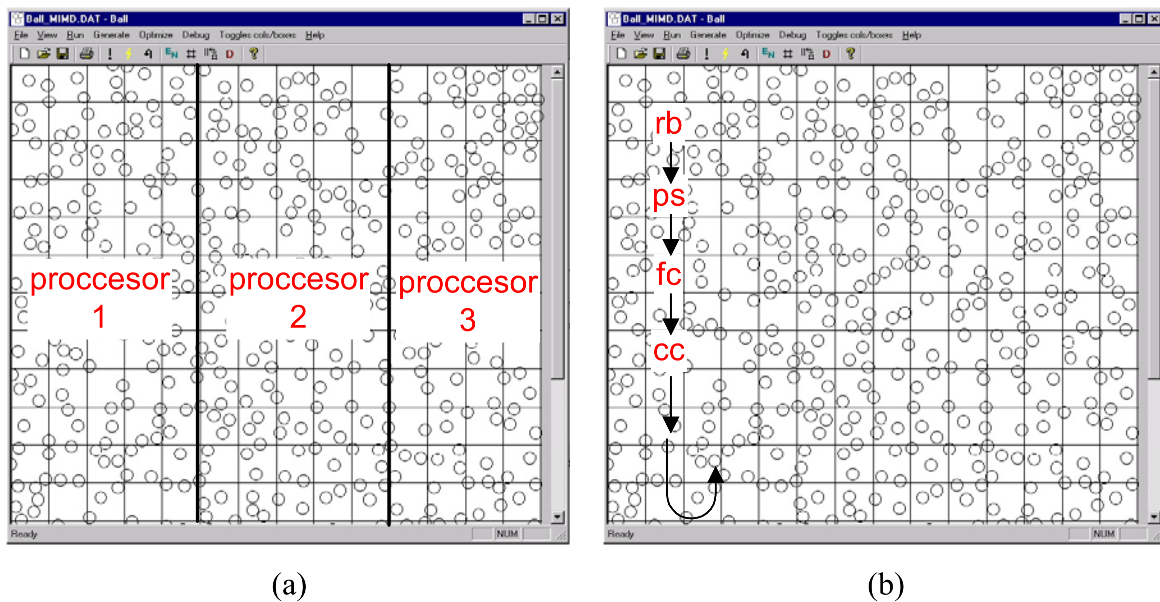


Figure 2–11 *Examples of domain decomposition in order to make use of the high level parallelism of the DEM. Division among multiple processors (a) and concurrent computation of the DEM steps (b)*

concurrently but on different parts of the domain. For the example in Figure 2–11(b), all three different tasks are computed in a pipelined manner, i.e. all operate at the same time,

but on a different part of the domain. In this figure cc stands for contact checking, fc for forces update, ps for position updating and rb for reboxing.

2.7.3 Discussion of the Application of the Low and High Level Parallelism of the DEM on FPGAs

The low level parallelism of the DEM can be fully exploited in an FPGA since as many arithmetic operations as are needed can be instantiated in parallel, so far as the logic resources allow it.

The high level parallelism is more a factor of scheduling than of arithmetic. As the FPGA has a large amount of logic resources with embedded memory, tasks can be scheduled in any way, as the control unit, which generates the control signals and steers data across the device, will also be customized.

Designing a dedicated hardware architecture to perform a specific task is extremely time consuming. The introduction of Intellectual Property (IP) Cores, which are pre-designed units that perform a specific task in the design, has alleviated this, allowing the implementation of hardware designs much faster than before. FPGAs benefit from these, allowing designs to be implemented faster and more reliably, as these IP cores have already been validated.

2.8 Summary and Conclusions

This chapter has demonstrated the importance of the study of the behaviour of granular materials, since they have an important role in many different disciplines e.g. civil, mechanical and chemical engineering. New approaches have also considered entire engineering structures built from bonded particles, and some authors predict that these methods could replace the continuum approaches used today, such as the finite element method.

The DEM is a computationally very expensive algorithm, because:

- The behaviour of every single particle is computed separately for every time step
- The time step has to be very small so that disturbances cannot travel beyond neighbouring balls in every step.

An in-depth study of the algorithm has been made studying the parallelism involved in every task and the associated arithmetic operations. A summary of the properties of the DEM steps is shown in Table 2–5.

Table 2–5 *Summaries of the properties of the main steps in the DEM (assuming N is reasonably large)*

	Operations/time step	Execution Time	Arithmetic ops
Contact checking	$O(N^2)$	Slowest	8
Forces updated	$O(6\ N)$	Intermediate	41
Positions update	$O(N)$	Fast	21
Re-boxing	$O(N_{cells} x[4xcell_{length}/diameter])$	Intermediate	Only data management

The contact checking task is the one that requires the smallest number of arithmetic operations, but it is the one that needs to be performed most often ($O(N^2)$)(if no domain decomposition is used).

The forces update step is the computationally most expensive task. It needs 41 arithmetic operations, but needs only a maximum of $O(6\ N)$ operations to be performed which is intermediate between the contact checking and the position updating task.

The position update task is the one that needs the least operations to be performed, i.e. the fastest to be computed, and is the one that needs an intermediate number of arithmetic operations, but the potential of parallelism of this task is very large as shown in Figure 2–10. This is because there are three major independent paths in the computation the velocities and positions of the balls.

Re-boxing particles transitioning from one cell to another does not require any arithmetic operations, and is purely a data management issue. The number of particles transitioning from one cell to another is heavily dependant on the cell size and cell occupancy. The smaller the cells are, the more likely a particle is to pass from one cell to another. If the cell is to be assumed to be full with balls, then the estimated number of balls that will transition to the neighbouring boxes (and therefore the number of re-boxing operations needed per cell) is: $O(4xcell_{length}/diameter)$.

The enormous amount of parallelism in the DEM suggests that a dedicated hardware architecture could bring significant benefits in terms of speed-ups to the calculation.

2.9 References

[1] Cundall, P.A. *"A Computer model for simulation of progressive, large scale movements in blocky rock system"*, Proc. Symp. Int. Rock Mechanics, Nancy 2, No. 8. 1971.

[2] Bardenhagen S.G, Brackbill J.U, Sulsky D, "*The material-point method for granular materials"*. Computer methods in applied mechanics and engineering 187, 529-541, 2000.

[3] Heinrich M. Jaeger, Sidney R. Nagel, Robert P. Behringer, Physics Today, April 1996. "*An introduction to granular physics*", *page 32.* Available at http://arnold.uchicago.edu:80/~jaeger/granular2/introduction.

[4] Oda M, Iwashita, K, *"Mechanics of Granular Materials"*, A.A. Balkema, Rotterdam 1999.

[5] Cundall, P.A. Strack, O.D.L., "*A discrete numerical model for granular assemblies*", Geotechnique 29, pp. 1-8, 1979.

[6] Cundall, P.A.*,"A discontinuous future for numerical modelling in geomechanics?"* Proceedings of the Institution of Civil Engineering, Geotechnical Engineering 149, January 2001, Issue 1 Pages 41-47.

[7] Cundall, P.A. Strack, O.D.L., *"The Distinct Element Method as a tool for research in Granular Media",* Report to the National Science Foundation Concerning NSF Grant ENG76-20711, Appendix 2, pp 20-21, University of Minnesota, November 1978.

[8] Holst JMFG, Rotter JM, Ooi JY, Rong GH, "*Numerical modelling of silo filling II: discrete element analysis*". J Engng Mech ASCE; 125(1): 104-110, 1999.

[9] Munjiza A, Owen DRJ, Bicanic N. "*A combined finite discrete element method in transient dynamics of fracturing solids"*, Engng Comput 1995: 12(2): 145-74.

[10] Hertz, H., "*Ueber die Beruehrungfester Elastischer Koerper,"* J. Renie Angew Math., 92, pp 156-171, 1992

[11] Wanhammer, L. *"DSP Integrated Circuits"*, Academic Press, San Diego, chapter 6, San Diego, 1999.

Chapter 3

SOFTWARE IMPLEMENTATION OF THE DEM

3.1 Introduction

This chapter describes and analyses the software implementation of a DEM simulator. A software simulator for the DEM was written in order to understand the processes and operations behind the DEM better, to allow the functional verification and validation of the hardware implementation, and to allow a run time comparison between the sequential program and the parallel one implemented on an FPGA, since the ultimate goal of this work is to achieve a faster running system.

In order make a fair comparison between the hardware and the software implementation, the software was optimised as much as possible.

The DEM software simulator developed for this study is based on the one written by Cundall and Strack [1] in 1978 called Ball, which was written in FORTRAN. The software is written in C and has the following enhancements and improvements over the Ball code:

- It uses the superior data structure facilities of the C language, together with a number of other optimisations, to give better performance

- It has a Visual C++ wrapper which provides a Graphical User Interface (GUI) that assists in visualisation of the DEM simulations
- It has facilities for a variety of domain decomposition approaches
- It can communicate with the reconfigurable computing platform, so the numerically intensive portions of the code can run in hardware if the user wishes
- It has facilities for comparing the results produced by the hardware with the corresponding results produced by software (both for the bulk and for the individual particles)
- It has facilities for emulating the partition of the problem across multi-processor platforms. The simulator produces data about inter-processor communication and synchronisation overheads, which can be used to determine what degree of speed-up, could be achieved on various platforms, and what are the factors that limit the speed-up.

The simulator consists of 5000 lines of code and is capable of modelling the behaviour of assemblies of particles under conditions of loading and unloading. It is simpler than the Cundall and Strack simulator in that it requires all particles to have the same radius, and does not allow the domain to contain walls. These simplifications were made in order to reduce the complexity of the DEM algorithm sufficiently to be able to migrate the key stages of the algorithm into FPGA hardware.

The procedure used by the simulator is illustrated in Figure 3–1. It first reads a data file (an example is shown in Figure 3–2), which provides information about the system that is to be simulated. It then generates the number of particles requested by the user, within the domain area requested by the user.

The program then commences the simulation, stepping through the three principal tasks of the DEM: contact checking, forces update and positions update, as well as the reboxing task that re-allocates particles when they transition from one box of the grid to another. The grid is necessary in order to alleviate the time needed to perform the contact checks as explained in chapter 2. After every cycle, the simulator checks if the number of steps equals the number of cycles given in the initialisation file. If so the simulation ends and a log file is generated. The program writes the final position of the particles, the resultant forces between them, the total run time of the simulation, the total system energy (total kinetic energy and total potential energy), as well as the initial conditions of the system in the report. The energy calculations are used to provide an assessment of the numerical stability of the algorithm, as explained in chapter 6.

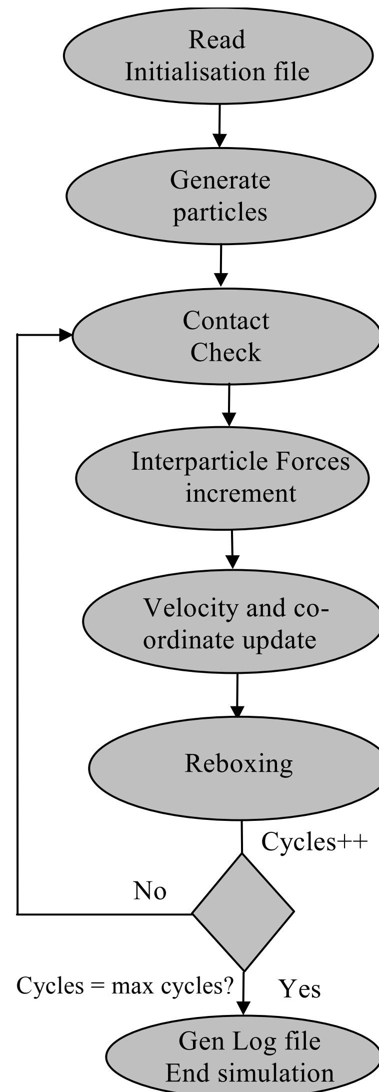


Figure 3–1 *DEM simulator flow graph*

3.2 Initialisation of the Simulation

The first thing the simulator does is to read the initialisation file. The program data are all stored in a file called **BALL.DAT.** The format of all input commands is a word followed, in most cases, by a number of parameters. Any input line starting with a semicolon (;) is regarded as a comment. The format of a typical input file is given below:

```
START 4000.00 4000.00 200 1
RADIUS 45.00
AUTO 0.00 4000.00 0.00 4000.00 500 1000 0 1
SHEARSTIFF 20.00
NORMSTIFF 400000.00
DENSITY 2.00
FRICTION 0.00
DAMPING 0.00 0.00 0.00 0.00
COHESION 4000.00
XGRAVITY 0.00
YGRAVITY 0.00
FRACTION 0.08
CYCLE 1000
```

Figure 3–2 *Example of the simulator initialisation file*

A short description of each command is given below:

START W, H, NBOX, COL_BOXES

The first command **has** to be START, as this defines the area in which the particles will be generated. The parameters are as follows.

W is the width of the domain (x dimension)

H is height of the domain (y dimension)

NBOX is the number of boxes or columns requested to form the grid

COL_BOXES is a flag to show whether the domain is to be divided into columns or boxes. (The hardware implementation only allows the domain to be split into columns, not boxes).

RADIUS R

R is defined as the radius for the particles. All the particles will be of the same radius, because this simplifies the hardware implementation dramatically as will be shown in chapter 5.

AUTO XL, XU, YL, YU, N, NTRY, SEED, INIT_VEL,

The program will try to generate N particles within the area within the rectangle with corner coordinates (XL,YL) and (XU, YU), using a random number generator for the co-ordinates. If the program is not able to produce N particles after NTRY attempts (because it is not allowed to create an initial condition in which particles overlap), it will give up and write a message informing the user how many were actually generated. If NTRY is omitted, or given as zero, it defaults to 1000. SEED is a flag that establishes whether the program should use the same pseudo-random initialisation sequence each time the program runs, or whether it should generate a new sequence. INIT_VEL is a flag that tells the simulator if the particles should be given an initial velocity of zero, or a randomly generated non-zero value.

SHEARSTIFF k_s

k_s is the value of the shear contact stiffness of the particles.

NORMSTIFF k_n

k_n is the value of the normal contact stiffness of the particles.

DENSITY ρ,

ρ is the value of the density of the particles

FRICTION μ

μ is the value of the friction coefficient between the particles

COHESION c

c is the value of the cohesion between the particles.

DAMPING λ_{min}, f_{min}

Sets the damping parameters for Rayleigh damping, where f_{min} is the frequency at which the minimum damping occurs, and λ_{min} is the damping ratio i.e. fraction of the critical damping at that frequency.

XGRAVITY g_x

YGRAVITY g_y

These are the gravitational accelerations in the x and y directions respectively.

CREATE x, y, v_x, v_y

Creates a particle with centre at x, y and initial velocities of v_x, and v_y.

CYCLE n

The program performs n calculation cycles (i.e time steps). No communication with the simulator is possible once it starts.

FRACTION f

Sets the time step used to a fraction f of the critical time step.

Some of the commands must be provided in an appropriate order. For example, no balls can be generated if the area in which the balls are to be generated has not yet been defined, therefore the first command must always be START.

Figure 3–3 shows a screenshot of the simulator's program data window once the initialisation file, given in Figure 3–2, has been read.

It can be noticed from Figure 3–2 that no units are specified in the initialisation file. The input parameters are considered to be given either in S.I. units or in a consistent scaling of these.

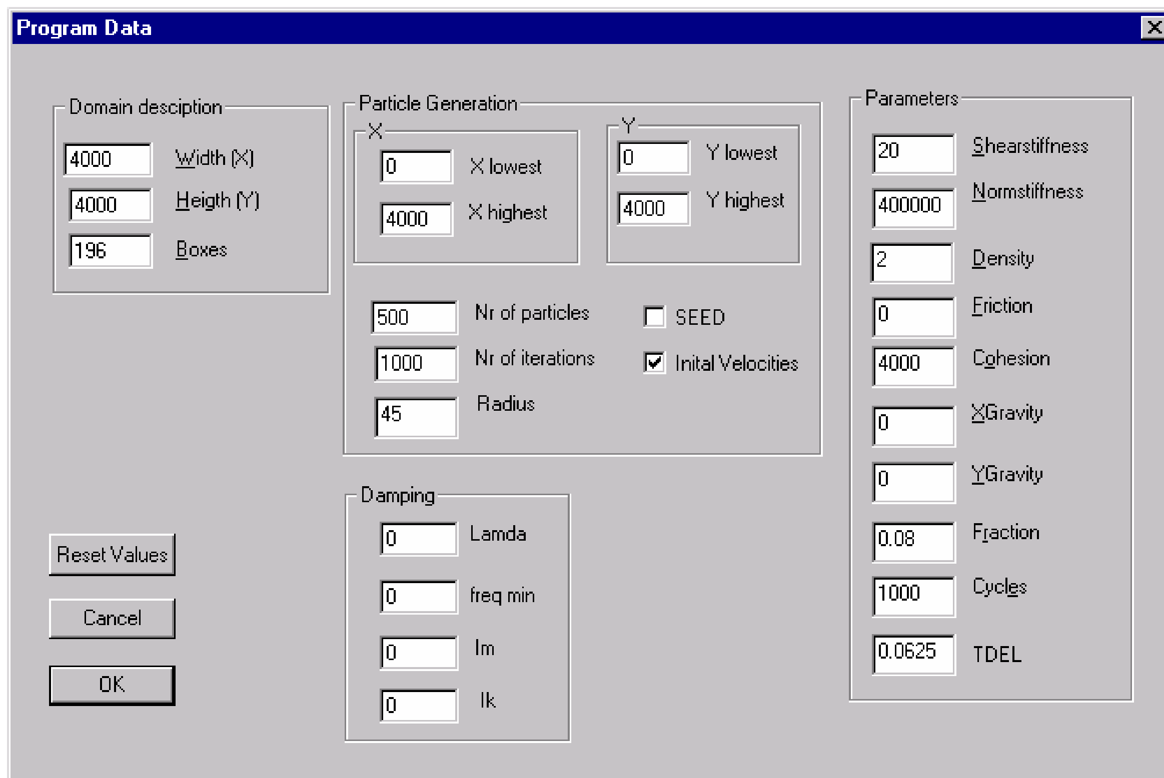


Figure 3–3 *Screenshot of the simulators initial data after the particles have been generated*

Figure 3–4 shows a screenshot of the simulator's initial state, once the initialisation file given in Figure 3–2 has been read and the particles generated. Once the particles are generated, it waits for the user to either make any changes with its parameters or to start the simulation.

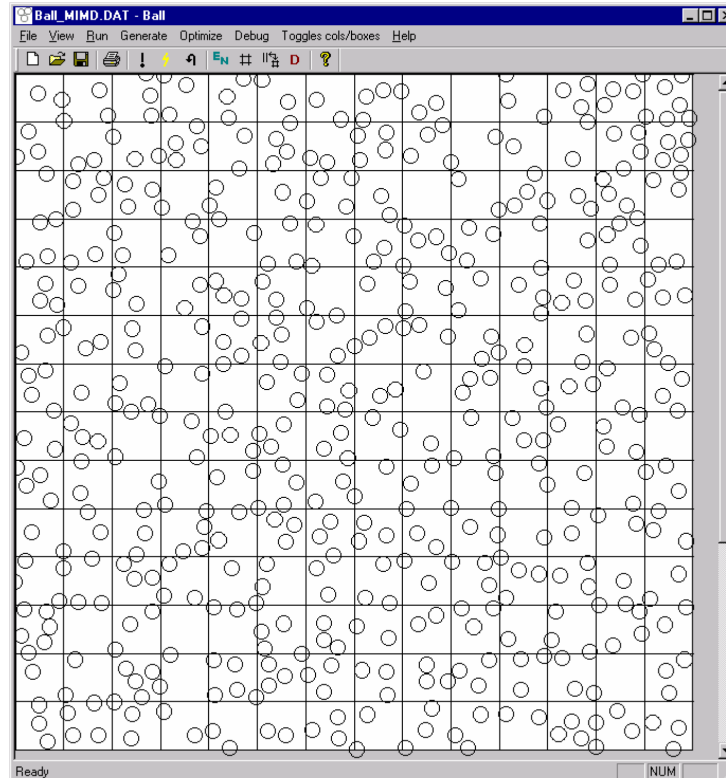


Figure 3–4 *Screen shot of an initial state of the simulator after reading in the data file*

3.2.1 Data Structure

Having such an enormous amount of data, with thousands or even millions of particles, one needs a well-designed data structure in order minimize the searching time required. It was decided to use a linked list data structure. The balls are defined as an object and they are linked to each other using a complex singly linked list.

```
class CBALL: public CBall_data{
public:
        float x;
        float y;
        float xs;
        float ys;
        float Os;
        float Fx;
```

```
float Fy;
float M;
float O;
float FTN[6];
float FTS[6];
class CBALL *next_ball;
class CBALL *same_entry;
class CBALL *ball_contact[6];
class CBALL *same_debug;
};
```

Figure 3–5 *Data structure of the particles*

As seen in Figure 3–5every ball structure contains its intrinsic data, plus four pointers to other balls.

***Next ball:** Points to the next ball generated by the simulator after the initialisation file is read.

*__Same entry:__ Points to the next ball in the same box as the current ball.

***Ball_contact [6]:** Points to the balls that are in contact with this one. For the case of a 2D system with balls of the same radius, there can be a maximum of 6 balls in contact, as shown in section 2.3.2.

***Same_debug:** Used only for debugging purposes when the software system is being compared to the equivalent hardware system in order to check for similarities in their behaviours.

Figure 3–6 gives a graphical representation of the linked list structure described above. Ball 1, 2 and 3 build the linked list of particles in the order that the simulator has generated them. Ball 1 points to Ball M, which is a particle in the same box as Ball 1 and in turn Ball M points to Ball N, which is also in the same box as Ball M and Ball 1. The last pointer entry for Ball 1 also points to Ball X, which is a Ball with which Ball 1 is in contact.

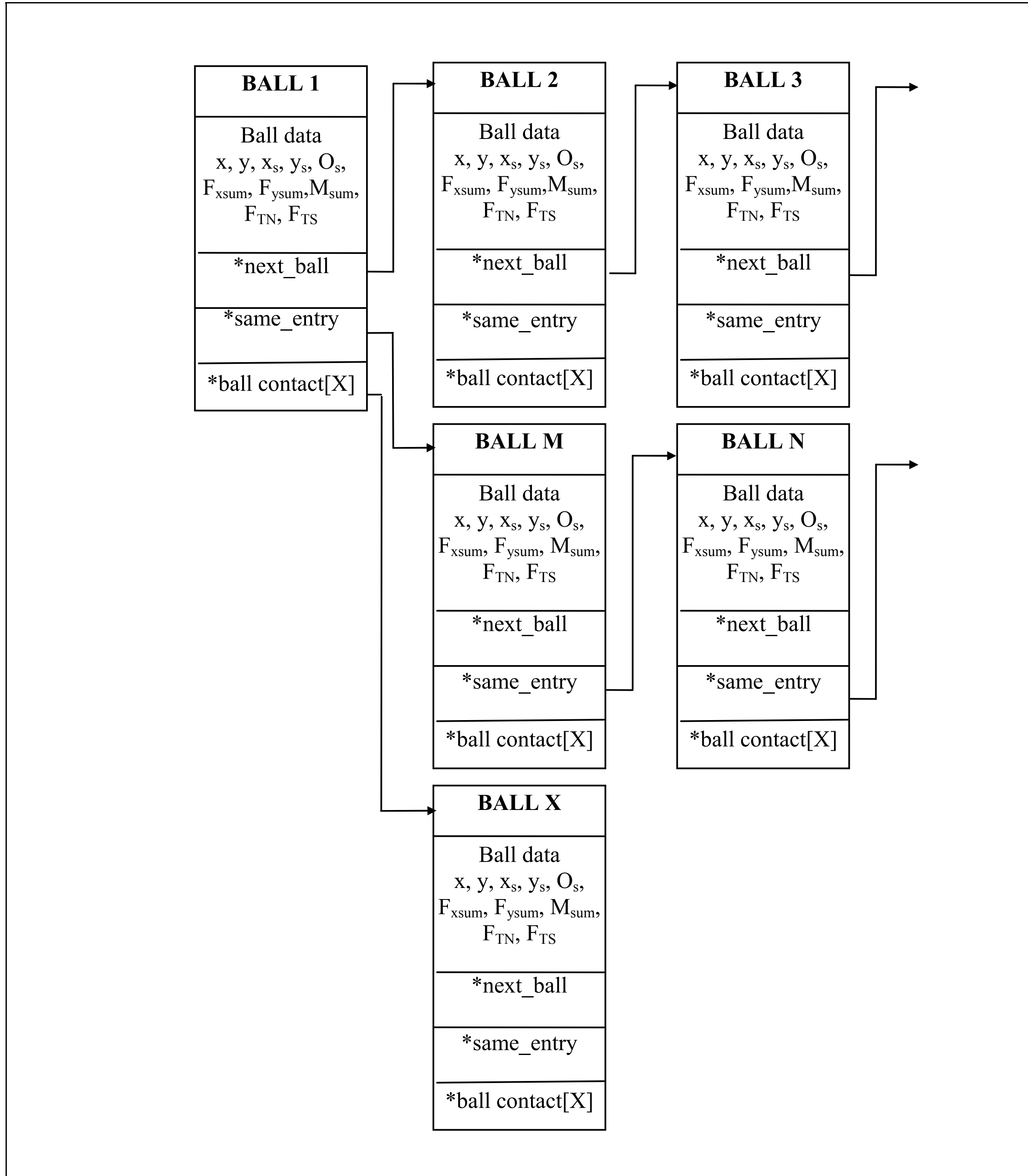


Figure 3–6 *Data linked list structure*

3.3 Simulator Features

This section will give a brief overview of the features of the simulator. The main features are an in-built grid optimiser, which generates the initial grid for the system depending on the number of particles, their size and the domain size. Another important feature is the energy monitoring option. This option opens a window which shows the current total energy of the system, the current total kinetic energy and the current total potential energy during each time step.

Figure 3–7 shows the tool bar of the simulator with the different options available.

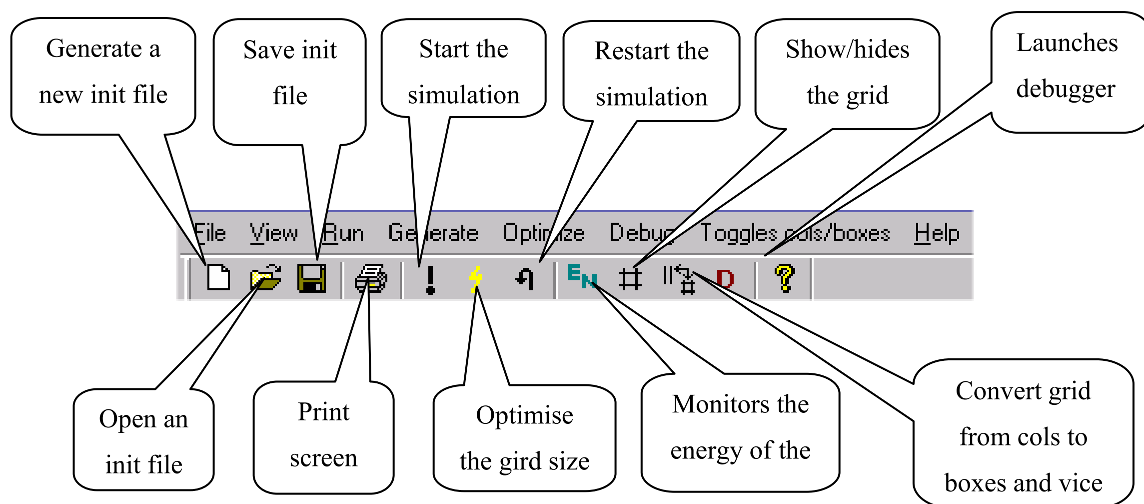


Figure 3–7 *Simulator's tool bar*

The grid converter, from columns to boxes and vice versa, is needed in order to compare the hardware system with the software one, as the hardware system only allows the domain to be split into columns and not into boxes.

The Debugger button in Figure 3–7 launches an in-built debugger which prints information about the particles in the software simulator and those of the hardware system on the same screen, monitors how they behave concurrently and reports any deviations.

3.3.1 Optimal Runtime Grid Size

As mentioned in the previous section, the software simulator has an in-built grid optimiser, which selects the optimum grid size depending on the system parameters i.e. balls' radius, domain size and number of particles The grid size has a dramatic influence on the runtime of the simulation as the time to perform the contact check varies as the square of the number of particles in each box. Figure 3–8 shows how the simulation time changes as a

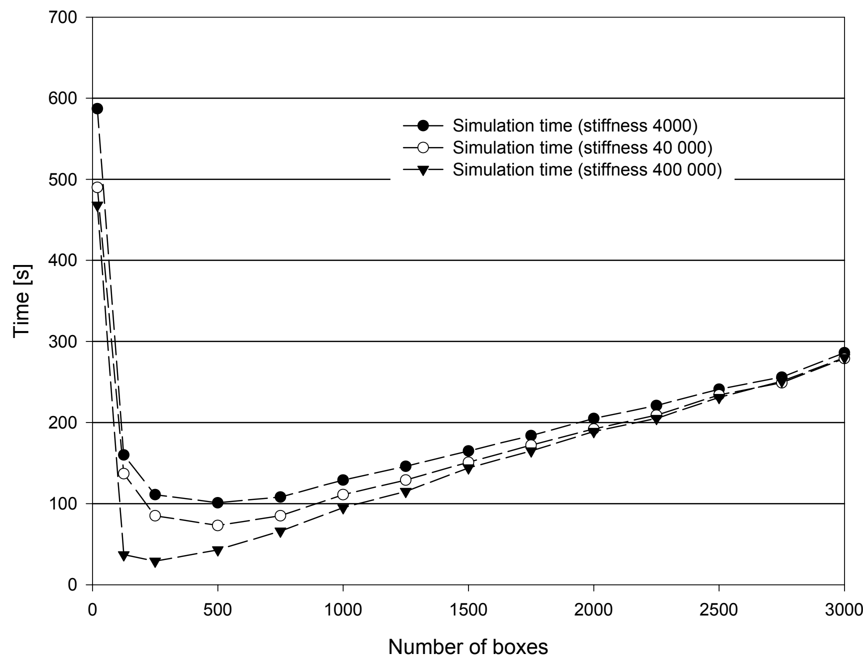


Figure 3–8 *Run time graphed as a function of the number of boxes*

function of the number of boxes used to build the grid for a domain of 2500 particles. A number of curves are plotted, corresponding to different values of the system's stiffness. The simulation time for the system without a grid (see Figure 3–9 (a)) is extremely high. This is because the contact check task grows with the square of the number of particles per box. By making the grid finer, the computing time for the same system falls sharply until a minimum is reached; for this system, it is approximately 500 boxes. From this point onwards the computing time starts to grow again linearly with the number of boxes in the system. This is due to the fact that the time needed to compute the contact check is not dominant anymore. Instead the time needed to rebox the particles transitioning from one

box to the other starts to become dominant as the grid is very fine and there are many particle transitions from one box to another during the simulation.

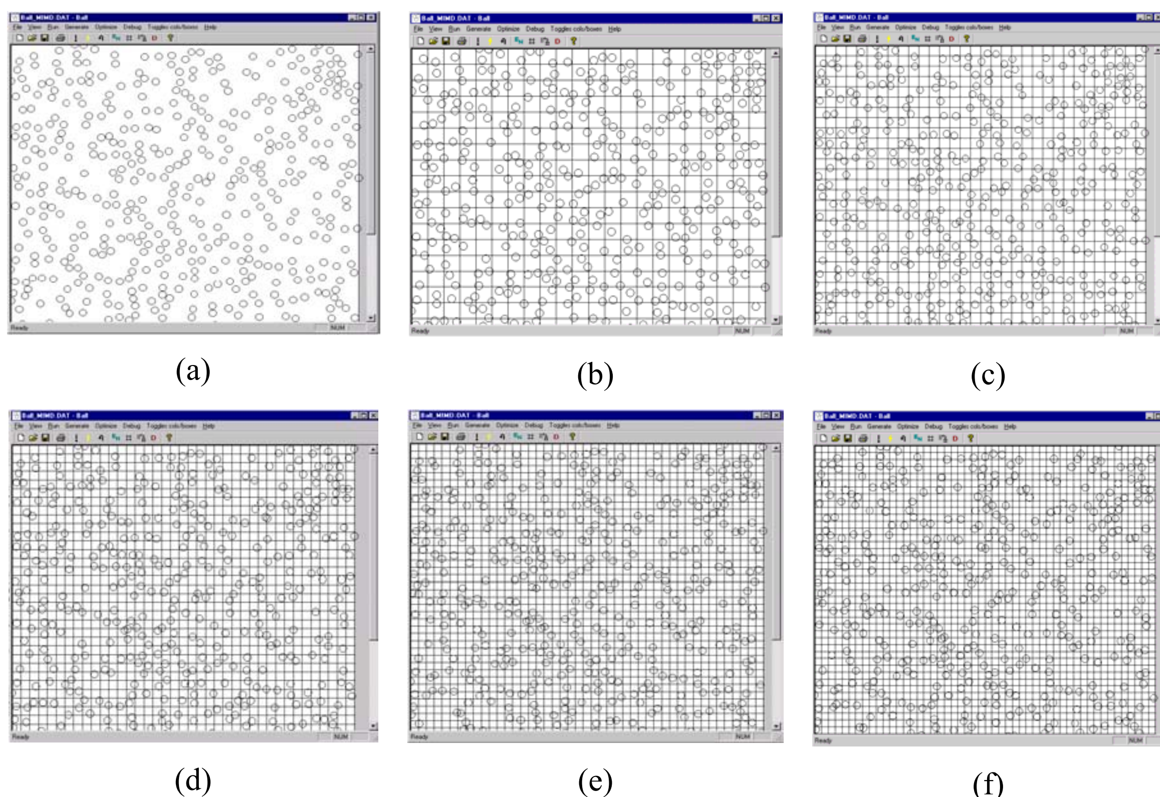


Figure 3–9 *Screenshots of the simulator with different grid sizes*

(a) No grid (b) 500 boxes(c) 1000 boxes (d) 1500 boxes (e) 2000 boxes (f) 2500 boxes

From these results it was established that the ideal grid size is between 4 and 6 times the particle radius, and its exact value depends on the system's stiffness.

The graphs in Figure 3–8 show that the higher the stiffness the lower the runtime of the simulation is (for an optimised grid size). This is because a high stiffness means that the particles will be in contact for less time than for a system with low stiffness. Thus for high stiffness systems the interparticle forces are computed less often than for a low stiffness system. An example of a high stiffness system is an assembly of billiard balls, whereas an example of a low stiffness system is an assembly of squash balls.

3.3.2 Re-draw Option

For some purposes it is important for the user to be able to visualize how the particles move, whereas for other purposes only the final state is of interest to the user. A facility was provided to allow the user to choose how often the system should be re-drawn, since re-drawing the complete assembly can be more time consuming than performing the computations. Figure 3–10 shows a screen shot of this window.

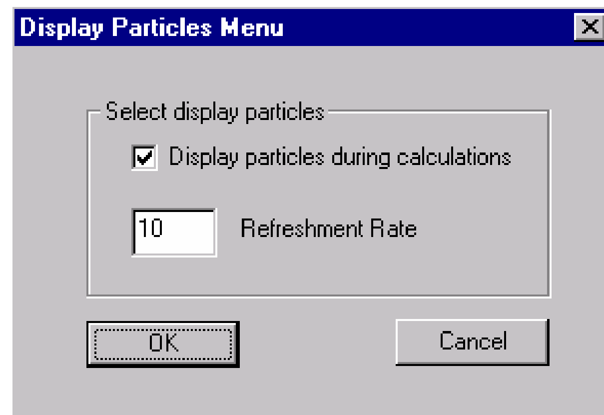


Figure 3–10 *Re-draw option window*

In order to illustrate the effect of re-draw frequency on simulation time, a simulation was set up with 500 particles and run for 1000 time steps. The elapsed time required to perform the simulation without redraw, with redraw every 10 cycles, and with re-draw after every cycle was monitored and is shown in Table 3–1.

Table 3–1 *Comparison of the simulation time needed for an assembly of 500 particles ran for 1000 cycles depending on the number of times the assembly is re-drawn.*

	Re-draw after every cycle	Re-draw after every 10 cycles	Re-draw only at the end of the simulation
Time to run 1000 cycles	37 s	26 s	18 s

3.4 Runtime comparison between the Fortran and the C simulator

Several simulations were performed in order to verify that the optimisations of the C simulator work. Table 3–2 show the time needed by the orignial FORTRAN and the C simulator for assemblies of particles ranging from 1000 to 5000 particles.

Table 3–2 *Runtime comparison between the Fortran and the C simulator*

NUMBER OF PARTICLES	C SIMULATOR (SEC)	FORTRAN SIMULATOR (SEC)
1000	4.00	10.00
2000	12.00	22.00
3000	23.00	33.00
4000	39.00	59.00
5000	57.00	80.00

Figure 3–11 plots the results given in Table 3–2. It can be observed that the C simulation is approximately 1.5 to 2 times faster than the FORTRAN simulation.

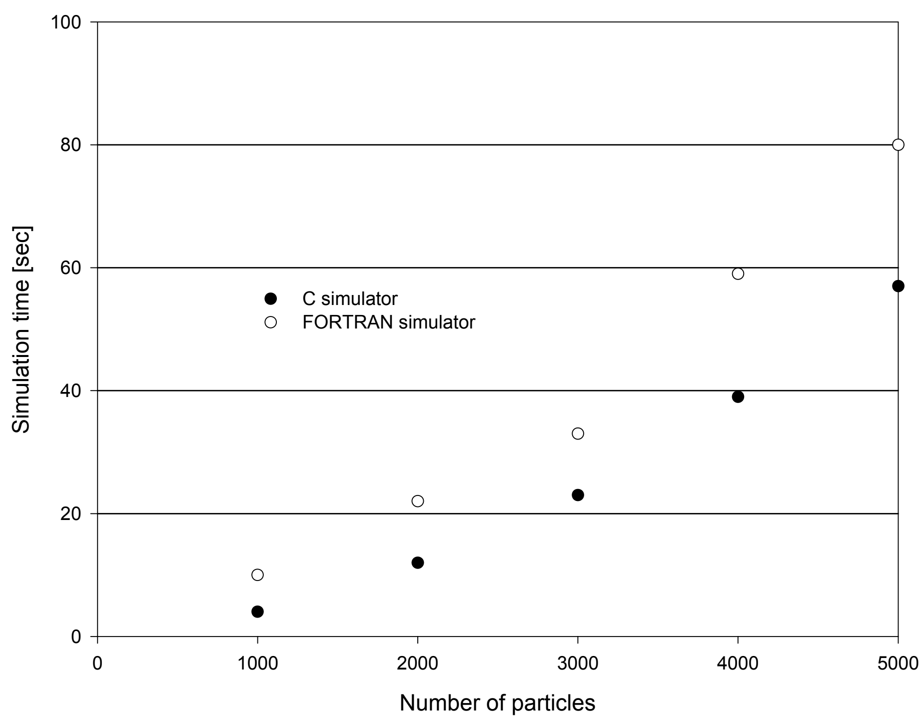


Figure 3–11 *Runtime comparison between the FORTRAN and the C simulator*

These result show that the optimisations made at the C simulator had an effective impact on the runtime of the simulations.

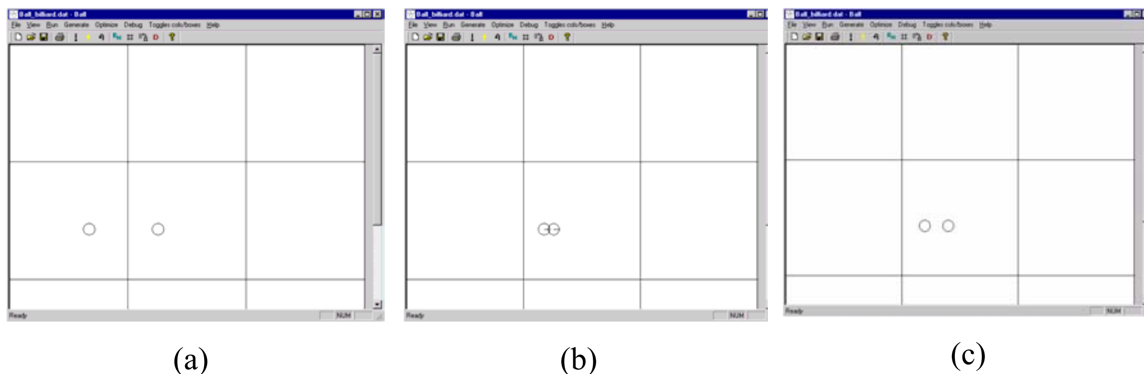
3.5 Validation of the Simulator

The previous sections have described the functionality and behaviour of the software simulator. The next step is to verify that the simulator is doing what it is intended to do. In order to validate the simulator, simulations were performed in the original FORTRAN simulator and then compared with this simulator. Two cases were considered. First, two balls were made to collide and their positions and energies monitored in each cycle. In the second case, 500 particles were run for 1000 cycles. These cases are described in the following sub-sections.

3.5.1 Collision of two balls

In this case only two balls were generated in the system, one with an initial velocity of zero, and the other with an initial velocity of 10 units targeted towards the first ball (see Figure 3–12).

This model was also analysed analytically. In the ideal case, without damping, the moving ball stops once it has collided with the stationary ball and all its momentum and energy are



(a) (b) (c)

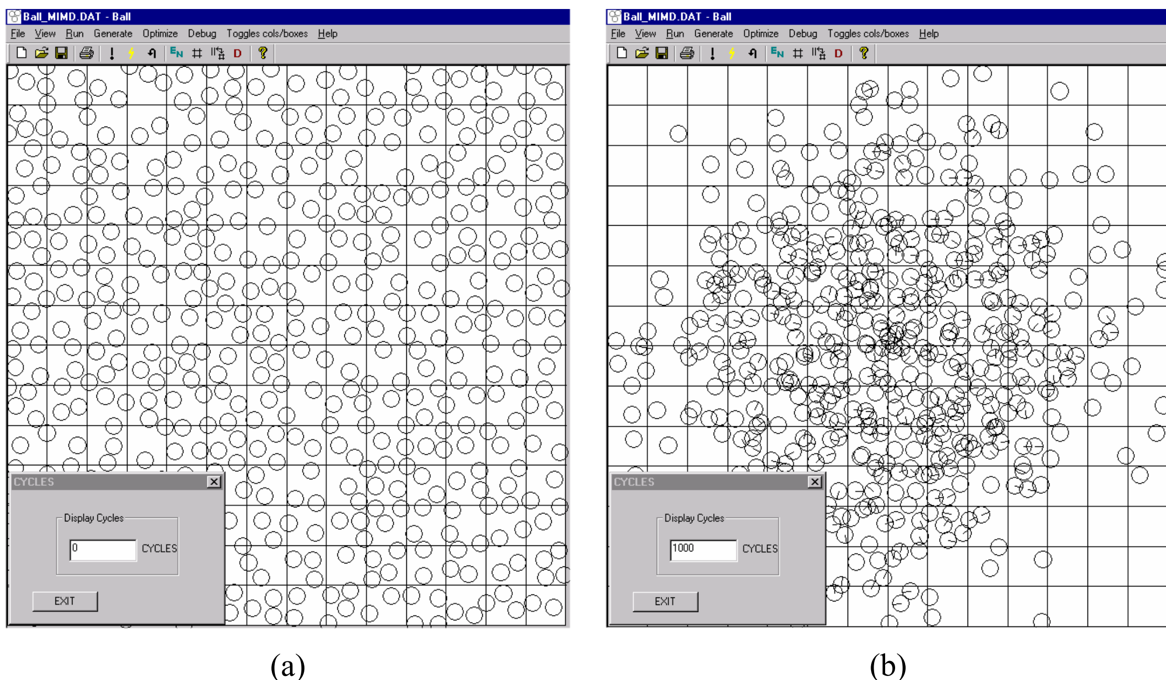
Figure 3–12 *Sequence of the collisions of two balls. (a)Left with initial velocity and right without. (b) Balls colliding (c) right ball has stopped and left ball moves*

transferred to the stationary particle, which starts moving after the collision. Once the contact between both balls is broken, the ball with the initial velocity ceases its motion, while the second ball moves off with the same velocity as the first ball possessed when it was initialised.

Tests with the FORTRAN simulator of Cundall and Strack [1] were also performed for this system and the same results were obtained. The particles behaved identically in both simulations and in the analytical study.

3.5.2 Simulation of Particle Assembly of 500 Balls for 1000 Cycles

In the second validation case, a full assembly of 500 particles was generated (see Figure 3–13(a)) in the simulator and run for 1000 cycles (see Figure 3–13(b)). Differences in the FORTRAN and the C code made it impractical to generate exactly the same assembly of particles simply. Instead the system was initialised with the initial velocities of all particles set to zero (i.e. zero initial kinetic energy), but with gravity switched on in the x and y direction. The final energy was measured for the system at the end of both simulations. The systems energy consists of two components: kinetic energy and potential energy in the contacts.



(a) (b)

Figure 3–13 *(a) Initial and (b) final state of the simulation*

The kinetic energy (E_k) comes from the moving particles and is calculated using the expression given in Eq. 3–1. The potential energy in the contacts is found in the touching balls and is proportional to the indentation of the two particles (see Eq. 3–2 and Eq. 3–4), which is also responsible for the force between them. (Eq. 3–3 shows the force displacement law, by which the force is calculated in the DEM).

$$E_K = \frac{1}{2}mv^2 + \frac{1}{2}I\theta^2 \qquad \textbf{Eq. 3–1}$$

$$E_P = \frac{1}{2}k\Delta n^2 \qquad \textbf{Eq. 3–2}$$

$$\text{with } F = k\Delta n \qquad \textbf{Eq. 3–3}$$

$$E_P = \frac{1}{2}F\Delta n \qquad \textbf{Eq. 3–4}$$

As can be seen from Table 3–3 the energies of the systems once the simulation has finished are almost the same. The differences come from the different initial status of the particles, due to the different particle random generator.

Table 3–3 *Energy comparison between the original FORTRAN and the new C simulator*

	FORTRAN SIMULATOR	C SIMULATOR
Kinetic Energy [J]	570632.51	570630.34
Potential Energy [J]	81.78	80.11

3.6 Discussion

Some of the key features for an efficient software simulator were presented in this chapter; the most important of these features are the grid optimizer and the data structures. The way data is located, deleted and inserted between the linked lists is crucial to efficient simulation.

A never-ending number of improvements could be made, but a trade-off between optimization and time spent to design this simulator has to be accepted. Some examples of possible further improvements include using a different data structure, e.g. a binary tree, using dynamic grid adjustment to provide the optimal grid size in each region of the assembly, and having a more efficient contact check detection scheme [2].

3.7 Summary and Conclusions

A DEM simulator has been presented and analysed in this chapter. Its mechanism, features and data structure have been described in detail.

The simulator has been optimised in order to minimise its run time by using an efficient data structure and by having a grid optimiser. The data structures as well as the data management part have been implemented with most care as these will control the overall performance of the simulation. The in-built grid optimiser has also been described, as the contact check task is very sensitive to the grid size. Large grid sizes can make the simulation time grow dramatically. Too small a box size would make the simulation time grow as well, since the program spends too much time reboxing particles from one box to another.

The simulation time of the C simulator was compared to the FORTAN simulator in order to verify that the optimisations worked. A speed-up factor of 1.5 to 2 could be observed, showing that the advanced data structure and the inbuilt optimisations yield a faster simulation.

The program has been successfully validated with an existing and well-established FORTRAN simulator, achieving the same results, and two test cases were described.

It can be concluded that this software implementation of the DEM functions correctly. This simulator will be used to compare results with the hardware implementation and will be the reference for the hardware implementation in terms of run time, accuracy and numerical stability among other aspects, as will be explained in the following chapters.

3.8 References

[1] Cundall P.A, O.D.L. Strack, "*A discrete numerical model for granular assemblies"*. Geotechnique 29, pp. 1-8, 1979.

[2] Ferrez, J.A., *"Dynamic triangulations for efficient 3D simulations of granular materials"*, PhD thesis, Ecole Polytechnique Federale de Lausanne, Lausanne, 2001.

Chapter 4

REVIEW AND ANALYSIS OF PARALLEL DEM IMPLEMENTATIONS

4.1 Introduction

Multiprocessor systems are commonly used in order to alleviate the extremely time consuming nature of simulations, which would take too long to run on single processor machines. A very common example is weather forecasting, where scientists have only a few days to predict the weather. It makes no sense to have the results of the simulation after 1 or 2 weeks.

As the DEM is a very computationally expensive algorithm, which takes too long to run on single processor machines, many different researchers have tried to map it into parallel processor machines with different degrees of results.

This chapter will review and analyse the different implementations of the DEM on various parallel processing machines. It also presents a novel investigation using the simulator described in chapter 3 into the effect of domain decomposition and system geometry on the speed-up that can be achieved with multiprocessor computer systems.

In order to analyse some previous parallel implementations of the DEM some basic ideas of about parallelism will be presented in the next section.

4.2 Basic ideas about parallelism

There are many definitions of parallel computers. They can be defined as multiprocessor systems consisting of several interconnected processors that can share memory [6]. They can also been thought of as computers with a hierarchical memory, where the memory on another processor is relatively expensive to access, compared to local memory [1].

In a parallel computing environment, effectiveness is measured by run time instead of processor utilization, because the goal of parallel computing is to finish a task as soon as possible. In order to make run time smaller it seems obvious that having a larger number of processors should diminish the run time. However this is not always true in practice, since some algorithms are inherently sequential, and their performance will not be accelerated by a parallel machine. Indeed their behaviour may even be worsened due to synchronization and communication overheads between the multiple processors [2][3].

In 1972 Flynn introduced a taxonomy [4] of the various computer architectures based on the degree of parallelism they exhibit. He divided computer architectures into four main classes based on the number of instruction and data streams.

1. Single instruction stream, single data stream (SISD) machines. Single processor systems can be considered to be SISD machines.
2. Single Instruction stream, multiple data stream (SIMD) architectures, which are systems with multiple arithmetic logic units and a single control processor. Each arithmetic logic unit processes a data stream of its own directed by the single control processor.
3. Multiple instruction streams, single data stream (MISD) machines, in which multiple instruction streams simultaneously act upon the single data stream. Some sources consider that this definition does not apply to any sensible

machine; other sources view scalar pipelined processors as being examples of MISD machines.

4. Multiple instruction stream, multiple data stream (MIMD) machines, which contain multiple processors, each executing its own instruction stream to process the data stream allocated to it.

There are many ways to evaluate the performance of a parallel algorithm, and a very common way is to compare the run time of the best sequential algorithm with the best execution ime on a parallel machine. This comparison is called speed-up (see Eq. 4–1). Traditionally the speed-up has been used to judge the quality of parallel algorithms running

$$speed-up = \frac{runtime\ \ of\ \ the\ \ fastest\ \ sequential\ \ \mathrm{algorithm}}{runtime\ \ of\ \ the\ \ fastest\ \ parallel\ \ \mathrm{algorithm}}$$ ***Eq. 4–1***

on multiprocessors systems. The speed-up achieved by using *N* processors divided by the

$$E_f = \frac{T_s}{T_p N} = \frac{speed-up}{N}$$ ***Eq. 4–2***

value of *N* gives the efficiency (E_f) of the system (Eq. 4–2), as defined by Kuck [5]. For an N-processor system, the ideal speed-up would be of a factor of N. This is also called perfect speed-up or linear speed-up [6]. Linear speed-up is almost impossible to achieve due to synchronization and communication overheads and load imbalance between processors.

Every algorithm can be decomposed into a serial part and a parallel part, as seen in Eq. 4–3, where the sum of the parallel and the serial part is equal to the unity, which represents

$$s + p = 1$$ ***Eq. 4–3***

the entire algorithm. If s represents the serial fraction of the algorithm and p the fraction that can be performed in parallel, then if the parallel component is large, then in principle a good degree of speed-up should be achievable using either a multiple processor system, or a dedicated hardware architecture. Amdahl's law [7] gives an expression for the speed-up

that can be achieved for a certain algorithm as a function of the number of processors used to compute it (see Eq. 4–4). If the algorithm has no parallel part then p=0 and the speed-up would be 1. This is the worst-case

$$speed-up = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}} \qquad \textbf{Eq. 4–4}$$

scenario**,** and means that no speed-up would be achieved. The bigger the parallel part of the algorithm, and the more processors are used, the higher the speed-up should be. The most important consequence of Amdahl's law is that speed-up saturates at a value of 1/s.

Gustafson [8], on the other hand, argued that this is only true when a *fixed* sized problem is run on a varying number of processors, but NOT if the problem size is increased according to the number of processors available. Essentially Gustafson's insight was that for most parallel processing problems, the user can adjust the computational load of a problem (e.g. by using more particles in a DEM, or more elements in a finite element simulation), and will choose the load that can be solved within the available time budget. If a computer becomes available that offers more parallel processing, the user will respond by tackling a bigger problem. Gustafson said that for many problems the parallel part of the program scales with the problem size, while the serial part does NOT grow with it. Eq. 4–5 gives

$$speed-up = \frac{s+p\times N}{s+p} = s+p\times N = N-s\,(N-1) \qquad \textbf{Eq. 4–5}$$

the mathematical expression for Gustafson's law (assuming that s+p=1). This equation also predicts that speed-up is less than linear, but avoids the saturation predicted by Amdahl's law. Gustafson's expression reflects much better the speed-up results obtained for the DEM implementations, as the serial part of the DEM remains almost constant when the problem scales, while the parallel part scales with the problem size, as will be shown in the following chapters.

4.2.1.1 Factors affecting speed-up

There are certain factors that prevent a parallel system from achieving a perfect/linear speed-up. Some of the most important ones are listed below:

1. **Algorithm Penalty:** This penalty is due to the algorithm being unable to keep the processors busy with work. This penalty can further be split into four categories. Distribution, termination, suspension and synchronization overheads.
   - **Distribution Overheads:** These are the costs of having to split the tasks into different processes for the various processors.
   - **Termination Overheads:** Overheads due to idle processes at the end of the computation.
   - **Suspension Overheads:** Total time a process is suspended while it waits to be assigned a task.
   - **Synchronization overhead:** When a process, after completing a part of its task, becomes idle while waiting for other processes to reach a similar point of execution.
2. **Concurrency:** Concurrency is the number of operations that can be performed at the same time. The speed-up is affected directly by the amount of concurrency in the algorithm.
3. **Granularity:** The performance of an algorithm depends on the program's granularity, which refers to the size of the processes in terms of the amount of work for each process.
   - **Fine Granularity:** provides greater parallelism, but leads to greater scheduling and synchronization costs.
   - **Coarse granularity:** has lower scheduling and synchronization overheads, but has significant loss of parallelism by having larger tasks.

4.3 Parallel DEM Implementations

In order to make it possible to simulate present day large-scale DEM problems, parallel processor systems are used. Having multiple processors working in parallel should accelerate the simulation time considerably, but the factors described in the previous section will prevent these systems from achieving linear speed-ups.

This section will discuss some of the previous attempts to parallelize the DEM on different multiprocessor platforms. These attempts are presented in chronological order, so that they can also be evaluated in terms of computational resources available at that time.

4.3.1 Parallel Implementation of the DEM on a Transputer Array

The parallel processing lab at the Colorado School of Mines was one of the first to parallelize the DEM using a parallel computer from Alter Technologies, which has 64 T805 processors [9].

4.3.1.1 System Description

Each node is a 32-bit T805 transputer with its own memory and high-speed communication links. The transputer chip was developed by Inmos Ltd. The name transputer was derived from TRANSistor and comPUTER, since the component was to be a basic building block, like a transistor, but a complete computer on a chip [10].

Each T805 transputer is a full 32-bit processor with an on chip floating point unit, 4 kBytes of on-chip RAM, and 4 MBytes of external RAM. The T805 processor is rated at 30 MIPS (Mega Instructions Per Second). Each processor also has four bi-directional communication links, which can transmit data at rates up to 20 Mbits/second

The transputers are connected physically in a 2-D rectangular grid structure, with each transputer connected only to its north, east, south and west neighbours, as shown in Figure 4–1.

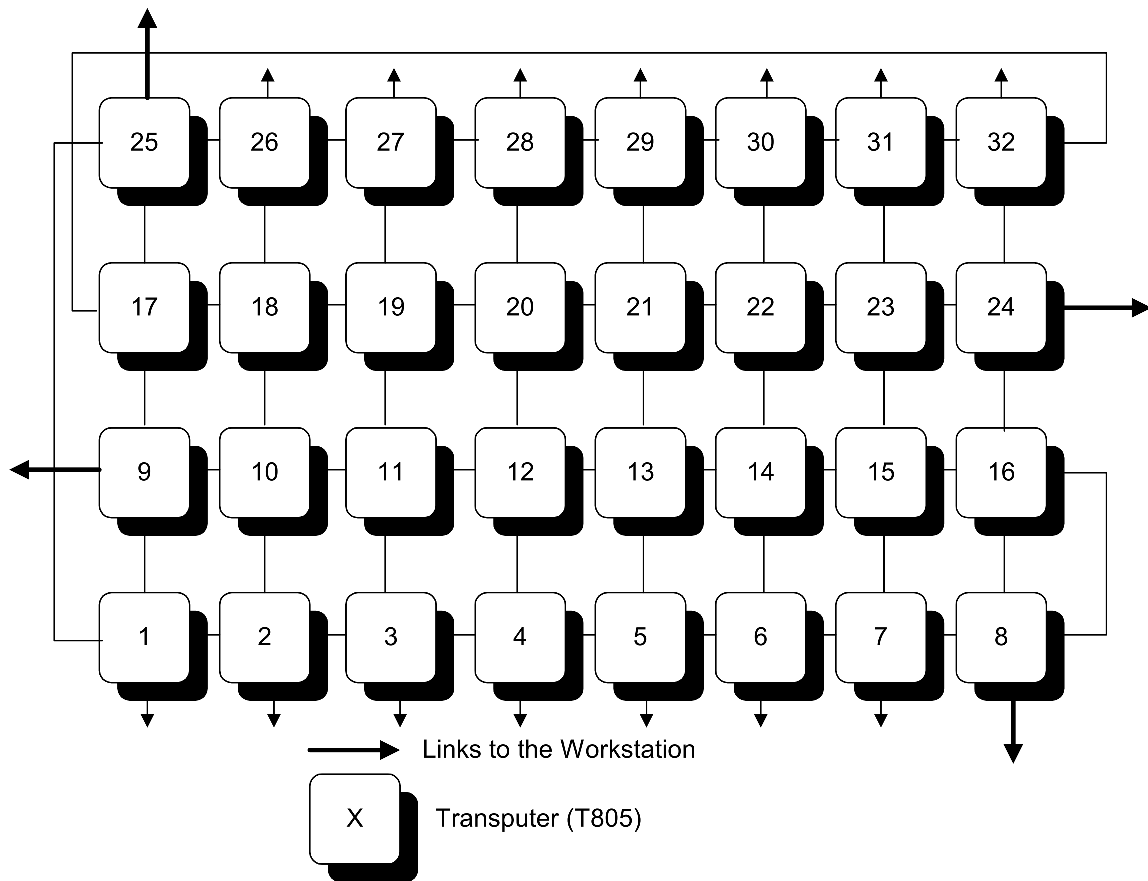


Figure 4–1 *Transputer Network Configuration [9].*

4.3.1.2 Domain Decomposition

One of the most important choices when mapping the DEM onto a multi-processor machine is the way in which the domain is decomposed, and how each part is assigned to the single processors. This means that different processors handle the different geometric areas. The obvious choice is to divide the domain into rows, columns or a grid. In this case the authors decided to split the domain into vertical columns in order to give good load balance, as the only external force applied to these simulations was gravity. This will of course have other disadvantages, depending on the simulation performed, as shown in section 4.4.

A big advantage of the domain decomposition method is that each processor runs a code that is only a minor modification of the serial version. The only notable changes that are needed are in the set up stage, where the domain has to be split among the different

processors, and during simulation, where a new step is needed in order to exchange information between processors.

The basic steps for the simulation of the DEM implemented on a parallel processor machine now become the following, show in Figure 4–2:

```
FOR EACH TIME STEP
     Perform contact checks
     Calculate interparticle forces
     Update particle positions
     Rebox particles transitioning from one cell to another
     Exchange border cells
NEXT TIME STEP
```

Figure 4–2 *Pseudo code for the Multi processor systems*

A new step is introduced in the process, which involves the exchange of data from the border cells from one processor to another.

4.3.1.3 Results

A parallel version of the DEM was implemented on this platform. However, because of synchronization and communication overheads between the transputers, and an uneven load balance, the resulting speed-up was less than linear. As particles move around, some processors get more than others, and the workload of each processor varies significantly. Figure 4–3 shows the speed-up plotted against the number of transputers used. A speed-up of nearly 8 times was achieved for an assembly of 625 particles on a 32-transputer system. This result exhibits a very poor system efficiency (25%) (see Eq. 4–2)

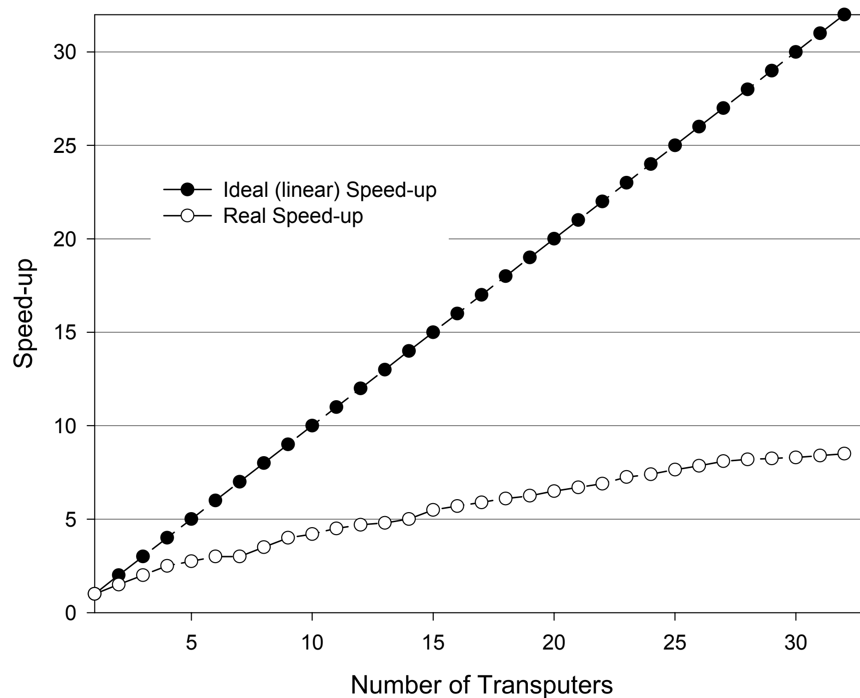


Figure 4–3 *Measured Speed-up [9].*

4.3.2 Parallel Implementation of the DEM on a Cray T3D

The department of mathematics of the École Polytechnique of Lausanne also designed a parallel version of the DEM running on a Cray T3D massively parallel computer [11].

4.3.2.1 System Description

The Cray T3D is a Multiple Instruction stream Multiple Data stream (MIMD) machine. This means that the data is distributed among the processing elements and each processor works independently from each other. The number of processing elements (PE's) can be anywhere from 32 to 2048. Each element of the T3D is a DEC 21064 (EV-4) RISC chip with its own memory, memory controller, and prefetch queue. A single node on the T3D consists of two PE's combined with a network switch. The DEC 21064 Alpha chip used runs at a speed of 150 MHz, with a theoretical peak performance level of 150 megaflops.

The nodes on the Cray T3D are connected in a 3-D torus configuration. The connecting network operates at a speed of 150 Mhz, with bidirectional transfers and separate routing

for data and communication information. The bandwidth for a 2048-processor T3D is approximately 154 gigabytes per second.

4.3.2.2 Domain Decomposition

The authors simulated granular media assemblies confined to a rectangular box where the only the external force was gravity. In this case it is enough to divide the domain into simple vertical stripes, assigning each strip to a processing element in order to have an acceptably balanced system.

As in the previous example, calculations rely on locally available data and therefore each processor works on its own. Near stripe borders, there are many discs that contact discs in the neighbouring stripe, so access to remote data is needed.

4.3.2.3 Results

The benchmark used to test the efficiency of this parallel algorithm was a medium composed of 200 000 small discs stacked at the bottom of a box. A very large disc was allowed to fall under gravity onto the medium, and subsequent behaviour was simulated. The simulation of 0.005 seconds of real time took 500 steps of the parallel algorithm, which took about 12 minutes on the Silicon Graphics Indigo machine where the serial version was run. The results of this parallel implementation are shown in Figure 4–4. Though the speed-up starts off almost linear with a small number of processors, when more processors are added the speed-up curve starts to bend and for 128 processors only around half of the expected speed-up is achieved. The authors of [11] suggest the following improvements in order to retain linearity in the speed-up curve:

- Better data and workload repartition
- Reduction of communications
- Better local memory management (more efficient cache utilization)

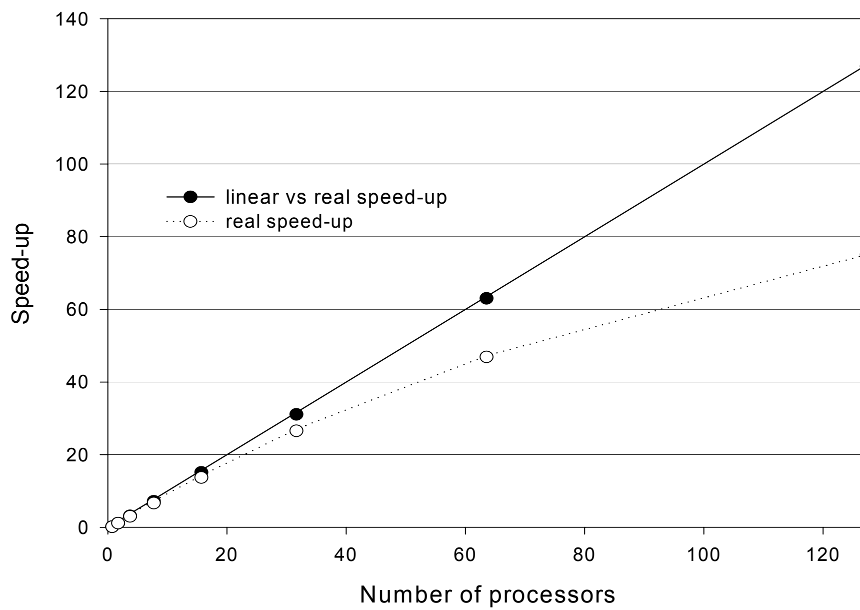


Figure 4–4 *Speed-up of the parallel implementation as a function of the number of processors [11].*

4.3.3 Parallel Implementation on a Swiss-T0-Dual machine [12]

The CSIRO Mathematical & Information Sciences performed another parallel implementation of the DEM, this time on a Swiss-T0-Dual machine.

4.3.3.1 System Description

The system was implemented on a Swiss-T0-Dual machine. This is a cluster computer system consisting of 8 Digital Alpha 21164 dual-processor boxes [13]. Each processor has a 4 Mbytes level 3 cache, with the total system having a distributed memory of 8 Gbytes and a peak performance of 16 GFlops. The processors are connected via both an EasyNet bus and a Fast Ethernet switch. Each dual processor box has one PCI-based connection to the EasyNet bus and a Fast Ethernet port. This leads to a memory bandwidth between boxes of 35 Mbytes/s for the EasyNet bus and 10Mbytes/s for the Fast Ethernet port. The boxes are run using Digital's UNIX system.

4.3.3.2 Domain Decomposition

In this case the domain was divided into subdomains by slicing the domain into columns, choosing the number of subdomains equal to the number of processors available. In order

to have load balance, the subdomains are chosen to contain the same number of particles, as shown in Figure 4–5.

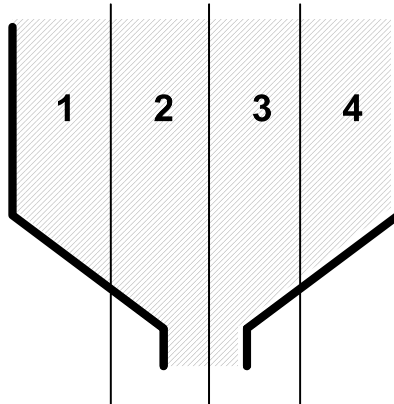


Figure 4–5 *Domain decomposition for the Hopper discharge [12]*

As in the previous examples, the particles from the neighbouring subdomains are copied to the neighbouring processor. The width of this layer is chosen to ensure that all interactions can be calculated from local data.

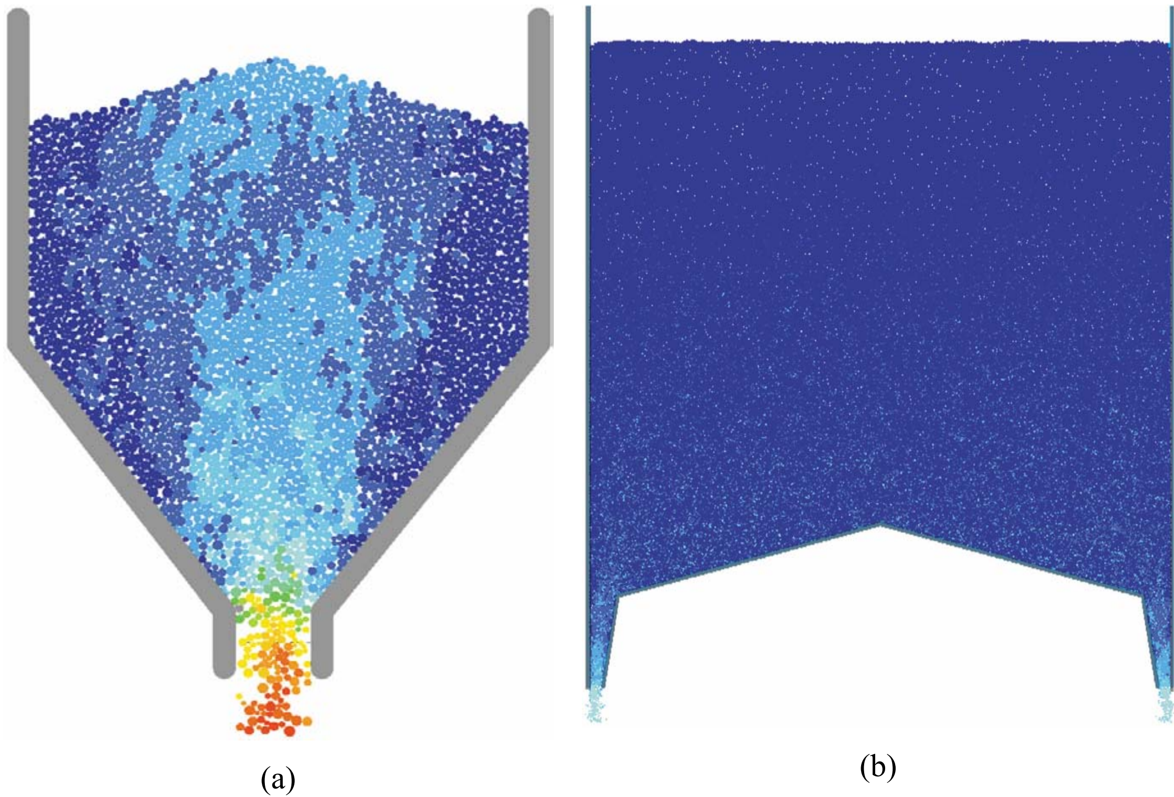
4.3.3.3 Results

Two experiments were set up in order to measure the performance of this parallel implementation. As hoppers are common storage devices for granular media, the 2-D flow from two different slot hoppers were considered:

1. A generic single-port hopper, with a width of 2.4 m initially filled with 3545 circular particles with a distribution of diameters from 20 to 100 mm (see Figure 4–6a).
2. A dual-port hopper of 40 m width. This hopper initially contained 200 000 particles having a distribution of diameters from 50 to 200 mm (see Figure 4–6b).

For each hopper flow, computations using different numbers of processors were performed using the Fast Ethernet (100 Mbits/s) and the EasyNet interconnect. For comparison, computations of the serial code on a single processor system were also made. These performance measurements were computed for:

1. 1 second (around 55 000 time-steps) for the single-port hopper
2. 0.1 seconds (around 1100 time-steps) for the dual port hopper



(a) (b)

Figure 4–6 *Single–port (a) and dual-port hopper (b) [12]*

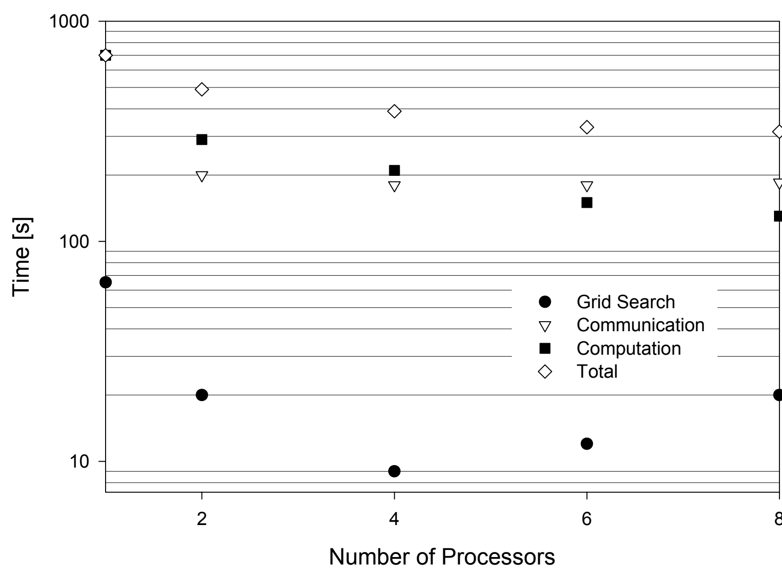


Figure 4–7 *Computation time required for the individual tasks of the DEM simulation of the single-port hopper [12]*

Measurements of the time needed by the individual tasks of the DEM were also made, in order to determine the time spent in computation, communication and synchronization. The results of timing measurements for the code are presented in Figure 4–7 and Figure 4–

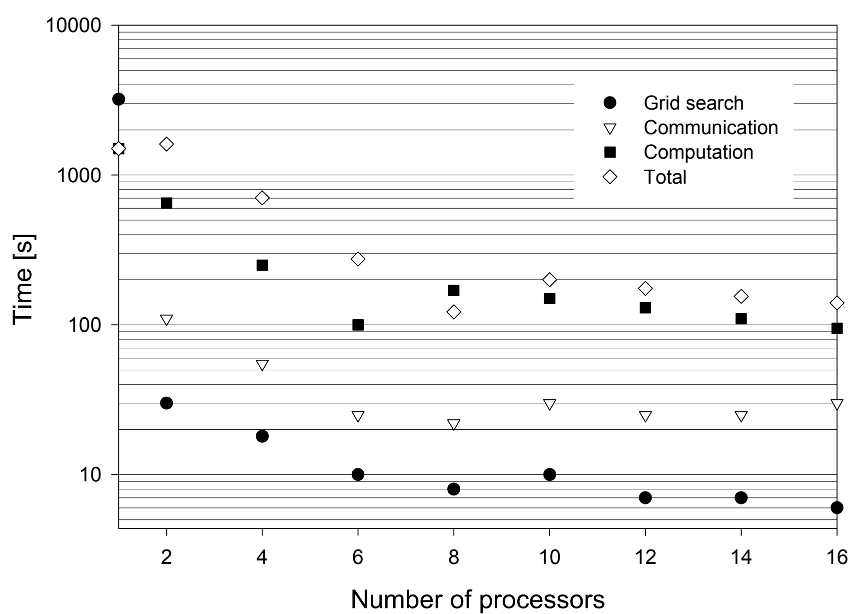


Figure 4–8 *Computational time required for the individual tasks of the DEM simulation of the dual-port hopper [12]*

8. Only small differences were measured for the simulation using the Easy Net and the Fast Ethernet interconnect bus.

For both hopper discharges the computation time decreases with the number of processors, as would be expected. However the inter-processor communication time remains approximately constant, flattening the speed-up curve (see Figure 4–9) as the number of processors increases, because the communication time dominates the total time taken.

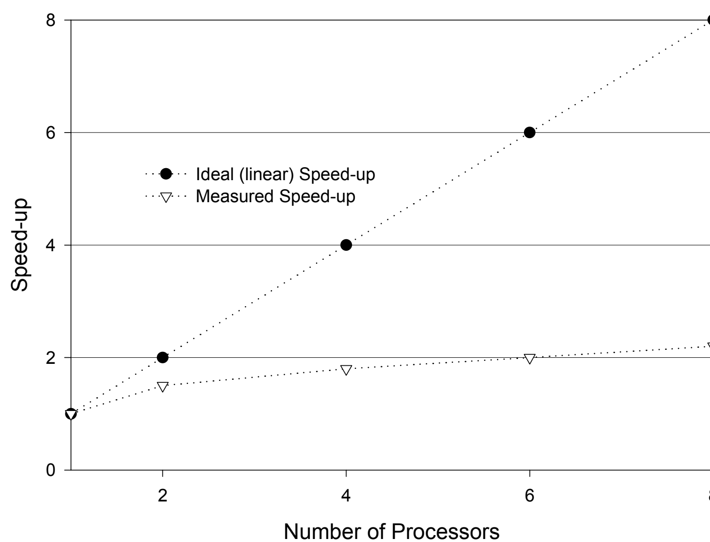


Figure 4–9 *Measured speed-up for the DEM simulations of the single-port hopper [12]*

For the dual-hopper case, the performance scales linearly until the 8th processor, after which synchronization and communications overheads become substantial, degrading the speed-up (see Figure 4–10).

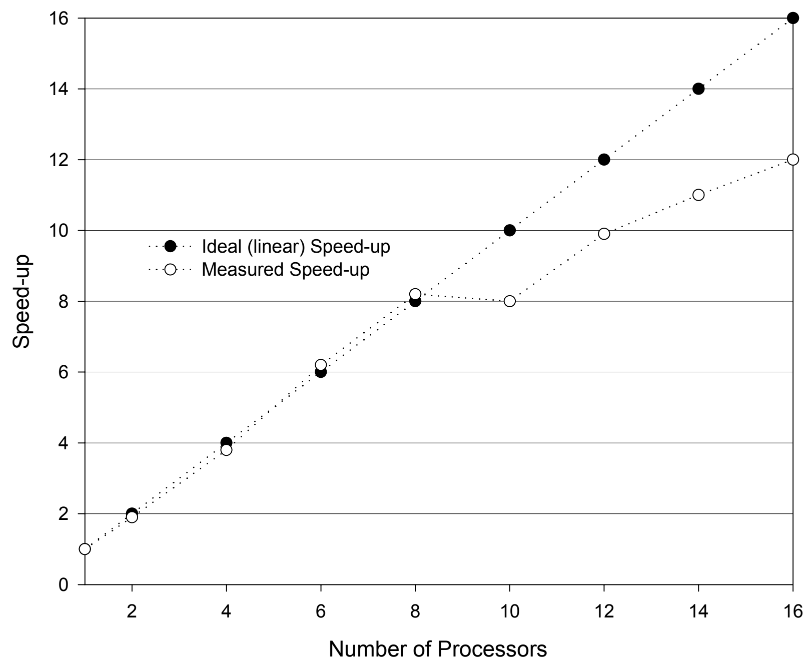


Figure 4–10 *Measured speed-up for the DEM simulations of the dual-port hopper [12]*

4.3.4 Parallel Implementation of the DEM on various Hardware Platforms

Another parallel version of the DEM was implemented at the Northwestern University on a variety of hardware platforms and compared with their serial version. Comparison of Single Instruction Multiple Data stream (SIMD) and Multiple Instruction Multiple Data stream (MIMD) operation were also performed [14], showing that the MIMD implementations provided the best overall parallelization.

4.3.4.1 System Description

The SIMD code was implemented on a Connection Machine 5 (CM5) system, manufactured by Thinking Machines. The MIMD codes were ran on a CM5 system, on a Scalable POWER Parallel System 2(SP2) IBM system, and on a small network of Intel™ Pentium© PRO PC systems using Microsoft's™ Windows© NT operating system.

The CM-5 system consists of a set of *processing nodes* Each processing node consists of a SPARC processor operating at 32 MHz or 40 MHz. Together with its four vector units, a 32 MHz processing node is capable of performing 64-bit floating point arithmetic at a rate of 128 megaflops. The control processor, which is also referred to as the partition manager,

is a Sun SPARCstation. The CM-5 is a distributed memory machine. The data network is capable of delivering messages to nearby nodes at rates up to 20 MBytes/s

The IBM SP2 is a general-purpose scalable parallel system based on a distributed memory message-passing architecture. The SP2 system consists of 2 to 10 POWER2 Architecture RISC System/6000 processor nodes interconnected by a switched network. Each processing node has its own private memory and its own copy of the AIX operating system.

The author of [14] states in the paper that communication between processors is the bottleneck of these systems, as each processor can rapidly access its own local memory, but must access data local to other processors through a slow communication network.

The idea behind the MIMD implementation is to schedule inter-node communication concurrently with computation to achieve the greatest possible level of parallelism.

4.3.4.2 Results

The MIMD algorithm is based on the optimised serial algorithm; however the entire model is divided into volumetric zones that are assigned to separate processors. During any time step a processor calculates block forces and motions within its zone until data arrives from another processor. While the processor continues with the rest of the zone, data travels to another processor.

A system with 10 000 particles was generated and ran on the different platforms. Table 4–1 compares the effect of hardware for both serial and parallel implementations.

Table 4–1 *Listing of the SIMD and MIMD implementations of the DEM on different HW platforms*

	Run Time [s]	Speed up over serial version	Parallelization
CM5 serial version	2035	1	N/A
CM5 64 nodes	34	59.85	93 %
PC (Pentium© Pro at 200 MHz)	252	1	N/A
2 PCs (Pentium© PRO at 200 MHz) networked	133	1.89	95 %
IBM SP2 1 processor	277	1	N/A
IBM SP2 10 processors	28	9.89	99 %

The author concludes that the MIMD provides an overall best parallelization for SMALL NUMBER OF PROCESSORS than the SIMD implementation

4.4 Modelling of a Multiprocessor System

The previous examples have shown some parallel implementations of the DEM on several different parallel machines. The results vary from one implementation to another, because of the simulations types, as well as the platform properties. However, there are some underlying similarities. Simulations with large numbers of processors achieve speed-ups that are far less than linear. Also, in most of these investigations, the problem was deliberately simplified and made easy for parallel processing by considering the movement of particles under gravity in a domain decomposed into vertical columns. This means that there is very little horizontal movement of particles across sub-domain boundaries, so there is low communications overhead, and little tendency for loads to become imbalanced.

In order to get a deeper understanding of these systems and their bottlenecks, and also to investigate pathological cases that are difficult for parallel processing, a DEM simulator was developed that models the behaviour of multiprocessor systems. This simulator is based on the DEM simulator described in detail in chapter 3, and emulates the effect of parallel processing by splitting the simulation into separate processes, and recording the

amount of communication required between processes, and when processes have lost synchronisation, so that some would have to stall. The data structure has been changed in order to have a regular domain decomposition depending on the number of processors of the system. The user can choose to divide the domain into:

- Columns
- Rows, or
- Cells

4.4.1 Domain Decomposition

If the user decides to split the domain into cells, the number of processors to be used in the x and in the y direction needs to be given as well. Figure 4–11 shows an example of the different regular domain decomposition techniques for 6 processors.

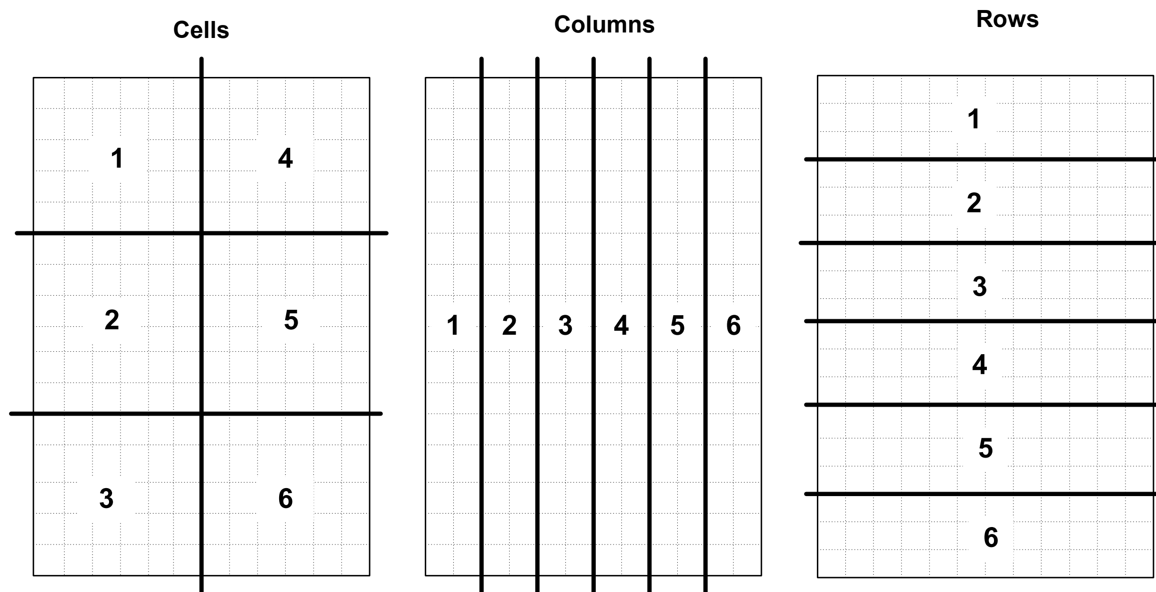


Figure 4–11 *Example of the different regular domain decomposition types*

Each processor is responsible for the domain it has been assigned. However, before the simulation can move on to the next time step, all the processors must have finished their computation for the present time step. Therefore the performance of the system will depend on the slowest processor, i.e. the one that has the most data to compute, i.e. the

processor that has the most particles in its domain. It is necessary to distribute the work amongst the processors as evenly as possible, so that the system is as balanced as possible.

Having an unbalanced system will result in a poor system efficiency. In some simulations, a prediction can be made of how the system will behave, and therefore the domain decomposition can be chosen in order to have a similar number of particles in each cell. In other cases a more complex solution can be taken and dynamically re-partition the domain to equalise the load.

4.4.2 Dynamic Load Balancing

The goal of load balancing can be defined as:

> Given a collection of tasks comprising a computation and a set of computers on which these tasks may be executed, find the mapping of tasks to computers that results in each processor having approximately equal amount of work [15].

In order to have a useful load balancing scheme it must be determined, *when* to perform the load balance. This implies two stages:

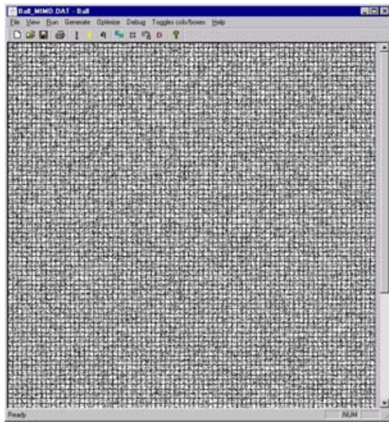
1. Detect the load imbalance
2. Determine if the cost of load balancing exceeds the possible benefits.

In the case of the DEM, if the efficiency of the system reaches a certain user defined minimum threshold value, load balancing can be performed.

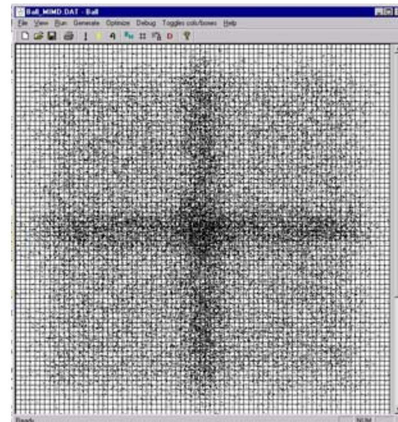
In all the examples given in section 4.3, none of the systems made use of dynamic load balancing. The reason for this is that the simulations in these examples are either quasi-static or the only external force is gravity, which makes particles head toward the bottom of the domain, making it unnecessary to perform dynamic load balancing.

4.4.3 Multi-Processor Modelling

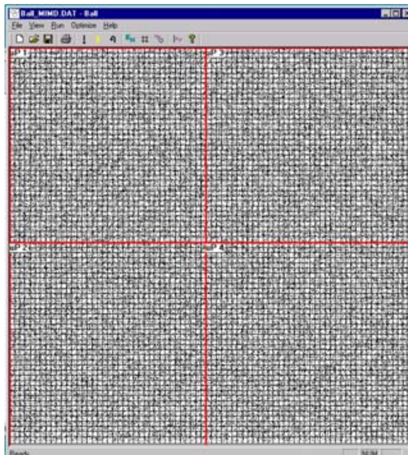
Different kinds of simulations were performed in order to understand the bottleneck of these systems. The simulator registers in a log file the time needed by each processor to perform the contact checking, forces and position update as well as the time needed to pass the particles from one processor to another based on the bus bandwidth, which is specified as a variable in the simulator, and for these simulations was set to 32bits@33 MHz.



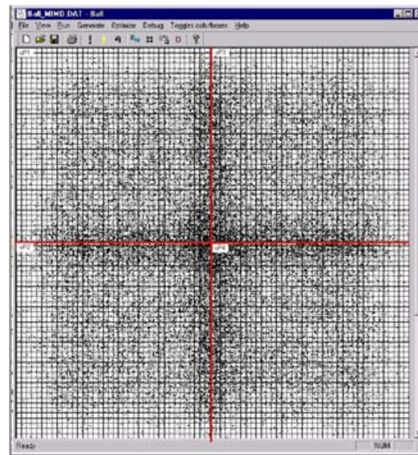
(a_1)



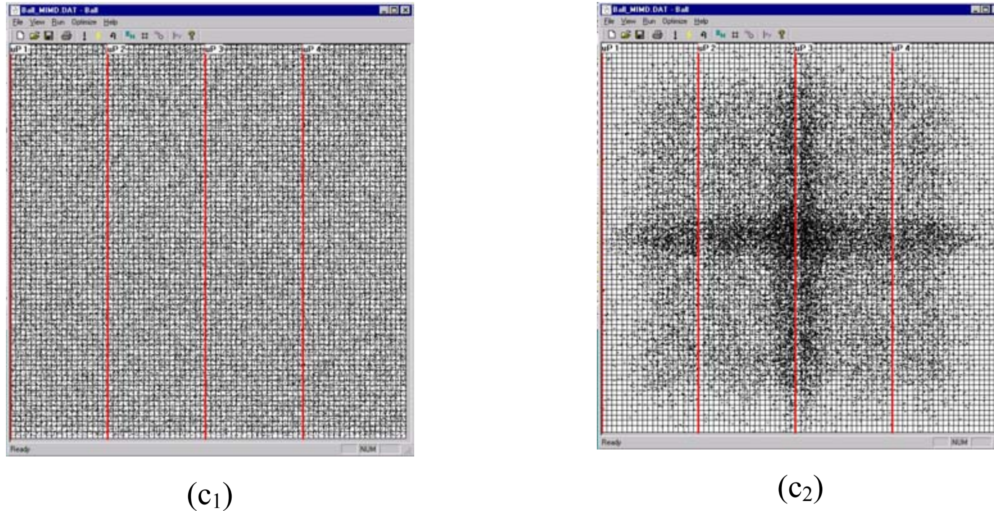
(a_2)



(b_1)



(b_2)



(c_1) (c_2)

Figure 4–12 *Domain decomposition types in the multi-processor modelling SW*
(a_1) is the initial state of the uniprocessor system's and (a_2) its final state, (b_1) is the initial state of a 4 processor system split into cells and (b_2) its final state, (c_1)is the initial state of a 4 processor system split into columns and (c_2) its final state

The domain was decomposed into columns and into cells in order to measure how the domain decomposition technique affects the total system performance in the worst case. A system of 50 000 particles was generated initialising the velocity of the particles to the centre of the domain (this was chosen to be pathological for the domain decomposition used). After 1000 cycles the system is completely unbalanced as shown in Figure 4–12.

Simulations of this system for different numbers of processors were performed. Figure 4–13 shows the time needed to run this simulation for 1000 cycles for different number of processors. It can be seen that the time needed to compute the same simulation is smaller if the domain is split into cells than if it is split into columns. This is due to the fact that, for this example system, the load is more balanced if the domain is split into cells.

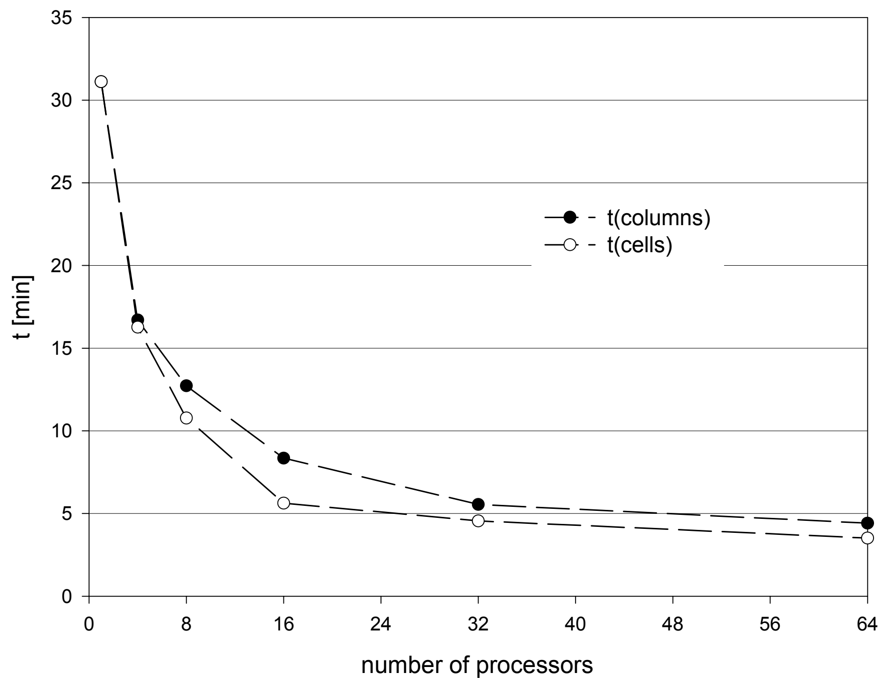


Figure 4–13 *Simulation time for different number of processor systems*

When the number of processors becomes large (32 to 64 processors) the time difference between the domain decomposed into columns and cells becomes almost constant, since the communication overheads become the dominant part of the simulation time.

The multiprocessor modelling software also registers the time needed in each cycle to perform the main computational tasks of the DEM (contact checking, forces update, position update, and communication between processors, see Figure 4–15). In this figure the asymptotic behaviour of the system can be observed, noting that after 32 processors little gain in term of speed-up can be made, as communication overheads remain almost constant.

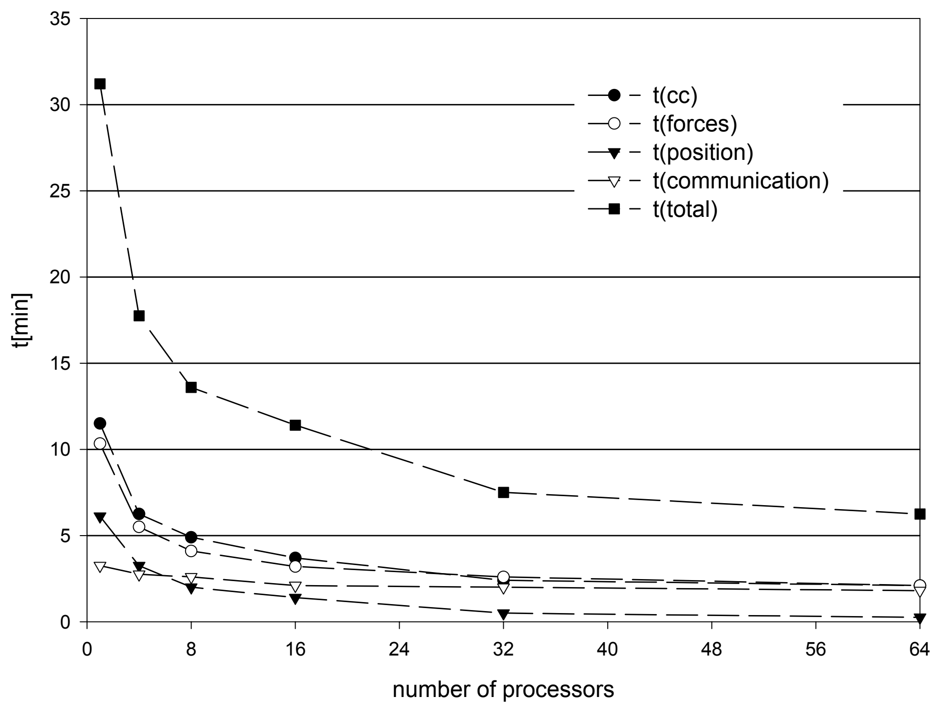


Figure 4–14 *Simulation time for different number of processor showing the time spent by each unit (domain split into cells)*

Figure 4–15 shows the initial and final condition of the 8 processor system, splitting the domain into columns and cells.

The same simulation as above was performed, initialising the particles' initial velocity towards the centre of the domain. The time needed to compute the main tasks was computed after every cycle.

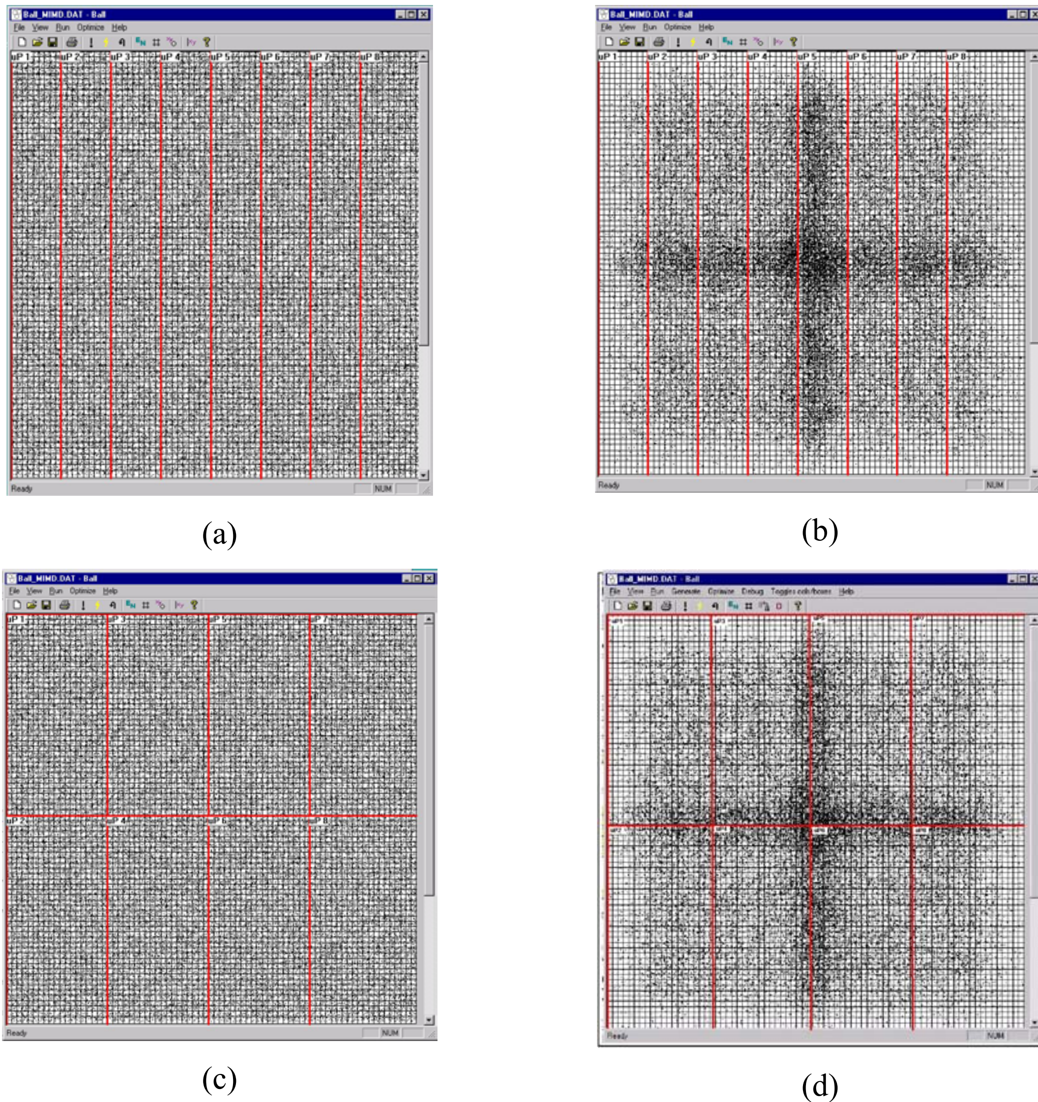


Figure 4–15 *Initial and final conditions for a simulation decomposing the domain in columns (a)(b) and cells(c)(d)*

Figure 4–16 and Figure 4–17 show the time spent by the processors in each of the tasks in the last simulation cycle (1000th cycle). As seen in Figure 4–16 (domain split into columns) the load is far less balanced when the domain is split into columns and therefore the processor in the centre of the domain (the 5th processor) has a much heavier load than the others. In the case where the domain is split into cells (see Figure 4–17), the work is spread more evenly among the processors in the centre of the domain.

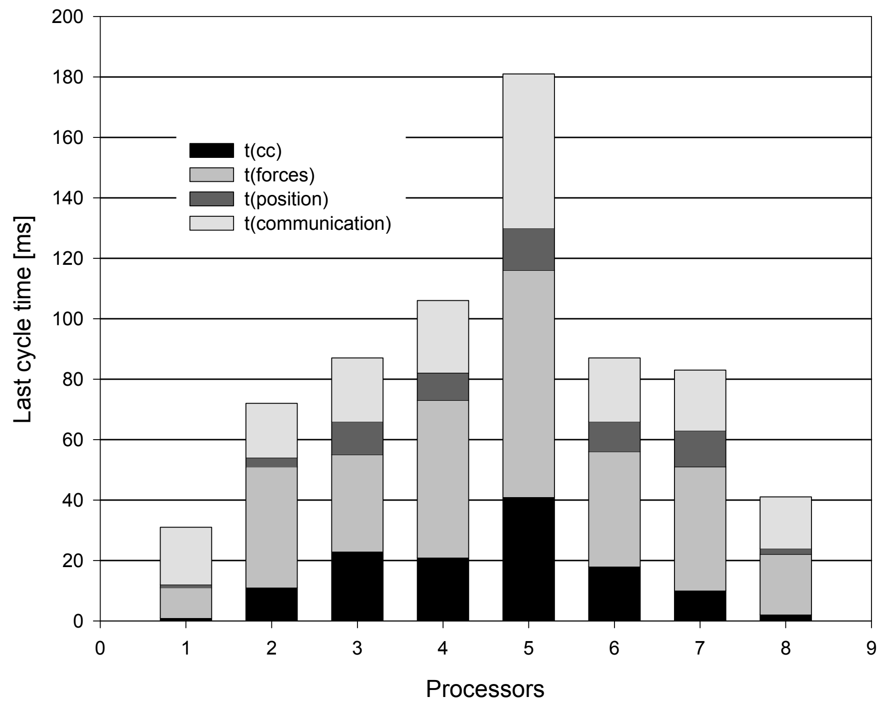


Figure 4–16 *Time needed to perform the last cycle in a simulation with 8 processors decomposition the domain in columns*

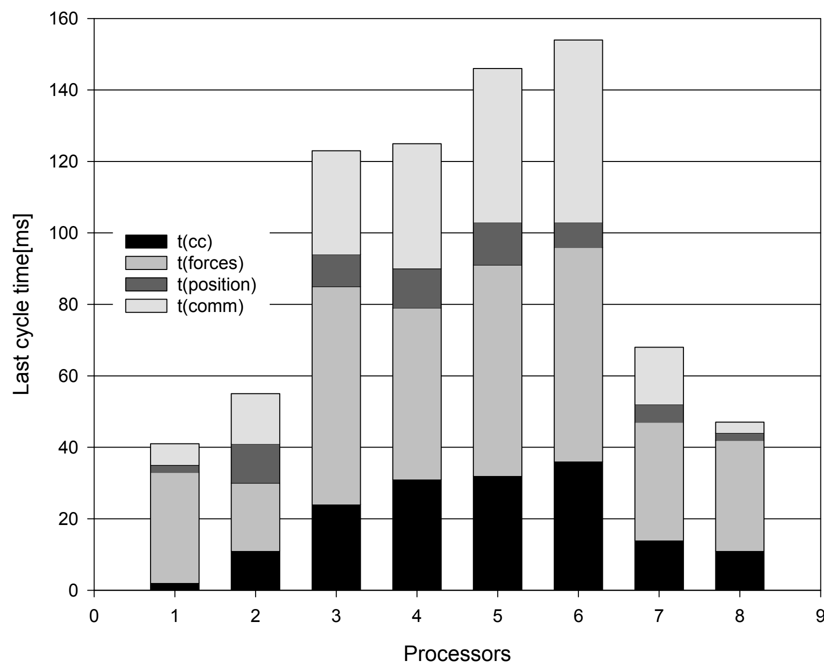


Figure 4–17 *Time needed to perform the last cycle in a simulation with 8 processors decomposition the domain in cells*

The previous simulation showed the importance of the domain decomposition technique in order to have the fastest possible simulation. Measurements were done for a pathological case, when all the particles' initial velocities pointed to the centre of the domain. In order to investigate the influence of the communication overheads on the total simulation time, three simulations were set up decomposing the domain into columns:

1. Only gravity is switched on and all particles have zero initial velocity (best case)
2. The particles' initial velocities point to the centre of the domain (pathological case)
3. The particles' initial velocities are randomly initialised.

Figure 4–18 shows how these systems behave for a different number of processors. For the case where only gravity is considered, given that the domain is split into columns no

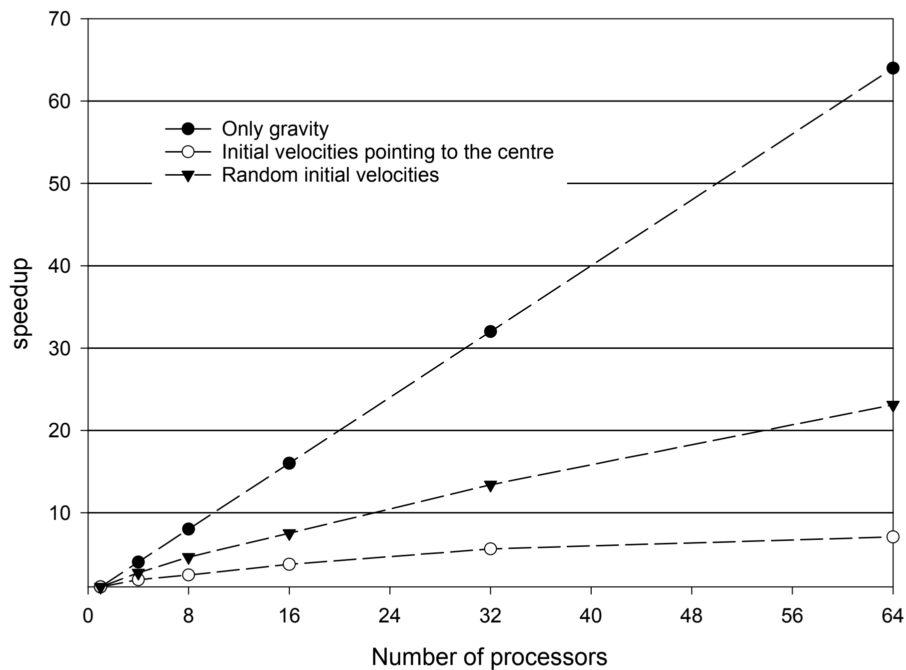


Figure 4–18 *Speed-up for different initial velocities*

particles transition from one domain to anther, hence no communication is needed between the processors. Perfect speed-up is achieved for this particular case.

For the worst case, when the initial velocities of the particles are initialised towards the centre of the domain, the worst possible results are obtained as communication overheads as well as load balancing problem appear heavily.

The third case, when the particles are initialised with random velocities shows an intermediate result, between the perfect, linear speed-up and the worst case.

4.5 Summary of the parallel DEM implementations

The DEM is an extremely effective way to simulate the behaviour of granular materials and even solids by building them as large conglomerates of particles bonded together. The only drawback is that it is computationally very expensive.

The only actual way to simulate realistic problems with cost effective systems (reasonable simulation time at a reasonable cost) is to use multiprocessor systems. The ideal situation using these systems is to have a linear speed-up. This means being N times faster with N processors than with solely one. In practice this is never achieved. Multiprocessor systems mean also that the problem has to be partitioned into sub-domains and every partition has to be assigned to one processor. Every processor needs to know what is happening in the contact area with the sub-domains allocated to other processors, and thus has to communicate with them. The processors also need to synchronize with each other in order to restart a cycle at the same time. This makes communication, synchronization and load balancing of the processors a very important issue, which could become the bottleneck of every design.

A number of implementations on multiprocessor systems were surveyed in section 4.3. Most of them were based on a SIMD approach, where all the processors were performing the same operation at the same time. A MIMD approach was also described which showed very good parallelism results, but only for a small number of processors. Communication and synchronization overheads grow with the number of processors degrading the performance dramatically.

The most important factors that affect the efficiency of these multiprocessor systems are:

- **Communication:** Data in the overlap region needs to be exchanged between processors.
- **Load balancing:** Allocation of subdomains to processors will invariably result in an uneven distribution of work.
- **Synchronization:** Processors need to be simultaneously at a certain point in the algorithm.

A multi-processor software simulator was presented in section 4.4, showing that the domain decomposition techniques has a very important impact on the system performance. In general the user should aim to minimize the contact area between the sub-domains allocated to the processors in order to have the samllest possible number or particles transitioning from one sub-domain to another. (Though there are specific cases, e.g. a domain of particles falling under gravity, where there is an overall systematic bias to particle motion that should also be taken into account in performing the domain decomposition.) This simulation results also showed that as the number of processors grow the system efficiency decreases dramatically, as the communication overheads become the predominant computational part of the simulation.

All these problems point to one question. Is there any other way to go? Can a different approach bring better results? The answer, in the opinion of the author, is yes. There are several possible directions that involve customisation of the computing resources to the DEM problem. These range from custom multiprocessor / DSP systems, to dedicated hardware architectures.

4.6 Use of Field Programmable Gate Arrays for the DEM

The complexity of Field Programmable Gate Arrays is continuously increasing. Modern devices allow designers to implement complete systems with minimal requirement for off-chip resources. One promising application area for these devices is to form FPGA-based reconfigurable co-processors within standard computers, which can be used for algorithm acceleration [16] [17]. For the right type of application, such a reconfigurable computer

can rival the expensive parallel computers that are normally used to accelerate computationally expensive algorithms. FPGAs thus open a new window to low cost hardware acceleration.

The DEM has properties that suggest that it may be suitable for acceleration using FPGAs:

- It exhibits an enormous degree of parallelism
- It is an explicit self-correcting algorithm.

It is therefore tempting to examine how well the DEM would map into an FPGA.

4.7 Summary and Conclusions

The DEM is one of the most suitable algorithms to simulate the behaviour of granular materials. Nevertheless its extensive use is hampered by its extremely high computational demands, since every single particle is considered individually, and disturbances cannot travel beyond neighbouring particles in one time-step, necessitating a very short time step (typically of the order of milliseconds of physical time).

A number of parallel implementations of the DEM on multiprocessor platforms were surveyed. They all suffered the same problems: synchronization and communication overheads between processors, as well as poor load balancing between processors, made the speed-up less than linear.

Novel computational approaches have to be considered in order to find a more efficient way to accelerate the DEM at an affordable price whilst obtaining the computational results within a reasonable real time. FPGAs are one way as they can serve, among many other applications, as co-processors within standard workstations to form hardware accelerators. This work will analyse the use of FPGAs as a hardware accelerator for the DEM and study two implementations of the method.

4.8 References

[1] Smith, B., Bjorstad, P., Gropp, W. *"Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations",* Cambridge University Press, New York, 1996.

[2] Shiva, S.G. *"Pipelined and Parallel Computer Architecture"*, Harper Collins College Publisher, New York, 1996

[3] Flynn, M.J. *"Computer Architecture"*, Jones and Barlett Publishers, Sudbury MA (USA)1995.

[4] Flynn, M.J. *" Some computer organizations and their effectiveness"*, IEEE Trans. Computers C-21 (9). 948-960, 1972.

[5] Kuck, D., Budnik, S., Chen, S., Towle R,Strebendt, R., Davis, E.,Han, J. Kraska, P., Muraoka, Y., *" Mesurements of parallelism in ordinary Fortran programs",* IEEE Computer, 7(1), pages 37-46, January 1974.

[6] Zargham, Mehdi R., *"Computer Architecture. Single and Parallel Systems"*, Prentice-Hall, New Jersey, pp 300 –302,1996.

[7] Amdahl G. H., "*Validity of a Single-Processor Approach to Achieving Large-scale Computer Capabilities*", AFIPS Conf. Proc., Vol. 30, 1967, pp 483-485

[8] Gustafson, J.L., *Re-evaluating Amdahl's Law*, " Commun. ACM, 31(5), pp. 532-533. May 1988.

[9] Hustrulid, A. I." *Parallel implementation of the discrete element method*", Colorado school of mines, USA. Available at http://egweb.mines.edu/dem/

[10] Hull M.E.C., Crookes D. and Sweeney, P.J., *" Parallel Processing. The Transputer and its Applications"*. Addision-Weseley Publishing Company, 1994

[11] Ferrez J.A., Mueller D., Liebling T. M. "*Parallel Implementation of a distinct element method for granular media simulation on the Cray T3D*." EPFL Supercomputing review -SCR No 08, Lausanne, Nov. 96

[12] Sawley M.L, Clearly P.W "*A Parallel discrete element method for industrial granular flow simulations*", CSIRO Mathematical & Information Sciences, Clayton, Australia., EPLF Supercomputing review -SCR No 11, Nov. 99

[13] Gruber, R., Dubois-Pelerin, Y., *"Swiss-Tx: First experiences on the T0 system",* EPFL Supercomputing Review, SCR 10, 19-23, 1998.

[14] Dowding, C.H., Dymytryshyn, O., Belytschko, T.B., *"Parallel processing for a discrete element program"*, Elsevir Science Ltd., Computer and Geotechnics 25, 281-285, 1999

[15] Watts, J., Taylor S., *" A Practical Approach to Dynamic Load Balancing"*, IEEE Transactions on Parallel and Distributed Systems, 1996

[16] Hartenstein, R. W., Becker, J., et al. "*High-Performance Computing Using a Reconfigurable Accelerator"*. In CPE Journal, Special Issue of Concurrency: Practice and Experience, John Wiley & Sons Ltd., 1996.

[17] Hartenstein, R., Herz, M., Hoffmann, T., Nageldinger, U. "*On Reconfigurable Co-Processing Units"*. In Proceedings of the Reconfigurable Architectures Workshop (RAW'98), Orlando, Florida, USA, March 30, 1998.

CHAPTER 5

HARDWARE IMPLEMENTATIONS OF THE DEM ON A FIELD PROGRAMMABLE GATE ARRAY (FPGA)

5.1 Introduction

Reconfigurable Computing is based around the use of Field Programmable Gate Arrays (FPGAs) to form co-processors that can be configured to provide custom hardware accelerators. The types of problem that can benefit from reconfigurable computing are established by the properties of the FPGA. In general, FPGAs are good at tasks that use short word length integer or fixed-point data, and exhibit a high degree of parallelism.

Traditionally reconfigurable computing has been regarded as unsuitable for problems in computational mechanics, such as are used in civil, mechanical and chemical engineering, because these problems generally require floating point arithmetic and long word length. A notable exception to this generalisation, found in this research, is the Discrete Element Method. The DEM uses simple arithmetic operations in a massively parallel way on a large data set, and, as shown by this current work, it *can* retain numerical stability using short word length fixed-point data.

This chapter will present two hardware designs for the DEM implemented on an FPGA. An introduction to FPGAs, their evolution and the current state of the art, will also be presented.

5.2 Motivation

The complexity of Field Programmable Gate Arrays is continuously increasing, and state of the art FPGAs now have up to 10 million system gates [1]. Such devices allow designers to implement complete computational systems with minimal requirements for off-chip resources. One promising application area for these devices is to form FPGA-based reconfigurable co-processors within standard computers, which can be used for algorithm acceleration. For the right type of application, such a reconfigurable computer can rival the expensive parallel computers that are normally used to accelerate computationally expensive algorithms. FPGAs thus open a new window to low cost hardware acceleration.

Conventional parallel computers suffer from poor system efficiency when solving the DEM, which means that they give a relatively disappointing speed-up. The DEM has properties that suggest that it may be suitable for acceleration using FPGAs: it exhibits an enormous degree of parallelism, and, as found in the current work, can be processed using short wordlength arithmetic. It is therefore tempting to examine how well this algorithm would map into an FPGA or a number of FPGAs.

The next section will give a brief overview of what FPGAs are, how they have developed since they first appeared, their different types and the current state of the art.

5.3 Field Programmable Gate Arrays (FPGAs)

This section gives an overview on the evolution of programmable logic as well as a brief description of the current technologies.

5.3.1 Evolution

The first commercial Field Programmable Logic (FPL) industry appeared around 1978 when Monolithic Memories introduced the PAL (Programmable Array Logic) architecture. These PALs had a matrix array in which only combinational logic could be implemented, and were used mostly as glue logic (e.g. to decode addresses between memories and

microprocessors). AMD acquired Monolithic Memories in 1984 and the industry came to be dominated solely by AMD and its 22V10 architecture. All the remaining competitors worked on variations of the 22V10 architecture.

In 1984 Xilinx introduced the Look up Table (LUT) based FPGA architecture, which was intended to transform the FPL industry and provide a new direction. Two main reasons allowed Xilinx to successfully compete against AMD:

- CMOS technology was scaling to the point where entire subsystems could be implemented on a single programmable device. PAL architectures do not scale well to larger sizes, whereas FPGAs do.
- Xilinx went for a fabless model; without having to maintain the expensive fabs, it could concentrate only on its designs and their licensing

AMD continued to develop its successful 22V10 architecture into the MACH family of PLDs, but it progressively lost market share until 1999, when it exited the FPL market.
Xilinx and Altera have become the largest and most important FPL vendors since the end of the 80’s

The third wave of FPL devices, which arrived around the millennium, is Complex System on a Chip (CSoC). These combine FPL with microprocessors, memory and fast I/Os on a single die, and allow single chip solution for entire embedded systems.

Figure 5–1 presents a graph of the three FPL waves since 1978, and shows revenues achieved/projected by each wave against time.

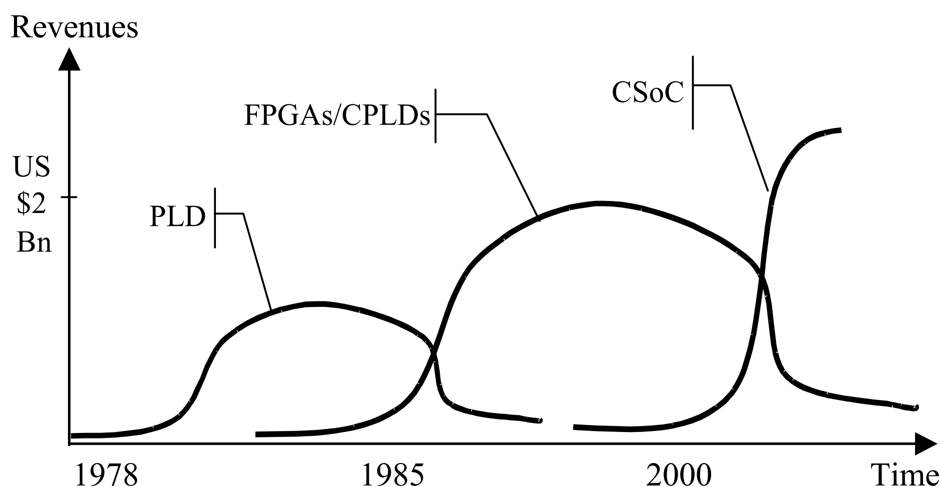


Figure 5–1 *Three FPL waves, PLDs, FPGAs/CPLD and CSoC [2]*

Nowadays CPLDs represent around 35 % of the FPL market, whereas FPGAs represent approximately 53 % of the market [3].

5.3.2 FPGA Technologies

FPGAs can be classified according to many different criteria, e.g. vendors and logic densities, but the most common and basic classification is according to their technology. There are, at the moment, three main programming technologies: anti-fuse, E/EEPROM and SRAM.

1.- ANTI-FUSE based FPGAs are only one-time-programmable. They serve a niche market with applications that have very tight timing constraints, as these FPGAs have the lowest routing delays of all the technologies. Their logic density is far less than the other two technologies, and will have great difficulties reaching equality with them, as these devices cannot migrate to the newest and most advanced CMOS processes.

2.- EPROM based FPGAs are re-programmable, but this must be done outside of the circuit using a programmer, whereas **EEPROM** based FPGAs can be re-programmed in-circuit. One of the main advantages of this technology is that the device does not need to be re-configured after a power down and will always retain its configuration. This avoids the need to use an external PROM that holds the configuration, saving board space. It also avoids the problem experienced by SRAM based FPGAs that the design bit stream can

easily be read by third parties and reverse engineered. The problem with these devices is that the logic density is still smaller than SRAM based FPGAs and that their manufacturing process needs some extra steps compared to standard logic devices such as microprocessors.

3.-SRAM based FPGAs are the most successful technology at present. These devices store their configuration in on-chip latches that in turn control pass transistors to establish the connections. SRAM based FPGAs offer the highest logic capacity and flip-flop count. They can be configured in milliseconds, depending how big the device is, and re-programmed in-circuit an unlimited number of times. The device needs to be reconfigured every time the power is turned off, normally from an on-board PROM. However its major advantage is that it can be easily reconfigured with a new and different program after installation. Another major strength of these type of FPGAs is that their manufacturing process is the standard CMOS process, allowing it to migrate quickly and easily to the most advanced technology available.

The next section will describe the internal architecture of a typical FPGA.

5.3.3 Internal Structure

This section will discuss the internal structure of the typical FPGA, which is shown in Figure 5–2. It consists of a regular matrix of configurable logic blocks (CLBs), surrounded by programmable input/output blocks (IOBs) to connect the package pins with the CLBs (This terminology is only used by Xilinx™; other manufacturers, such as Altera™ (the 2nd largest FPGA vendor)use a different terminology). The CLBs are interconnected by intermediate routing switches. Embedded on-chip RAM is also provided on modern FPGAs. In the case of Xilinx™ each CLB is built of 4 logic cells (LCs). Each logic cell includes one function generator in the form of a 4-input LUT, one storage element and carry logic. Altera™ has a similar approach. Its basic building blocks are called Logic Elements (LE) and consist of one function generator in form of a 4-input LUT, one storage element and carry logic. Ten of these LEs are grouped to form a Logic Array Block (LAB).

The overall functionality of the FPGA is determined by configuration data that establishes

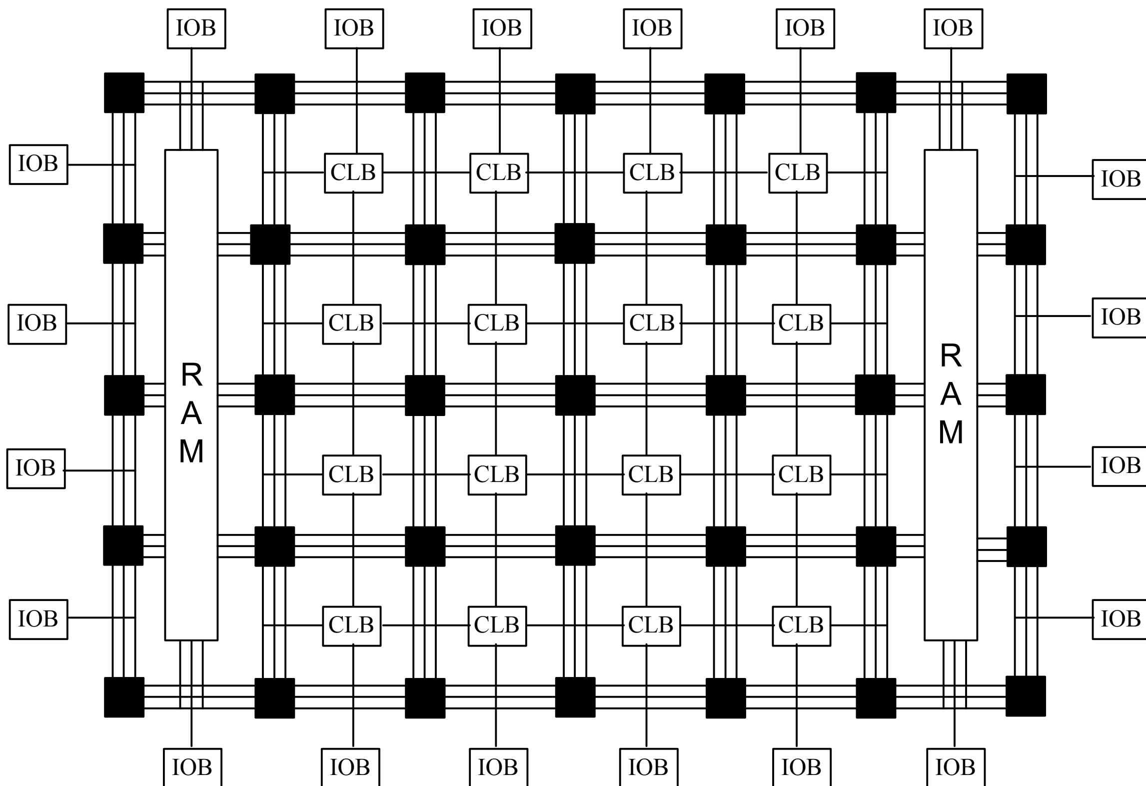


Figure 5–2 *Field Programmable Gate Array (FPGA) internal structure*

the function of each individual CLB, IOB and switchbox. The FPGA is turned into a custom coprocessor for a particular task by downloading the appropriate configuration data into its configuration memory.

5.4 Reconfigurable Computing Platform

The reconfigurable computing platform used in this current work was a PC reconfigurable computing PCI plug-in card. Two cards were used: a Celoxica [4] RC1000-PP PCI card containing a single Xilinx Virtex 1000–6 with 4 banks of 2 Mbytes of RAM and another RC1000-PP board with a Virtex 2000E –6 FPGA also with 4 banks of 2 Mbytes of RAM. Figure 5–3 shows a picture of one of the RC1000-PP boards.

The RC1000-PP hardware platform is a standard PCI bus card equipped with either a XCV1000-6 or a XCV2000E-6 chip. It has 8Mbytes of SRAM directly connected to the

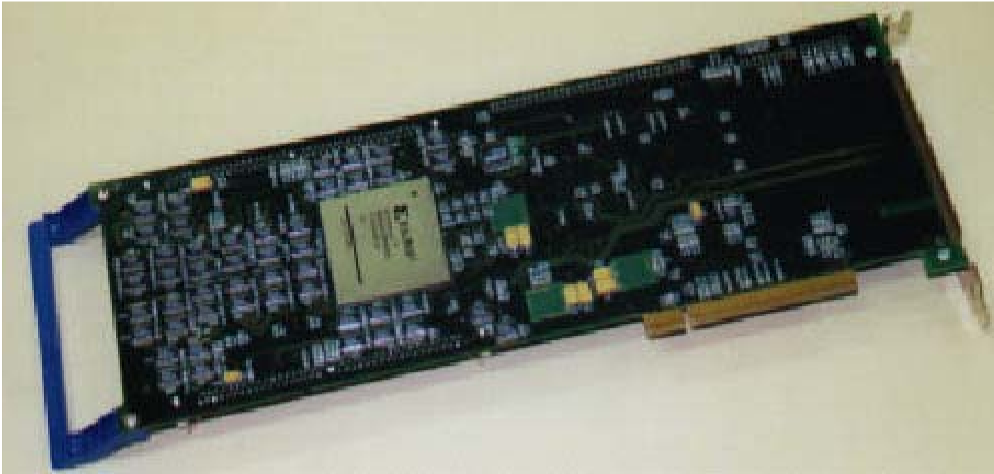


Figure 5–3 *RC100-PP Picture*

FPGA in four 32-bit wide memory banks. The memory is also visible to the host CPU across the PCI bus as if it were normal memory. Each of the 4 banks may be granted to either the host CPU or the FPGA at any one time. Data can therefore be shared between the FPGA and host CPU by placing it in the SRAM on the board. It is then accessible to the FPGA directly and to the host CPU either by DMA transfers across the PCI bus or simply as a virtual address. The board is equipped with two industry standard PMC connectors for directly connecting other processors and I/O devices to the FPGA; a PCI-PCI bridge chip also connects these interfaces to the host PCI bus, thereby protecting the available bandwidth from the PMC to the FPGA from host PCI bus traffic.

A block diagram of the RC1000-PP architecture is shown in Figure 5–4.

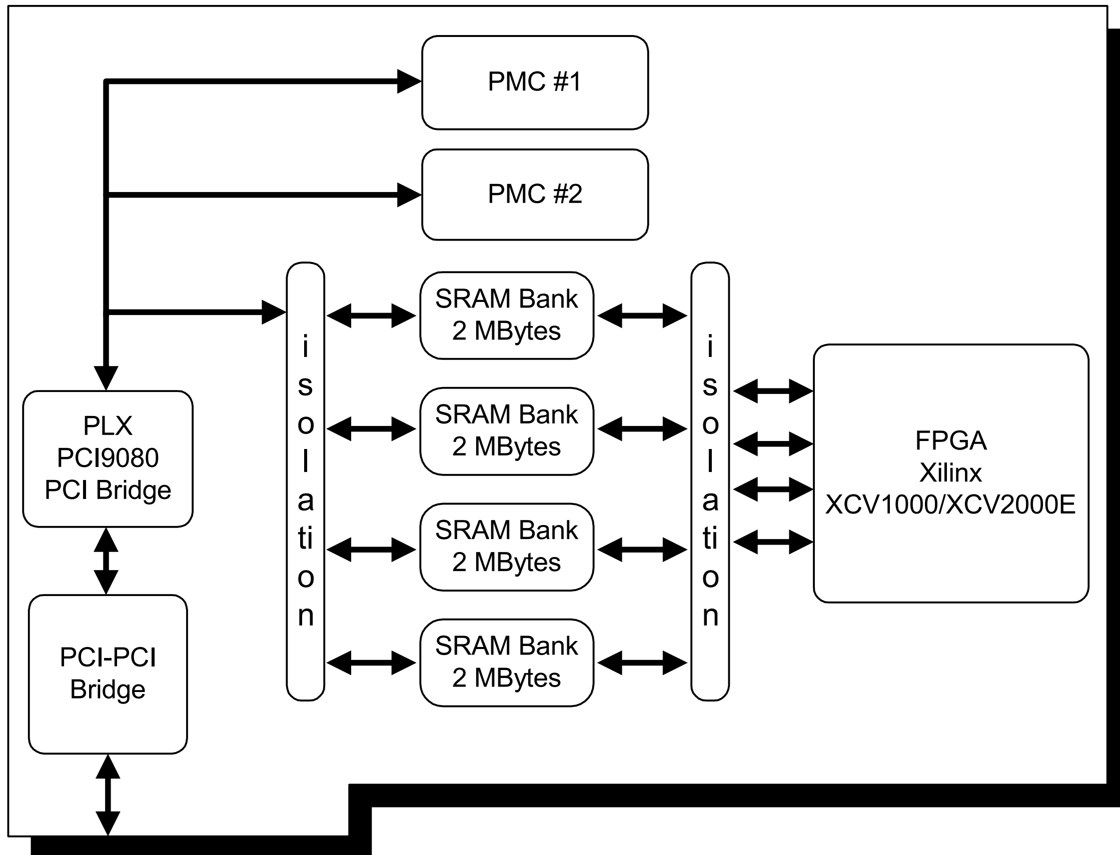


Figure 5–4 *RC100-PP Block Diagram*

5.5 Hardware Implementations

This section will describe the hardware designs implemented on the reconfigurable computing platform just described. The first implementation only makes use of the low level (fine grain) parallelism of the DEM (arithmetic operations) as shown in section 2.7.2.1. The second implementation is a more complex one and makes also use of the high level (coarse grain) parallelism of the DEM by operating on the main tasks: contact check, forces update, position update and reboxing concurrently.

The hardware designs were implemented using VHDL (Very high speed integrated circuit Hardware Description Language) and consists of approximately 10 000 lines of code, plus 6 different IP (intellectual property) cores (e.g. dividers, multipliers, FIFOs)

5.5.1 System Description (Software-Hardware Partition)

As explained in chapter 3, where the software implementation was described, the first thing the software does is to read an initialisation file where the system data is stored. It afterwards generates the requested particles and once it finishes it waits for the simulation to start. This initialisation section is performed by the program for both the software and the hardware implementation. Figure 5–5 shows a system layout.

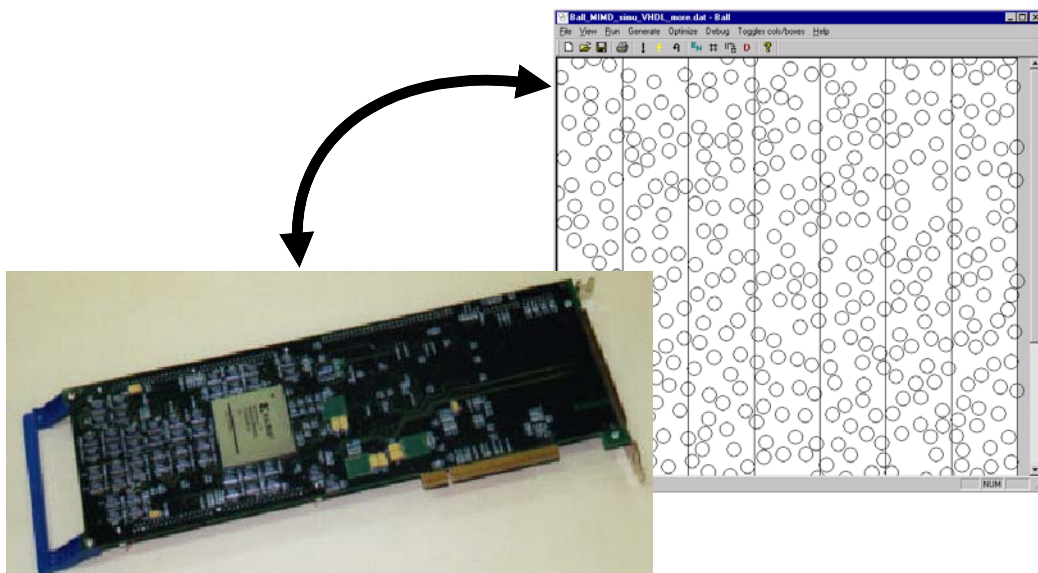


Figure 5–5 *System layout*

Once the particles have been generated by the software program, the user has the option to choose to run the simulation in software or in hardware. Figure 5–6 shows the window that displays the various options available to the user. The simulation can be run in *software,* i.e. on the PC's microprocessor, or in *hardware* i.e. on the reconfigurable computing PCI board. If it is decided to go for the hardware simulation, the user also needs to select which of the two available hardware configurations should be used. Either the low level parallelism or the high and low level parallelism implementation configuration can be chosen. In either case the user also needs to select the appropriate FPGA configuration file (the *bit* file), the RC1000-PP board to be used (in case there is more than one board installed in the same PC) and the clock rate at which the FPGA should be operated. For the high and low level parallelism implementation, the user can also select how many contact check units should be used. (As will be described in more detail in later sections, the

hardware implementation can instantiate more than one of these units to work in parallel.)

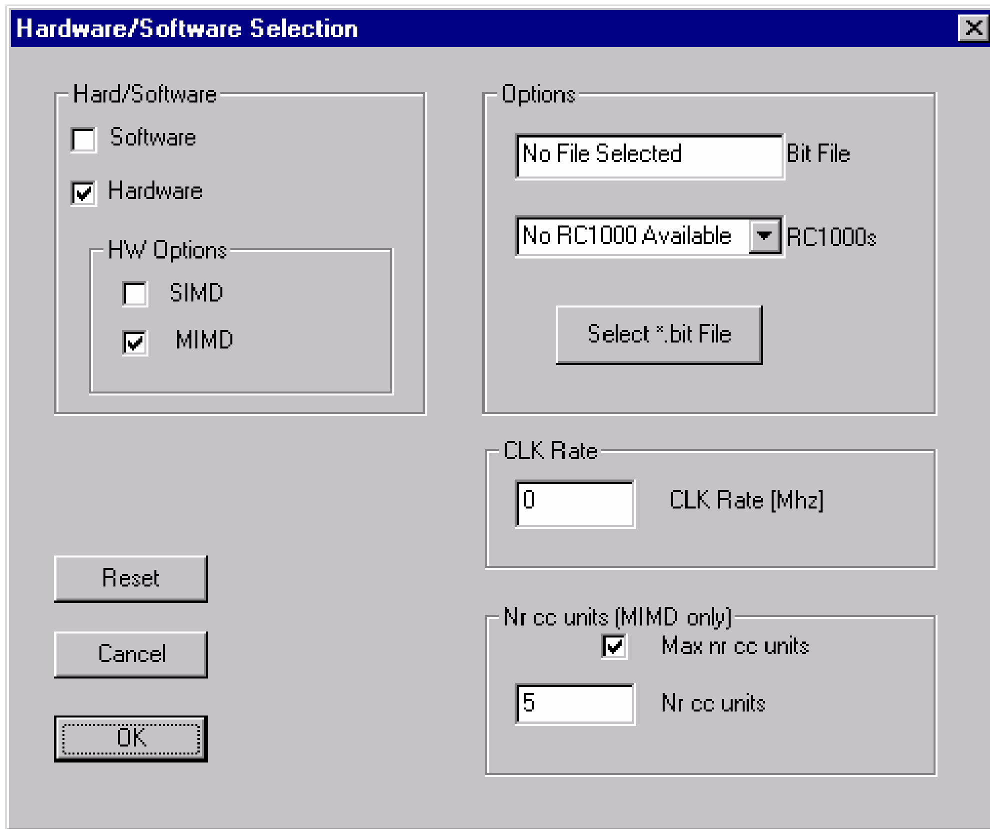


Figure 5–6 *Hardware Software selection*

By default, the maximum number of contact check units is selected, but this can be varied if the user wishes to perform performance measurements on the system.

5.6 Data Format

The DEM can be processed using short wordlength arithmetic, so that it uses simple arithmetic operations in a massively parallel way on large data sets. By proper scaling of the problems, very large and very small numerical values can be avoided, so a large dynamic range is not needed. The main features of interest for most discrete element simulation is the bulk behaviour of the system (not the detailed behaviour of individual particles), which also reduces the need for high precision. It was therefore decided to use fixed-point arithmetic instead of floating point, as this maps much better on the FPGAs, and needs far less resources. In order to make the design fit onto the available FPGAs, a 16 bit data format was chosen. When an FPGA with more resources becomes available, the

design can be easily modified to use a 24 bit or even a 32-bit data format as the design is implemented with the bit width as a generic parameter.

Computer hardware usually represents negative numbers in fixed-point arithmetic in one of three different ways: Sign and magnitude, one's complement or two's complement. Two's complement was chosen in this work, as this is the most widely used and convenient representation, and also because this representation was supported by the pre-designed hardware cores used in the design.

Within the 16 bit data format used, 4-bits were used to represent the fractional part as 2^{-4} = 0.0625 gives sufficient precision for the DEM and the twelve remaining bits were used for the integer part, which means that a maximum number of + 2047/-2048 can be represented, as shown in Figure 5–7.

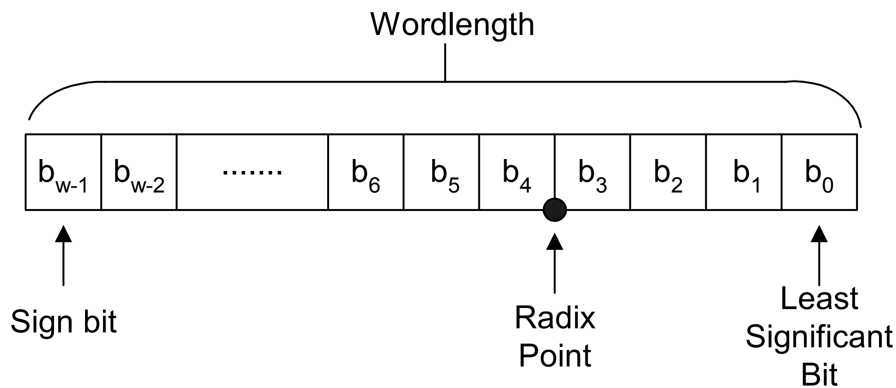


Figure 5–7 *Data format*

The data format is anyway adjusted to the nature of the operations taken place in order to safe as most possible HW resources (e.g. if a parameter is always smaller than 1 only four bits are allocated to it instead of the 16 bits)Less than 4 bits for the fractional part could not be allocated, because the data format needs to be able to represent the time step, which as said in chapter 2 is of the order of milliseconds. On the other hand more than 4 bits for the fractional part would mean that the maximum value representable with this data format would be less than 2048, which would make the stiffness so small that particles would transition over neighbouring particles, allowing only the simulation of very *soft* particle assemblies.

5.7 Implementation classifications

Two implementations will be described in this chapter. The first one is a simple implementation employed to obtain some preliminary results on how well the implementation of the DEM would fit onto an FPGA. The second is a more complex one. If the major tasks i.e. contact check, forces update and positions update are considered to be machine instructions, the first implementation can viewed as a Single Instruction Multiple Data (SIMD) design, as only one of these instructions is active at one time. The second implementation can be viewed as a Multiple Instruction Multiple Data (MIMD) design, as the three tasks, as well as the reboxing of particles transitioning from one subdomain to another, are all performed in parallel. The SIMD design only makes use of the low level parallelism of the arithmetic operations; the MIMD design also exploits the high level task parallelism. This classification is strictly speaking wrong as it would mean that all the processing units are exactly the same, which is not the case here. It is therefore more correct to call them low and high-level parallelism implementations, because they make use of only the low level parallelism and afterwards of the low and high-level parallelism of the algorithm. These two implementations are described in detail in the next two sections.

5.8 Low level Parallelism Implementation

This first design was implemented on a Celoxica RC1000 board containing a single Xilinx Virtex V1000-6 FPGA.

Figure 5–8 shows a block diagram of the hardware implementation. It consists of six main units:

1. A *contact check unit,* which identifies the particles in contact.
2. A *force update unit*, which updates the interparticle forces.
3. A *movement update unit*, which calculates the particles' new velocities and coordinates.
4. A *control unit*, which synchronizes all the units and generates all the control and address signals.
5. An interface unit to read and write data to and from the external memory

6. A write back unit to write the results of the arithmetic units back to the internal FPGA memory.

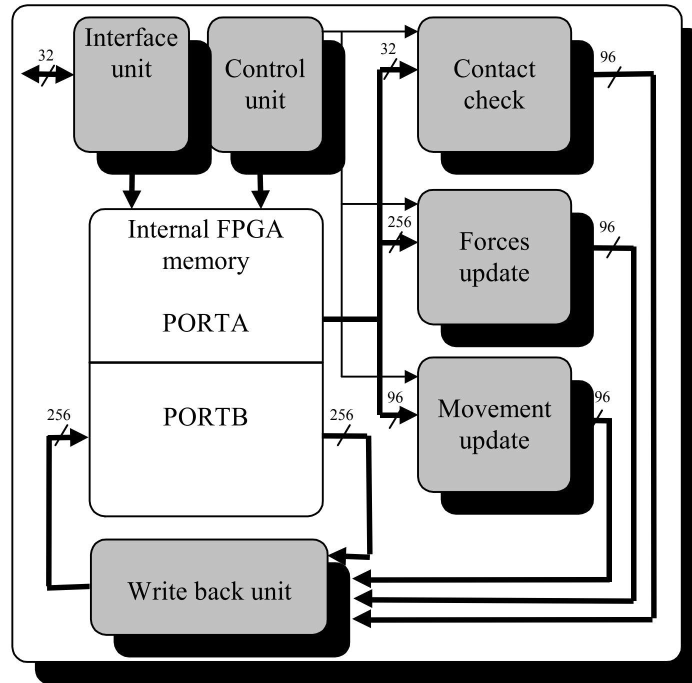


Figure 5–8 *Low level parallelism FPGA Implementation block diagram*

The block RAM of the FPGA is used to hold the data required to describe each particle This includes position, velocity, angular momentum, identity of neighbours, and the force that it is experiencing. Data is read from and written to the internal FPGA memory at a clock speed four times greater than that of the forces update units in order to keep its pipelines fully loaded (on each clock cycle of the force unit, it needs to read and write data of two particles simultaneously).

5.8.1 Detailed Unit Descriptions

Each unit of this implementation will be described in detail in this sub-section.

5.8.1.1 Control Unit

The control unit generates the necessary control signals to synchronise data between blocks, and to steer the data output from the RAMs to the inputs of the appropriate computation unit. The control unit also generates the addresses to read and write data from and back to the internal and external memory

5.8.1.2 Contact check

For each particle, a "contact list" is formed, which contains references to each of the particles with which it makes contact. In order to detect if two particles are in contact the following equation has to be solved:

$$\Delta n = R_1 + R_2 - \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \geq 0 \qquad \textbf{\textit{Eq. 5–1}}$$

Where x_i y_i are the co-ordinates of each particle's centre and R_1 and R_2 are the respective radii (see Figure 5–9); the repulsive force between the particles is directly proportional to this overlap. If the condition of Eq. 5–1 is true, the addresses of the two particles are added

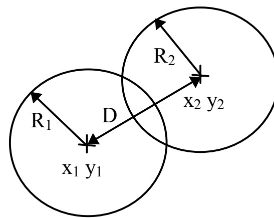


Figure 5–9 *Balls in contact*

to each others' adjacency list. For this investigation, all particles are assumed to have the same radius R. Under this circumstance, simple geometry (section 2.6.2) shows that for a 2-D simulation, the maximum number of contacts that each ball can have is 6. This means that contact information can be represented by a very simple data structure, in which each particle has six memory slots allocated to hold the identities of the particles potentially in contact.

If there are N particles within a region of the DEM, then the number of contact checks that must be performed is N^2. The square roots and multiplications used in Eq. 5–1 are very expensive to perform in FPGA hardware, with the implication that a full contact check would be prohibitively expensive.

Instead of checking for true contacts, it was decided to check which particles are within each others' bounding boxes, and this acts as a filter before the actual contact check. Under some circumstances (see Figure 5–10), this means that a pair of particles will be classified

as neighbours even though they are not truly in contact. This causes no real problem, since it is detected and correctly handled by the force increment unit.

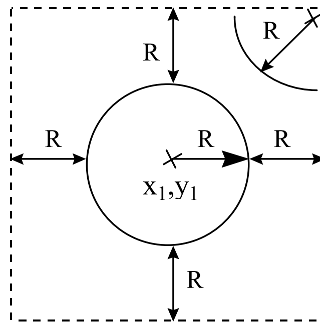


Figure 5–10 *Neighbour check model*

Using the bounding box method to perform contact checking makes this unit very cheap in terms of hardware resources, as it requires only 2 additions, 2 subtractions and 4 comparisons.

5.8.1.3 Inter-particle forces increment

Once the contact list for a particle has been established, the total force acting on it can be determined. This will require a full solution of Eq. 5–1 for each contact identified, but this will be needed to be performed only a maximum of 6N times.

For every contact identified between two particles, the resulting force is calculated. For this study, a simple force-displacement law is adopted: the resulting force between two balls is directly proportional to the indentation between the balls, as shown in chapter 2 where the DEM was described in detail.

The resultant force on a particle is the vector sum of the forces caused by each contact with its neighbours. The force update unit, which does require the computation of the terms in Eq. 5–1, requires a large amount of hardware. It also operates at a comparatively low clock speed of 7.5 MHz in contrast to the contact check unit which works at the full system clock speed of 30 MHz. Figure 5–11 shows the internal structure of this unit, where each column represents one pipeline stage.

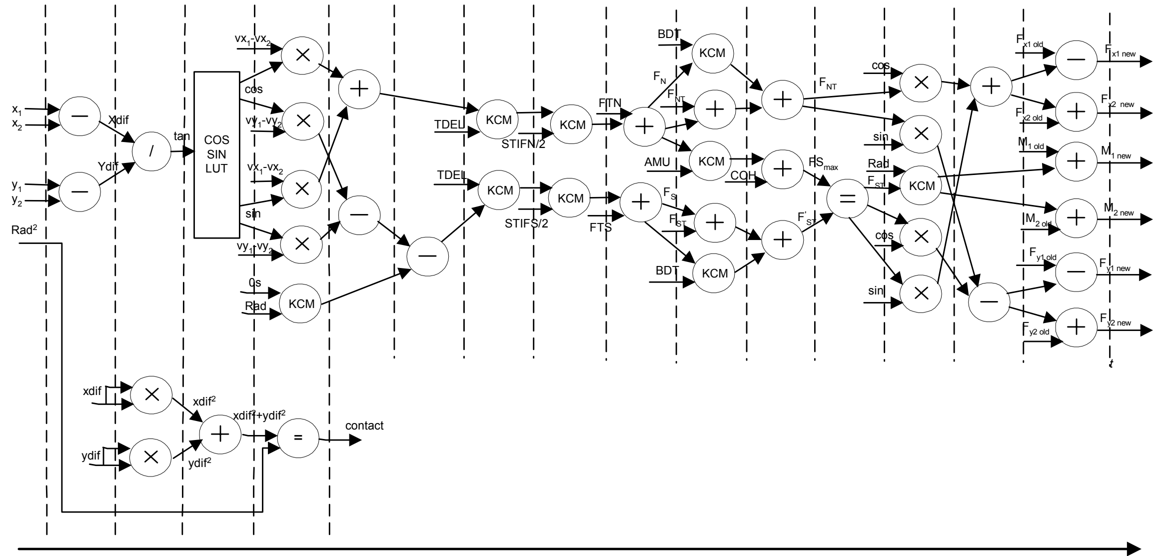


Figure 5–11 *Forces update unit internal structure*

It can be seen that there are three main paths in this structure. One that calculates the forces in the x direction, another that calculates the forces in the y direction, and another shorter path, which computes the terms in Eq. 5–1 (without doing the square root, which is expensive but unnecessary), to check if the particles are in contact.

In order to compute the x and y components of the force between two particles, it is necessary to compute the sine and cosine of the angle α of the line connecting the two particles' centroids, (see Eq. 5–2, Eq. 5–3 and Eq. 5–4).

$$\sin(\theta) = \frac{y_2 - y_1}{d} \qquad \textbf{Eq. 5–2}$$

$$\cos(\theta) = \frac{x_2 - x_1}{d} \qquad \textbf{Eq. 5–3}$$

$$with \quad d = \sqrt{(x_2 - x_1) + (y_2 - y_1)} \qquad \textbf{Eq. 5–4}$$

This is done using a Look Up Table (LUT) (in order to avoid using a square root, which is very expensive in terms of hardware resources). LUTs can be implemented easily in FPGAs, and take up block RAM rather than the logic resources that would be required by a square rooter. Predefined values of the cosine and sine are stored in this table.

5.8.1.4 Velocity and Position Update

Once the resultant force on each ball has been calculated, these forces are used to find new accelerations using Newton's second law. In this study, it is assumed that the masses of all the balls are identical. These accelerations are integrated to obtain the velocities in the x and y direction and the angular velocity

The new coordinates can be found by adding the original coordinates to the incremental displacement obtained by integrating the calculated velocities. The position update unit has an intermediate level of hardware complexity, and operates at the same speed as the force update unit (7.5 MHz), which is 4-times slower than the system clock, in order to have its pipeline fully loaded, achieving one new result per clock cycle.

It consists of three pipelines in parallel (see Figure 5–12). The first computes x and v_x, the second computes y and v_y and the third computes θ and $\dot{\theta}$.

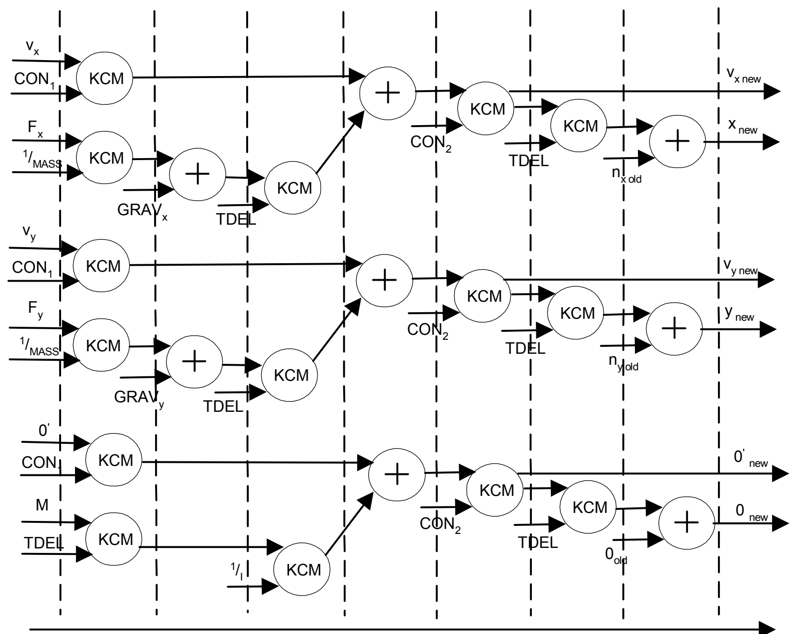


Figure 5–12 *Velocity and Position update unit internal structure*

5.8.1.5 Write back unit

The purpose of this unit is to merge the data for each particle that emerges from the arithmetic units. Each particle is represented by a 256-bit word, but only certain bits of this word are updated by each of the different units. For example, if a new contact list is generated, only the memory locations of the old contact list are overwritten, and the rest of the old data is preserved.

5.8.1.6 Interface Unit

The interface unit reads and writes data from and to the FPGA's internal memory when instructed to do so by the control unit in the low level parallelism implementation, it is only used twice in each analysis. Once at the beginning, it is used to read in the new data, and once at the end when the calculations have finished in order to write the data back to the external memory.

5.8.2 Hardware requirements

The hardware requirements for each of the main functional units are shown in Table 5–1. Constant coefficient multipliers (KCMs) require much less hardware resource than multipliers that allow both inputs to vary. Having balls of the same radius facilitates the widespread use of KCMs.

Table 5–1 *Hardware requirements for the low level parallelism units*

CONTACT CHECKING	FORCE UPDATE	MOVEMENT UPDATE
2 adders 2 subtractions	23 adders 10 multipliers 8 KCMs 1 dividers 1 Look Up Table (LUT)	8 adders 15 KCMs

The contact checking unit is very simple, requiring little hardware resources, and capable of operation at high clock speeds. The force update unit, which does require the computation of the terms in Figure 5–11, requires a large amount of hardware. The movement update unit has an intermediate level of hardware complexity.

5.8.3 Memory Map

Once it is decided to run the simulation in hardware, and the simulation has started, the program formats the data needed by the FPGA, as shown in Figure 5–7, and downloads it to the RC1000-PP board's memory. The data format is shown in Figure 5–13.

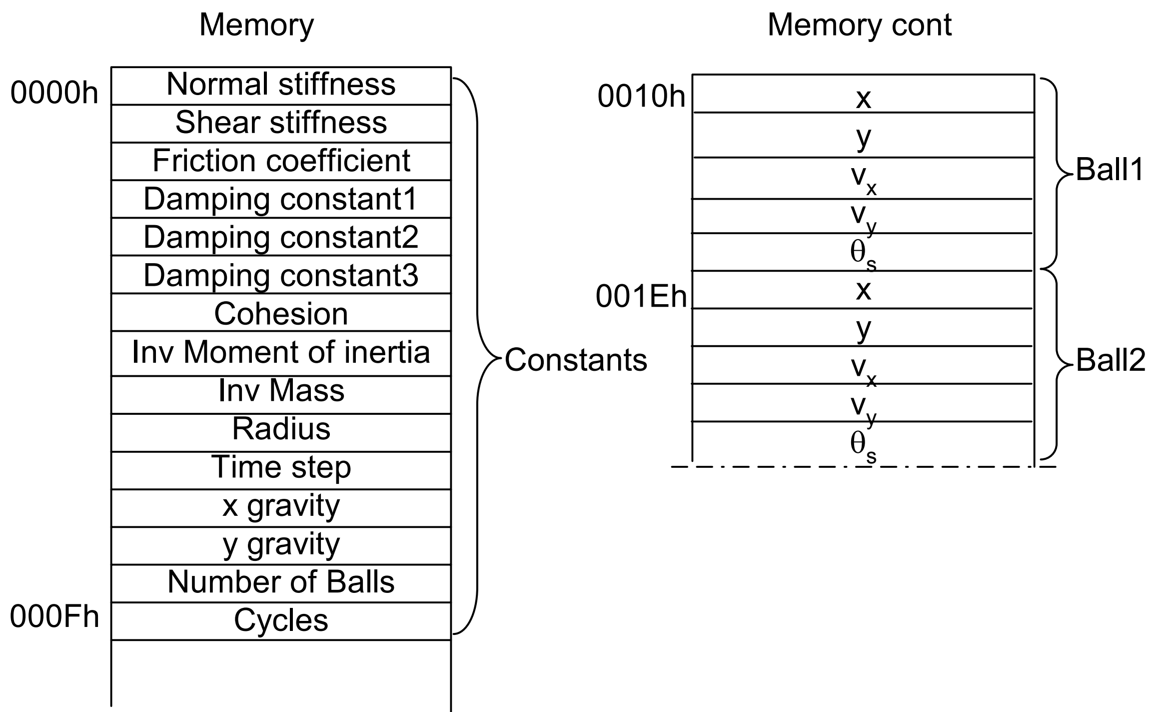


Figure 5–13 *Memory map for the low level parallelism implementation*

The board has four banks of memory of 512 Kbytes x 32 bits. Although each data element takes up only 16 bits, it was decided to store only one element per memory location so that in future 24-bit or 32-bit arithmetic can be used without major changes to the addressing of the control unit.

The first 16 memory locations are filled with the parameters/constants needed by the FPGA, which have either been pre-computed by the software or are constants. After that, only ball data is stored. The only data items that are needed by the FPGA to describe a ball are the x and y coordinates, the velocities in the x, y direction, and the rotational velocity.

Instead of passing the mass and the moment of inertia of the particles to the FPGA, it was decided to pass the inverse of these, as this would allow using a KCM instead of a more expensive divider.

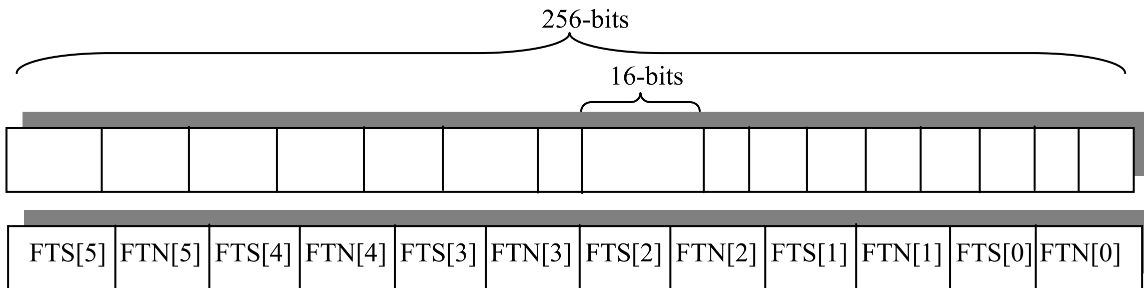


Figure 5–14 *FPGA's internal memory map*

Once the data has been downloaded onto the FPGA board, a ready signal is generated by the software program in order to wake the FPGA up and start the simulation. The first thing the FPGA does is to read in all the data stored in the external memory and store it in its internal memory in a new format shown in Figure 5–14. This format uses two words of 256 bits (16 x 16 bits) to represent each ball. Each 256 bit word contains 16-bit representations for the particle's position and angular co-ordinates x, y, θ, the velocities v_x, v_y, $\dot{\theta}$, the forces and moment F_x ,F_y, M, a type flag for the particles, and the identities of up to 6 neighbouring particles that have been identified during the contact check. For every contact, the normal and shear force needs to be stored as well, since the current DEM algorithm only calculates force *increments* at each time step; so an additional 12 items need to be stored in the internal FPGA memory. The use of incremental forces is to improve the accuracy of the calculations using limited precision.

5.8.4 Timing considerations

The number of clock cycles required in order to stream the data corresponding to N particles through each of the computations units is shown in Eq. 5–5, Eq. 5–6 and Eq. 5–7.

$$t(cc) = \frac{\frac{1}{2}N^2}{4} \qquad \textbf{\textit{Eq. 5–5}}$$

$$t(forces) = 6N \qquad \textbf{\textit{Eq. 5–6}}$$

$$t(position) = N \qquad \textbf{\textit{Eq. 5–7}}$$

Note that the contact check unit dominates the timing of the system for any realistic size of N (the number of balls in the analysis), due to its quadratic dependence on N. The total time for contact checking is divided by a factor of four as this unit is clocked at a 4 times faster rate than the forces and position update units, since its hardware is very simple. The loading and unloading time of the pipelines are not considered here; the number of particles is sufficiently large that this effect can be neglected.

5.8.5 Implementation Drawbacks

This initial implementation contains several significant inefficiencies. In particular, contact checking, force updating and position updating cannot be overlapped; since only one unit can be active at a time. This is because the force unit has to wait until the contact data list has been built before it can start work. This means that all the particle data held in the block RAM must be streamed through the contact check unit, and written back to RAM before the force unit can operate. Similarly, the movement update unit must wait for all data to be streamed through the force update unit before it can begin.

Another disadvantage of this simple implementation is that the number of particles that can be processed continuously is limited by the capacity of the block RAM. When a new frame of data needs to be paged from external RAM into the FPGA’s block RAM, all the processing of data must stall. This means that the largest number of particles that can be processed at full speed is 500 for a Virtex XCV1000 FPGA. This is a too small number of particles to simulate practical problems.

5.9 High and Low Level Parallelism Implementation

A new design was implemented in order to tackle the inefficiencies identified and described in the previous section. This new implementation differs from the previous design in the following major ways:

- All units can work simultaneously
- Multiple copies of each unit can be instantiated

- Paging data in and out of the external memory overlaps with computation, allowing an extremely large number of particles to be treated continuously and efficiently

The arithmetic units (forces update, position update and contact check) are identical to the ones described in the previous section.

In order to allow the computational units to operate in parallel, the domain is decomposed into k vertical columnar sub-domains, as shown in Figure 5–15. Each particle belongs to a particular cell, and for most particles contact checking and force updating need only be performed against the other particles within the same cell. For the small number of particles that are close to the boundary between two cells, more complicated arrangements are necessary.

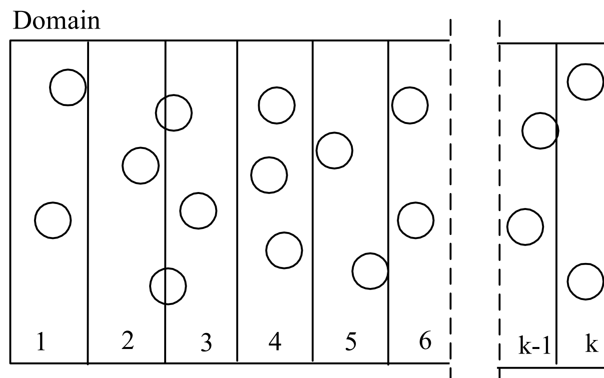


Figure 5–15 *Domain decomposition*

The hardware architecture used to process the domain is shown in Figure 5–16. As this implementation needs more hardware resources than the previous design, it was implemented on the RC1000-PP board equipped with the XCV 2000E device.

The architecture divides the internal block RAM of the FPGA into six dual port RAMs. At any given time, six of the columnar cells shown in Figure 5–15 are stored within the FPGA and undergo processing. The RAM contains two 256 bit entries for each particle within that cell consisting of 16 bit entries for x, y, θ, v_x, v_y, θ’, F_x ,F_y, M, a type flag, and the reference of up to 6 neighbouring particles and another to hold the normal and shear forces for every contact. This is the same as the first implementation.

The control units generate the necessary control signals to synchronise data between the blocks, and to steer the data output from the RAMs through the switch array to the inputs of the appropriate computation unit. The control units also generate the addresses to read and write data back to the internal and external memory.

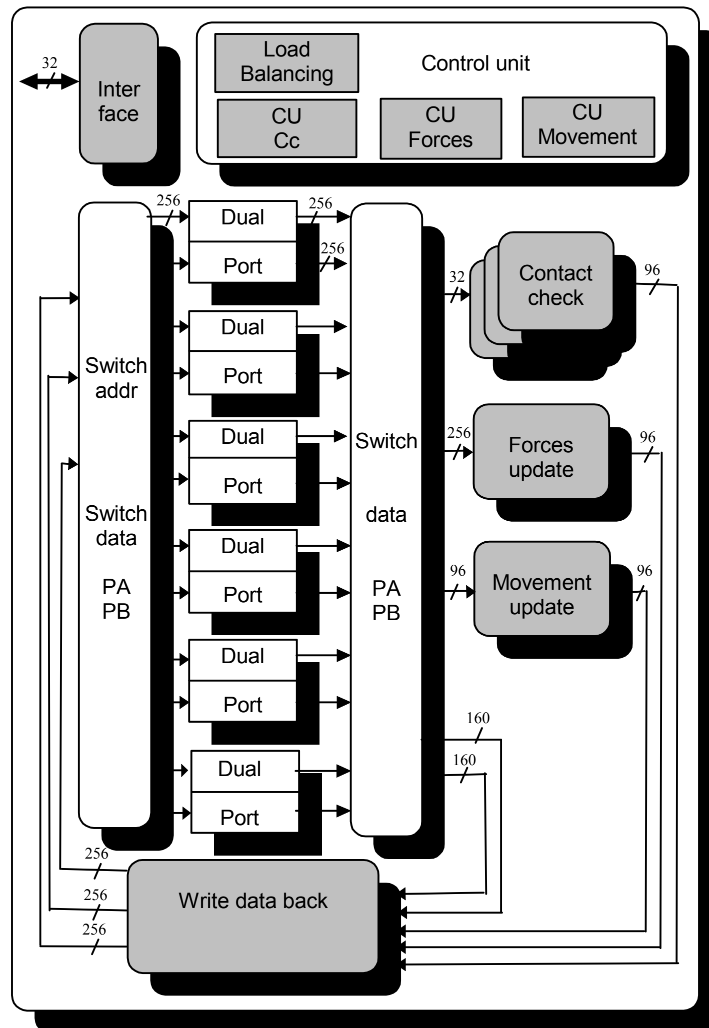


***Figure 5–16** High and low level FPGA Implementation block diagram*

As an example of the scheduling, consider the situation where the six dual port block RAMs of Figure 5–16 respectively contain the particle data for columns 1,2,3,4,5 and 6 of the domain of Figure 5–15. The particle x, y co-ordinate data for column 5 is streamed through the contact check unit and the particle contact list data is written back into the block RAM. At the same time, the data for column 2 is streamed through the force update unit, and the data for column 1 is streamed through the co-ordinate update unit. The results are written back into the appropriate region of the FPGA's block RAM. The data for

column 1 is then written into an external RAM, and new data for column 7 is read from external RAM.

Figure 5–17 shows how the computation progresses. During each epoch, the contents of one block RAM (corresponding to all the particles in one columnar cell) are streamed

Epoch 1

Columns held in block RAM	1,2,3,4,5,6
Column undergoing contact check	5
Column undergoing force update	2
Column undergoing co-ordinate update	1

Epoch 2

Columns held in block RAM	7,2,3,4,5,6
Column undergoing contact check	6
Column undergoing force update	3
Column undergoing co-ordinate update	2

Epoch 3

Columns held in block RAM	7,8,3,4,5,6
Column undergoing contact check	7
Column undergoing force update	4
Column undergoing co-ordinate update	3

And so on…

Figure 5–17 *Scheduling of the computation*

through one of the computation units. It can be seen from Figure 5–17 that for each column, first a contact check will be performed, then a force update, then a movement update in that order.

Due to complexities associated with handling particles close to the cell boundaries, the contact check unit may have to update the columns to the left and the right of the column

that is currently undergoing contact check, as the contact check unit also deals with particles that have transitioned from one column to another. It deletes the particles, which have moved from the column where the contact check has taken place and moves them either to the right or left column, depending where the particle has moved. Also, the force update unit may have to interact with the column to the right of the column currently undergoing force update, as a particle in this column might be in contact with particles in the neighbouring column. That is why the FPGA must hold six columns at any given time, rather than three. This is explained in more detail in section 5.9.1

5.9.1 Handling Cell Boundaries

Several complications arise as a result of interactions at boundaries between the columns of particles.

5.9.1.1 Performing Contact Check with Particles at the Neighbouring Sub-Domains

Firstly, a particle close the boundary may be in contact not only with particles from its own column, but also from an adjacent column. This situation is handled by an auxiliary memory located within the control unit that handles inter-cell boundaries. So for example, in epoch 1 of Figure 5–17, during the contact check of column 5, each particle of column 5 is checked to determine whether it is within 2R of the boundary with column 6 (2R, because it is the maximum distance at which a certain particle in one sub-domain can be in contact with another of the neighbouring sub-domain). If this is true, then after this particular particle has completed the contact check, its data is written back as normal into the block RAM for column 5 but it is *also* copied into the auxiliary memory within the control unit. In epoch 2, when column 6 is checked, each particle within column 6 is checked not only for contact with the other particles of column 6 but also with each particle stored in the auxiliary memory that contains the boundary data for column 5. Once the correct contact list has been generated for a particle, all subsequent computation will proceed correctly, even if a contact straddles a boundary.

5.9.1.2 Transition of Particles from one Sub-Domain to another

Secondly, a particle close to a boundary may transition from one column to another during coordinate update. For such a particle, after the results of the co-ordinate update are written

back to the RAM, the particle would have the correct coordinates, but its data would have been stored in the wrong block of RAM.

In order to illustrate how this case is handled, consider epoch 2 of Figure 5–17 and imagine that during the previous time step a particle had transitioned from column 6 to column 7, but had been written back to the RAM block corresponding to column 6. In epoch 2, the contact check unit examines every particle held within the block of RAM corresponding to column 6. Its control unit knows the boundaries of this column, and is capable of detecting that a particle ought to be in column 7. When it finds such a particle, it does not process the particle further, but simply writes it back the memory corresponding to column 7. In epoch 3, when column 7 is processed, the transited particle will be treated correctly.

The situation is more complicated if at the previous time step a particle had transitioned from column 6 to column 5, but had been written back to the RAM belonging to column 6. When such a particle is detected, it is checked against all particles in column 6, and against the auxiliary boundary list that had been constructed for column 5 during epoch 1. The data for this particle is then written back into column 5. This procedure works because it is a requirement of the DEM that the time step is sufficiently small that no particle can move through a distance greater than its own radius within one time step. This means a full check against all particles within column 5 is unnecessary; and a check against the boundary list of column 5 will suffice.

It can now be seen why it is necessary to leave a two-column separation between contact checking and force update (e.g. columns 3 and 4 of epoch 1 of Figure 5–17). Due to the boundary effects, the contact check process of column 5 can update the data in column 4, whilst force update of column 2 may make use of data in column 3.

5.9.1.3 Adaptive Cell Boundaries

A third complication is that as simulation progresses, particles will move between columns, and some columns may become heavily populated, whilst others are sparsely populated. It is then necessary to move the cell boundaries, thus expanding some cells and contracting others. This is needed in order to provide good load balancing, and also to prevent overflow of the block RAMs. Figure 5–18 shows an example of the adaptation of

cell boundaries to avoid having more particles than the FPGA can hold in one sub-domain, thus losing these particles.

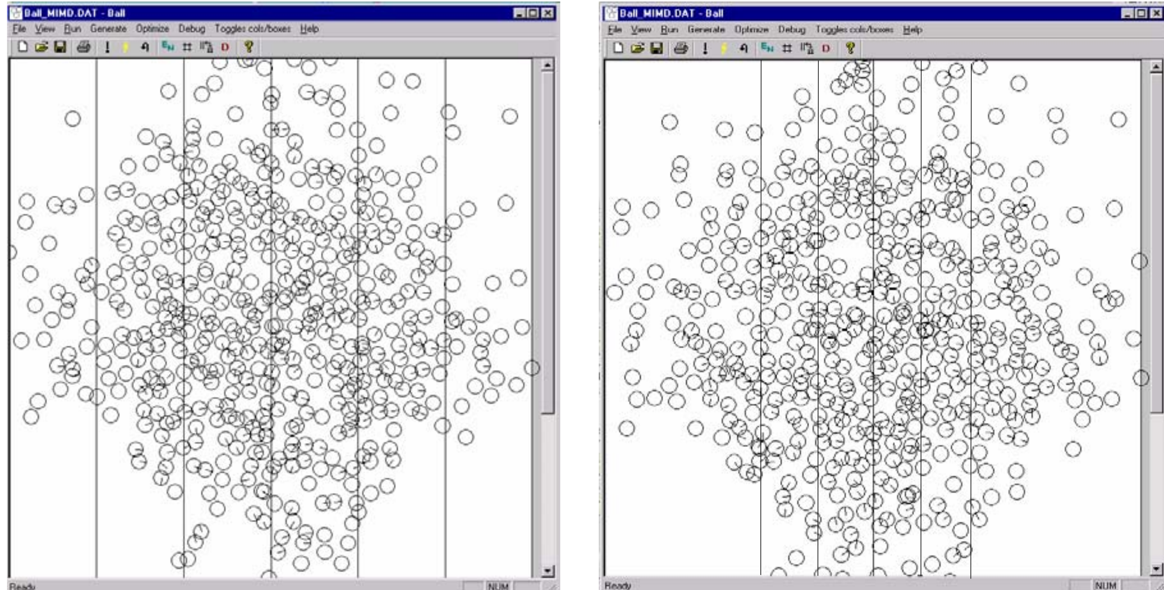


Figure 5–18 *Simulation example of the adaptive cell boundaries*

Movement of cell boundaries is fairly simple. The control unit monitors how many particles are held in each block RAM. When the number falls below a minimum threshold or rises above a maximum, the boundary is moved by a distance R so as to expand or contract the cell. When the boundary moves, a number of cells will find that their data is stored in the wrong column of RAM, but this will be automatically detected and corrected by the mechanisms described earlier for handling particles close to boundaries.

Using the procedures described above, the transition of particles from one cell to another is handled without causing any loss of performance. Also, the cell size is adaptively optimised so that good load balancing is always achieved.

The XCV2000E can only hold a maximum of 128 particles per column, because part of the FPGA’s embedded RAM has to be allocated to other functions to make the design fit. If a sub-domain were to have more than 128 particles, it would *lose* the excess particles. This can be avoided by dynamically balancing the load in each sub-domain so that no more than a certain maximum number (always smaller than 128) will be in each sub-domain. The

software program will also issue a warning signal if there is a danger of having more than 128 particles in a sub-domain. This may happen, for example, when the domain height compared to the balls radii is so large that more than 128 balls would fit in one column.

5.9.2 Memory Map

The memory map is different from the previous implementation, as data is read from and written to the external memory after each column has been treated. Therefore the total normal and shear force of every contact needs to be stored to the external memory. Every column data has a header with four elements.

- The position of that column (lower x coordinate)
- The current number of balls in that column
- The first address where this data is stored in the external memory
- The last address where this data is stored in the external memory

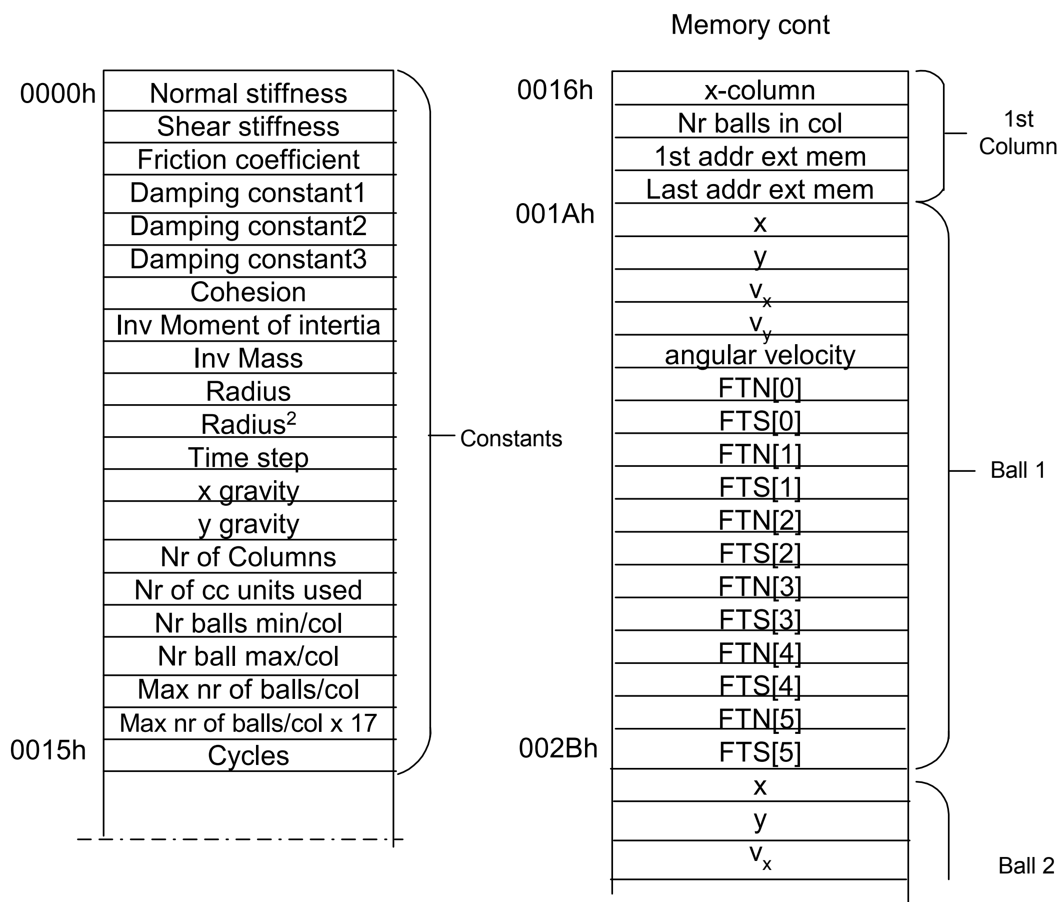


Figure 5–19 *Memory map for the high and low level parallelism implementation*

Some extra system parameters also are needed, e.g. the total number of columns, the maximum number of particles that the internal FPGA memory can store without overflow, and the maximum and minimum number of particles allowed in a column before the system re-adjusts the boundaries of its columns to balance the load.

Where possible, parameters (for instance, the square of the radius) are pre-calculated by the simulator and passed to the hardware.

5.9.3 Load Balancing

With contact checking, force updating and co-ordinate updating being performed in parallel, load balancing problems will inevitably appear, since the overall system speed will be limited by the speed of the slowest of the three units. As shown in chapter 2, where the DEM was described in detail**,** the coordinate check is the most time consuming, but requires very simple hardware and can operate at high clock speed.

In order to improve the load balance, several contact check units are instantiated, and operate in parallel. The number of contact check units to be used is a parameter of the design, which can be easily changed. The contact check control unit can generate all the required control signals to steer the data correctly between the different check units. Lastly, the contact check units can run at four times the clock speed (30 MHz) of the force update unit and co-ordinate update unit (7.5 MHz), because it is very simple, requiring little hardware resource.

It can also be seen from section 2.6 that the coordinate update unit will finish much earlier than the forces update unit. The spare time available at the end of the coordinate update is used to write the data from the block of RAM corresponding to co-ordinate update that has now been finished being processed for this time step, into external RAM. A new set of data is also read from external RAM, which corresponds to the next column of the domain that is to be processed. The ideal timing schedule is show in Figure 5–20.

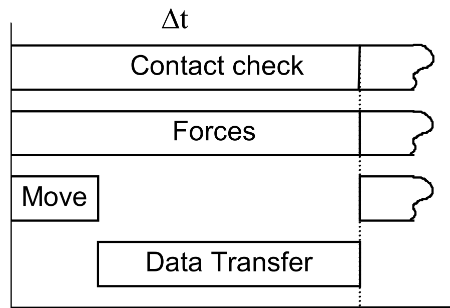


Figure 5–20 *High and low level parallelism scheduling*

In this way, writing to and reading from external RAM can be fully overlapped with computation, and the number of particles that can be processed at full speed is limited only by the size of the external RAM. This means that problems containing tens of millions of particles can be processed easily.

In order to have the system running at its maximal efficiency, there must be as many contact check units as needed to make the time for position update and data update (t(position)+t(interface)) equal to the time for contact checking t(cc). Figure 5–21

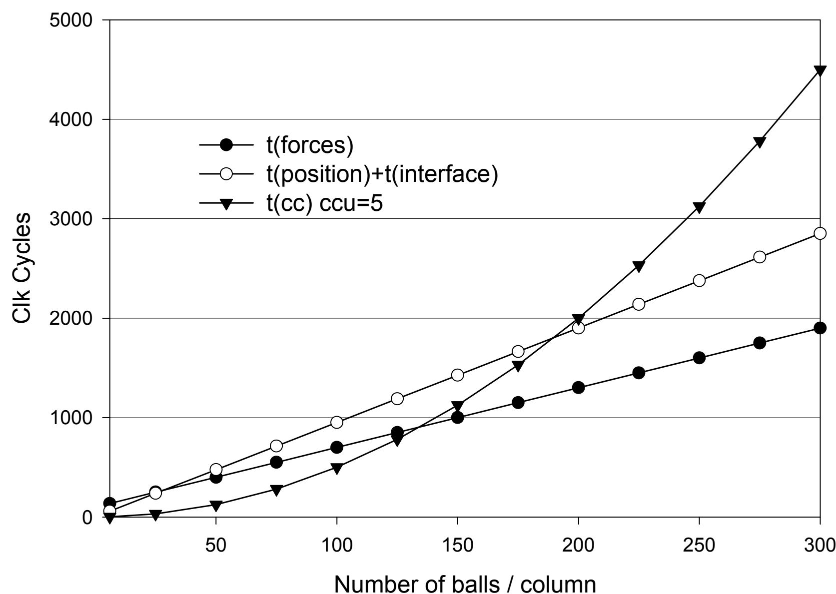


Figure 5–21 *Time needed for each task*

illustrates the number of contact check units needed, using theoretical calculations, for the above condition to be true. The straight line for t(position) + t(interface) is the number of clock cycles required to perform position update plus the time needed to write and read new data to and from the external memory, for all particles in one column. Where the t(cc) curve intersects with the t(cords + interface) line, this indicates the ideal load balance for that number of particles. So, for example, a simulation of 175 particles/column has almost ideal theoretical load balancing when 5 contact check units are instantiated.

The time needed to perform the contact check depends not only on the number of particles per column, but also on the domain topology and the size of the particles. The larger the height of the domain Y is, the larger the contact area is. Also the smaller the radius of the particles is, the more particles need to be checked for contacts with the neighbouring column. Figure 5–21 is given for a particular value of $^{Y}/_{d}$ with Y being domain height and d the balls' diameter.

5.9.4 Timing considerations

The previous section described the ideal case in which all main tasks finished at the same time, achieving a perfect load balance. Unfortunately this will never be the case as each unit needs a different amount of time and it is almost impossible to make them match. This section will analyse the timing requirements for each of the main tasks involved in the DEM, i.e. contact checking, forces and position update.

As can be seen from the equations below (Eq. 5–8, Eq. 5–9, Eq. 5–10, Eq. 5–11), the time required to compute the force and the positions of the particles and write data to the has

$$t(cc) = \frac{\dfrac{\frac{1}{2}N_{main}^{2}}{nrccunits} + \dfrac{\frac{1}{2}N_{right}^{2}}{nrccunits}}{4} \qquad \textbf{Eq. 5–8}$$

$$t(forces) = 6N \qquad \textbf{Eq. 5–9}$$

$$t(position) = N \qquad \textbf{Eq. 5–10}$$

$$t(read/write \quad ext\,mem) = \frac{(4 + 17 \times N) \times 2}{4} = \frac{4 + 17 \times N}{2} \qquad \textbf{Eq. 5–11}$$

not changed from that given by Eq. 5–6, Eq. 5–7. The only unit calculation time that has

changed is the time needed to perform the contact checks. Now it has two terms, and also depends on the number (nrccunits) of contact check units used. One term relates to the computation of the contact check in the main column (N_{main}) and another the computation of the particles that might be in contact with the balls in the column to the right (N_{right}). Once the contact check is performed for the main column, the particles within a distance of 2R from the column to the right are monitored (these are the only particles that could be in contact with the particles in the right column). If we assume that $N_{main} \approx N_{right}$, Eq. 5–8 can be simplified to:

$$t(cc) \approx \frac{N^2 / nrccunits}{4} \qquad \textbf{\textit{Eq. 5–12}}$$

This would mean that all the particles in the column would also be checked for contacts in the column to the right. Eq. 5–11 gives the expression of the time needed to read and write data to the external memory. Every ball is described by 17 parameters: x, y, v_x, v_y,θ, and the size of the normal and shear forces of each contact. Furthermore, every column has a header of 4 elements: coordinate of the column, number of balls in it and address of the first and last element of that column in the external memory. The whole expression is divided by a factor of four as it is performed at a four times faster clock rate than that of the forces and position update units and this clock speed is the same as the contact check units. Figure 5–22 shows a graphical representation of the theoretical equations given above with different cases in which the number of contact check units ranges from 5 to 20.

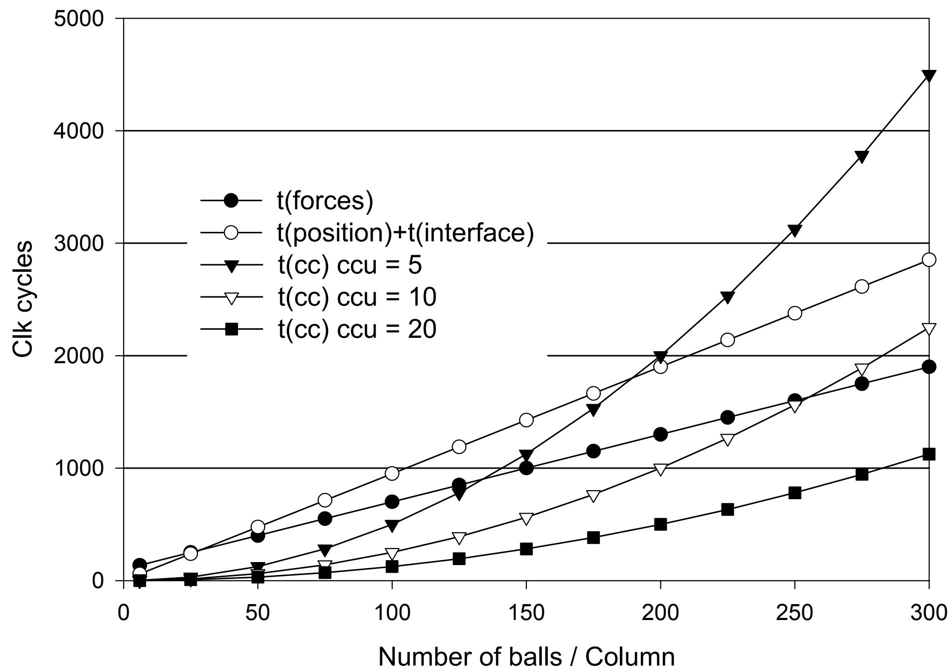


Figure 5–22 *Graph of clock cycles needed to compute t(forces) t (pos)+t(interface) and t(cc) for a different number of contact check units.*

In order to verify these expressions, a special modified hardware design was implemented. It was basically the same as in Figure 5–16, but the forces and position update units were omitted so that their resources could used to implement more contact check units in parallel. The time needed to perform the contact checking was measured running the system at a clock speed of 1 MHz for 500 cycles.

The measured experimental results are compared with theoretical predictions in Figure 5–23. The experimental and analytical values are very close, thus proving that the predictions obtained for the contact checking are sufficiently correct for practical design. Slight variations are inevitable as the number of particles in each column will never be exactly the same. Difference also occurs because the number of particles that need to be checked for contact with the particles in the column to the right of the main column will also change from column to column and from cycle to cycle.

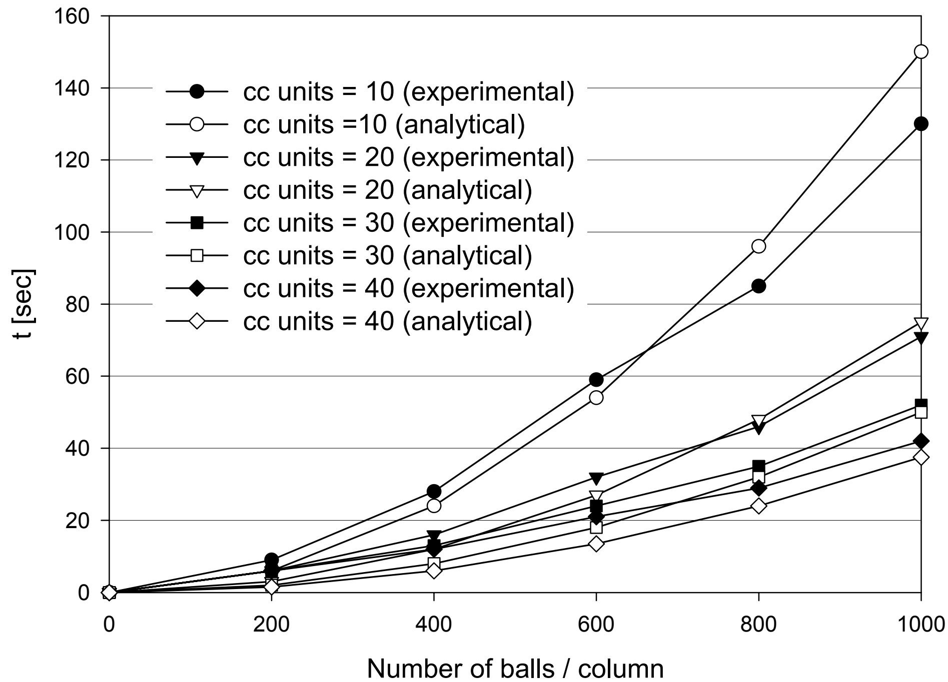


Figure 5–23 *Comparisons of experimental and analytical values to compute the contact checking for different number of contact check units.*

5.9.5 Hardware requirements

Due to limitations of hardware resources, only a maximum of five contact check units could be instantiated in parallel. Table 5–2 shows the amount of resources taken up by each individual unit as a percentage of the total FPGA resources. The design also makes use of 100% of the block RAM on the FPGA.

Table 5–2 *Hardware resources used for by this implementation*

XCV2000E	% OF NO. OF SLICES
Forces update	15 %
Coordinates update	11 %
Control unit	21 %
Interface to external memory	5 %
Switch inputs	10 %

Switch outputs	9 %
Write back unit	1 %
5 contact check units	8 %
Total	80 %

An FPGA logic resource utilization greater than 80% could not be achieved, since the designs would be impossible to route.

5.9.6 Internal Memory limitations

With the internal block RAM of the XCV 2000E device (655,360 bits), a maximum of 128 balls can be stored in each column, because although only 393,216 bits are used to store ball data, the rest is needed to implement e.g. FIFOs and KCMs, in order to save logic resources. One important question is whether additional internal RAM would bring any benefit to the design.

As the number of balls N is increased, the load balance between the different arithmetic units will change, because they have differing dependence on N. This can be offset by increasing the number of contact check units so that contact checking completes at the same time as the position update. It therefore appears reasonable that an increase of internal RAM would need to be accompanied by an increase in logic resources in order to have more contact check units working in parallel to speed the contact detection up.

From Table 5–2 it can be seen that 5 contact check units require 8% of the XCV 2000E resources. 3% of those resources are consumed by the top level, which controls the single contact detection units. Therefore, every contact detection unit requires approximately 1% of the FPGA resources. Eq. 5–13 and Eq. 5–14 show the number of extra bits needed to have one more ball in each of the six internal memory units.

$$1 \quad ball \quad is \quad equivalent \quad to \quad 256 \quad bits \times 2 = 512 \quad bits \qquad \textbf{\textit{Eq. 5–13}}$$

$$1 \quad ball \quad in \quad every \quad column \quad is \quad equivalent \quad 512 \quad bits \times 6 = 3.073 \quad kbits \qquad \textbf{\textit{Eq. 5–14}}$$

In order to get a good load balance, the time required to compute the position update, write the column data to the external memory should be equal to the length of time taken to

$$t(cc) = t(position) + t(read / write) \qquad \textbf{\textit{Eq. 5–15}}$$

compute all the contact checks (Eq. 5-17). This can be achieved by increasing the number of particles in each column.

For the case of 5 contact check units, substituting into Eq. 5–9 and Eq. 5–11and Eq. 5–12 gives a number of 175 balls per column in order to that this equation is satisfied.

As seen from this table (Table 5–3), every time the number of contact check units is doubled, this allows the doubling of the number of particles that can be stored in a column. This means that if the number of balls that can be held in the FPGA is doubled, the number

Table 5–3 *Growth of ideal number of balls/column as a function of the number of contact check units to make t(cc) = t(position)+t(r/w).*

NR OF CONTACT CHECK UNITS	NR OF BALLS/ PER COLUMN
5	175
10	350
20	700
40	1400

of contact check units have to be doubled as well as shown in Table 5–4,

Table 5–4 *Relation of number of balls allowed in the system to make t(cc)=t(pos)+ t(interface) and its memory requirements.*

NR OF BALLS/COLUMN	TOTAL NUMBER OF BALLS IN THE FPGA	MEM NEEDED (BITS)
175	1050	537,600
350	2100	1,075,200
700	4200	2,150,400
1400	8400	4,300,800

Thus if one contact check unit needs 1% of the XCV2000E device (192 slices), a doubling of the number of particles accompanying the doubling of memory size would require an increase an additional hardware resource of nrccunits × 192 slices.

5.10 Validation of the Hardware Designs

The previous sections have described in detail the hardware implementations. In order to validate the hardware implementations a debugger was incorporated into the software environment in order to compare the behaviour of the software and the hardware design. Figure 5–24 shows the initial state of the debugger. The blue lines and circles correspond to the hardware implementation and the black ones to the software implementation (but in the initial condition of Fig. 5-24 they lie on top of one another).

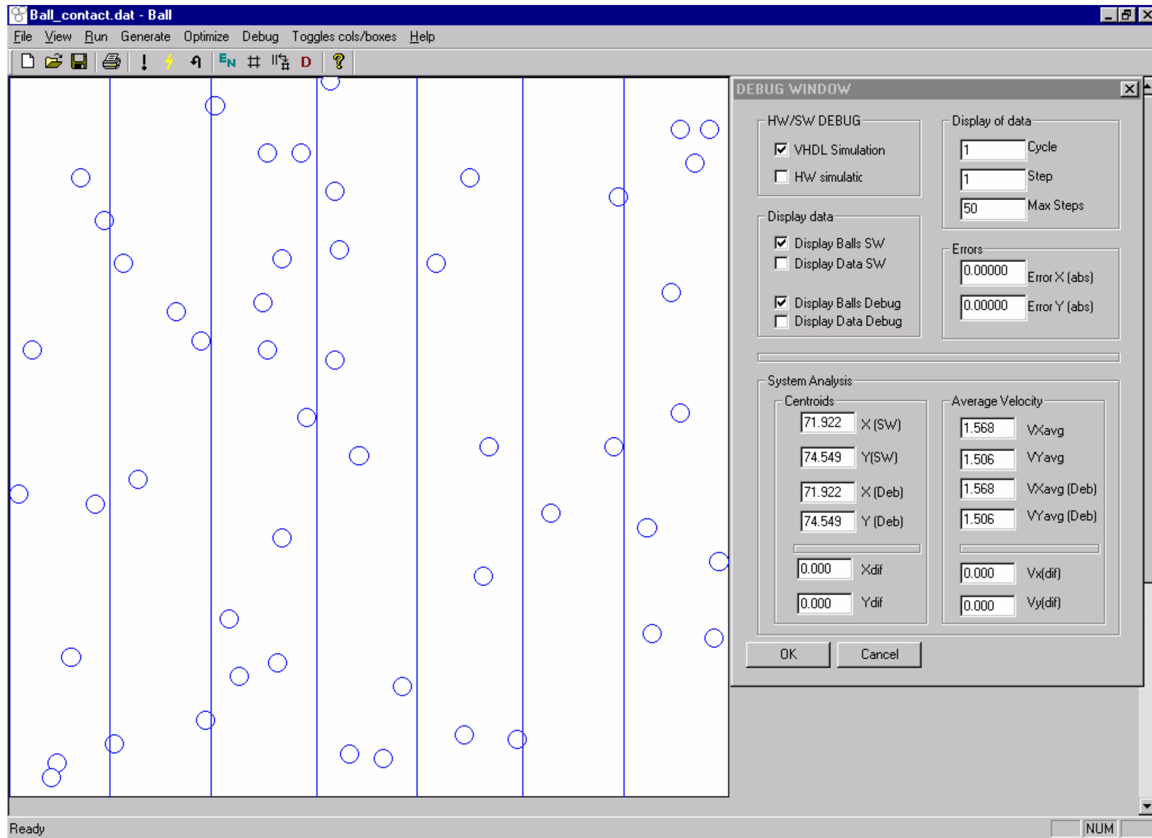


Figure 5–24 *Screen shot of the initial state of the hardware debugger*

There are two options available in the debugger:

1. Debug the result of the VHDL simulation.
2. Debug directly the hardware results obtained from the reconfigurable computing platform.

The debugger has options to give visual feedback, such as simultaneous display of the results for both hardware and software. Also, the debugger computes the difference between both sets of results. It calculates the sum of absolute differences (SAD) of the particles' coordinates and velocities. It also calculates bulk measures, such as the mean absolute velocity, and the centroid of the system.

As can be seen in Figure 5–25, the hardware implementation not only moves the particles, but also the column boundaries in order to maintain the same number of particles in each column so that the system remains balanced as described in section 5.9.1.3.

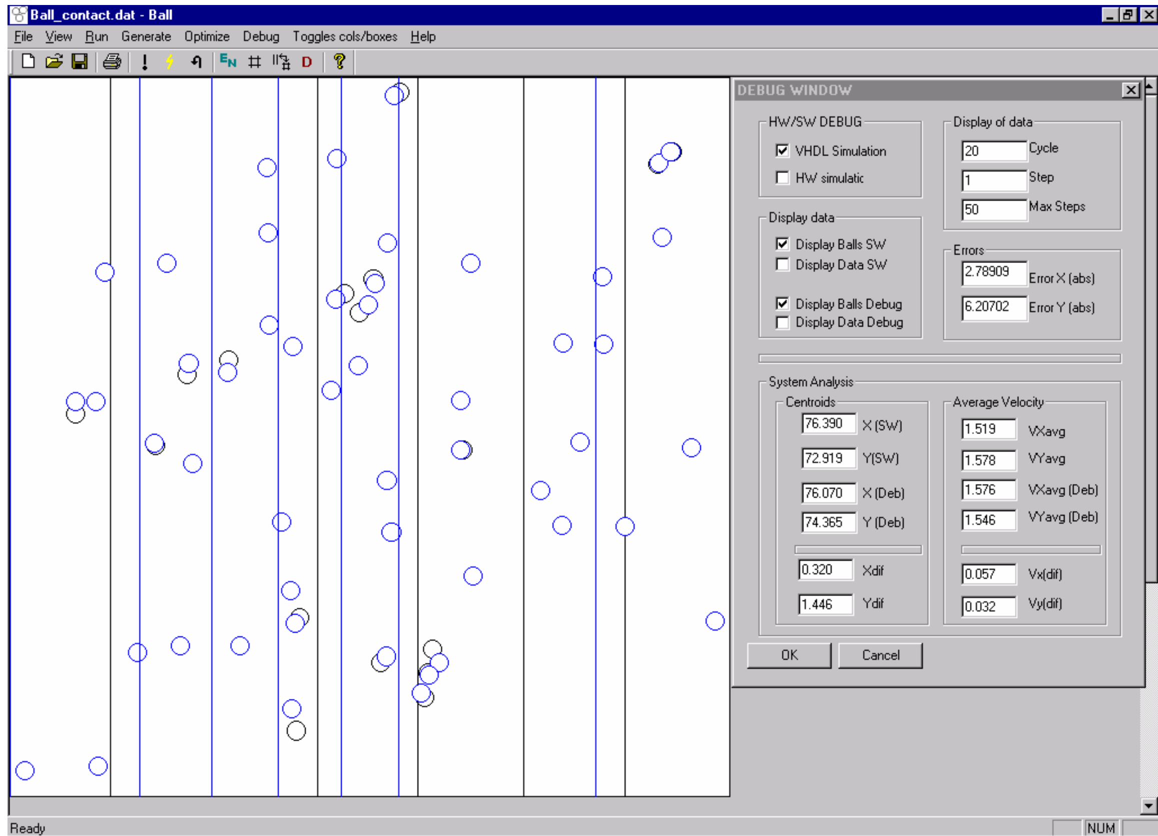


Figure 5–25 *Screen shot of a debugged system of 50 balls after 20 cycles*

From these analyses it was shown that the particles in the hardware system move slower than the ones in the software version. This was expected as the round-off error of the 16-bit arithmetic makes the values computed by the hardware implementation grow slower than their software counterparts. This will not make any major difference to the result of the complete simulations, since we are interested, as mentioned in section 5.6, in the simulated behaviour of the bulk, and not the behaviour of individual particles. Also in most discrete element simulations, the steady state (or static) result is sought and the dynamic path reaching it is less important.

16-bit arithmetic is sufficient to compute most of the simulations by scaling the problems. Care has to be taken when this scaling is performed in order to avoid many over and underflows. The next section will discuss the amount and effect of numerical under- and overflows in the design.

5.10.1 Over/Underflow Quantification

In order to quantify the number of over and underflows of the system, the software program was modified in order to simulate how the hardware is behaving in order to keep track of the under and overflows incurred during a simulation. An example pseudo code of this operation is shown in Figure 5–26. After every arithmetic operation an under and

```
ball_Fx_AM = ball->Fx/AM  // Any arithmetic operation

if(ball_Fx_AM > max_value OR ball_Fx_AM < - max_value)
  count_overflows_motion[0]++  //Register if an overflow happens

elseif((ball_Fx_AM<max_resolution OR (ball_Fx_AM>-max_resolution)
  count_underflows_motion[0]++// Register if an underflow happens
```

Figure 5–26 *pseudo code of under and overflow registration*

overflow check is inserted. If the value is bigger or smaller than the range of values representable on the hardware using either 16/24/32-bit arithmetic, then the program will register this. At the end of the simulation, the statistics will be written to a file.

Figure 5–27 shows a graphical representation of the number of underflows of a system of 50 balls, ran for 1000 cycles. Table 5–1 shows the number of arithmetic operands needed to compute the forces and the positions. In total approximately 7,500,000 arithmetic operations took place during this simulation (considering only the forces and position update unit). The dimensions and parameters were scaled in such a way that no overflow occurred. As can be seen, most of the underflows in the case of 16-bit arithmetic occur in the position update unit. Examination of Figure 5–12 shows the reason for this. As the inverse of the mass and the inverse of the moment of inertia are very small, the values multiplied by these parameters can easily become smaller than 0.0624 (2^{-4}).

By increasing the arithmetic to 24-bit or even 32-bits (fixed-point) the number of underflows is reduced significantly, as seen in the graph. This is of course at the expense of requiring far more hardware resources.

The parameters that will influence the amount of overflows and underflows will be the domain size (height and width), the particle radius (and hence the mass and moment of inertia), and the stiffness for the overflows and the time step for the underflow.

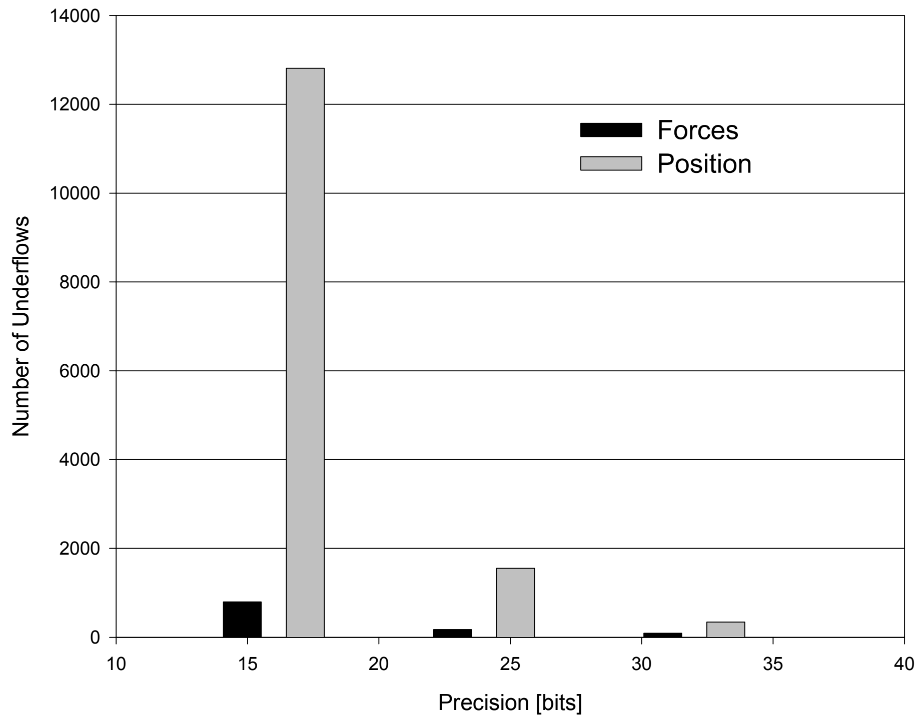


Figure 5–27 *Underflows in the forces and position update units as a function of the number of bits*

Knowing this, the software program, when it has read in the parameters from the data file, checks the key parameters to see if the generated system will be prone to overflow and underflow. If so a warning message is issued.

5.11 Discussion

A dedicated hardware architecture implemented on an FPGA was presented in this chapter. The low and high-level parallelism of the DEM is exploited in the last implementation, which overlaps the main computational tasks.

The validation of the implementations shows that the particles in the hardware simulations lag the ones in the software simulations after each cycle due to the limitation to 16-bit arithmetic that had to be used in order to fit the design into a Xilinx™XCV2000E FPGA. Scaling the problem parameters can avoid the need for a large dynamic range. This is, of

course valid for some simulations, but not for all. In some cases 16-bit fixed-point arithmetic will not be sufficient e.g. if an assembly with very large stiffness is to be simulated.

These implementations should be viewed as prototypes with the minimum requirements to allow DEM simulations, as the XCV2000E FPGA was state of the art around 3 years before the writing of this thesis. Since then, new larger FPGAs have become available, which would allow the system to be run at higher clock speeds, instantiating more units in parallel and using 32-bit arithmetic.

As seen from the timing considerations in section 5.9.4, reading and writing data in and out of the FPGA to the external memory is the slowest task. This is due to the RC1000 board's external SRAM memory, which allows a maximum transfer rate of 40 MHz.

Splitting the domain into columns to allow the overlapping of the computational tasks is a valid solution only if the number of particles in each column is less than the maximum number of particles that a memory unit of the FPGA can hold. The software simulator, before downloading the data to the FPGA board, checks that no column exceeds the maximum number of particles that the specific FPGA can hold per column. If the number of number of particles is larger, a warning signal is given to the user. Monitoring the number of particles in each column and adaptively moving the cell boundaries helps to balance the load (making all three tasks finish at nearly the same time) and to avoid overflow of the FPGA internal memory, which would result in loss of particles from the system.

5.12 Summary and Conclusions

A brief introduction to FPGAs and the reconfigurable computing platform were given in this chapter. Two hardware implementations were described. A simpler implementation made use only of the low level parallelism of the DEM. A more complex implementation was also described. This implementation made use of both the low level and high level parallelism of the DEM. This implementation was based on domain decomposition, which

meant that issues such as load balancing had to be considered. Both implementations were validated with the software implementation.

The main concepts on which the high and low level parallelsim implementation is based are briefly summarised below:

1. Firstly, splitting the internal FPGA memory into six independent units allows the overlapping of the three main computational tasks of the DEM. The re-boxing of particles transitioning from one column to another is completely free in terms of computing time as the left and right column of the particles being checked for moving to the neighbouring column are already cached in the FPGA's internal memory.

2. One of the aims of the design is to have all three units working in parallel and never having to stall any of them to wait for another to finish. As particles move across the domain, some sub-domains become much more heavily populated than others, with the result that the computation times for the sub-domains become unequal, so some units have to stall to wait for others to finish. Monitoring the number of particles in the system and moving the boundaries of each sub-domain adaptively alleviates this, giving a well balanced system throughout the simulation. As particles can be re-boxed with no time penalty this will not take any of the simulation time.

3. The contact check unit is very cheap in terms of hardware resources, because it does not look for true contacts, but instead just looks for particles whose bounding boxes overlap. This allows the instantiation of many of these units in parallel. In order to force the main tasks to finish at almost the same time, the instantiation of many contact check units is desirable, because the time needed to compute the contact between particles grows quadratically with the number of particles in that sub-domain.

4. Scaling the parameters of the simulation avoids the need for a large dynamic range, thus avoiding the need for floating point arithmetic. The main results of interest of DEM simulations is the bulk behaviour of the system and not the individual particles, making low precision arithmetic tolerable.

The great advantage of this architecture is that it exploits the massive parallelism inherent in the DEM. It has the advantages of being scalable to newer, faster and bigger FPGAs as it can easily be upgraded to 24 or 32 bit arithmetic, and more contact check units can be instantiated in parallel by changing a simple generic parameter.

5.13 References

[1] www.xilinx.com/datasheet

[2] Kean T., *"Tools of the third wave of Programmable Logic"*, 1st Celoxica User Conference, Stratford Upon Avon, 2001.

[3] Alfke P., *"The Future of Field Programmable Gate Arrays",* 5th Workshop on electronics for LHC (LEB99). Plenary session, University of Wisconsin, 1999.

[4] www.Celoxica.com

CHAPTER 6

SOFTWARE AND HARDWARE ANALYSIS

6.1 Introduction

This chapter compares the differences between the software and the hardware implementations in terms of speed-up, numerical precision and stability. It should not be forgotten that the primary aim of this work is not just to have a faster implementation, but one with correct results. It does not make sense to have a faster system if the results obtained are wrong. A careful precision and stability analysis is therefore needed, as well as a good understanding of the algorithm's behaviour.

In the current research, software (SW) and hardware (HW) implementations for the DEM were created. It is necessary to compare the results of each in terms of their numerical precision difference, as the 16-bit fixed point arithmetic of the hardware implementation will inevitably produce different results from those of the 32-bit floating point arithmetic of the SW implementation. The results obtained in software implementation using 32-bit floating-point arithmetic are considered to be the “correct” ones, against which the results obtained with the hardware implementation will be compared. (The reasonableness of this assumption is investigated in section 6.6).

6.2 Speed-up

There are many ways to evaluate the performance of a parallel algorithm. A very common criterion is speed-up, which is measure of how much faster a computation finishes on a parallel machine than on a uniprocessor machine (see Eq. 6-1). This concept was explained in detail in chapter 4, section 4.2.

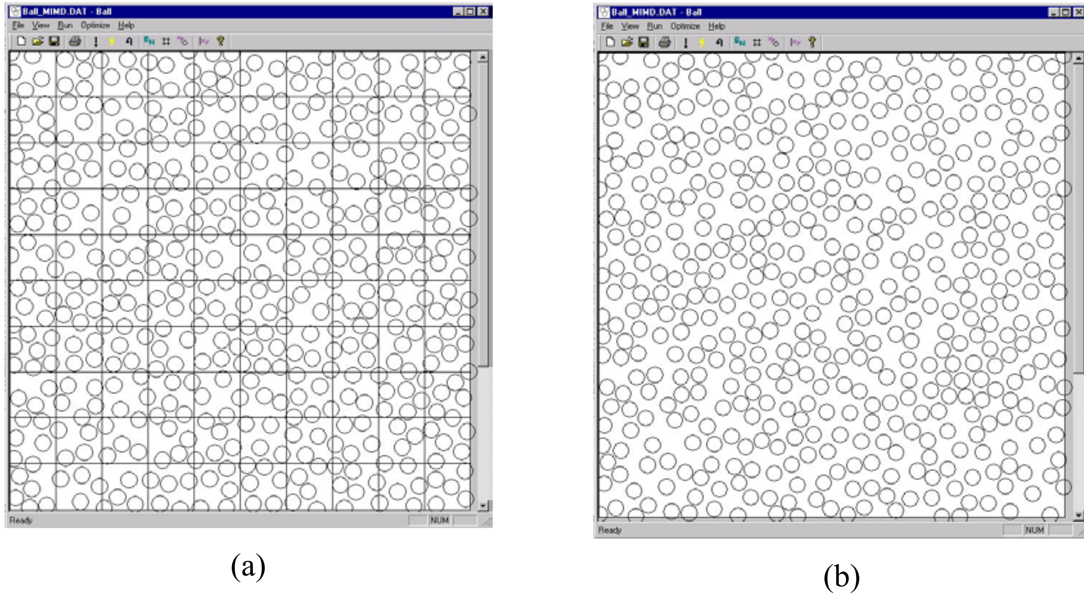
$$speedup = \frac{runtime \quad of \quad the \quad fastest \quad sequential \quad algorithm}{runtime \quad of \quad the \quad parallel \quad algorithm} \qquad \textbf{\textit{Eq. 6-1}}$$

The following sections will present a runtime comparison between the optimised software version presented in chapter 3, and the two hardware implementations (SIMD and MIMD) presented in chapter 5. In order to give a fair comparison, the domain decompositions used by the hardware and software are independently optimised, so that each method performs at its best.

6.2.1 Low Level Parallelism v. Software Implementation

A 2-dimensional DEM simulation using 500 particles generated randomly in a domain was carried out using the low level parallelism hardware implementation. 80% of the Xilinx™ XCV1000 FPGA resources were consumed using one instance each of the contact check unit, the coordinate update unit and the force update unit, as described in section 5.8. For comparison, a corresponding simulation was carried out on the optimised software version, described in chapter 4, on a PC with a 1GHz Athlon processor and 750 Mbytes of RAM. The DEM running in FPGA hardware was found to be 5.6 times faster.

The hardware version uses pipelining to obtain one new result (contact identification, force component update or movement update) on each clock cycle. The system works at a clock speed of 30 MHz, but within the arithmetic pipelines it uses a slower clock of about 7.5 MHz in order to keep these pipelines fully loaded. By contrast, the software version runs on a processor of very high clock speed, but takes many hundreds of clock cycles to generate each result. Figure 6-1 shows the initial state of the simulations for the software (a) and the hardware (b) simulation.



(a) (b)

Figure 6-1 *Initial state of the 500 domain assembly for the SW (a) and for the HW (b) simulation*

For the SW simulation, the domain is split into a near-optimal grid size in order to achieve the fastest possible simulation time. The HW simulation has no grid. Table 6–1 shows the simulation time for 1000 time steps each of the SW and the HW simulations.

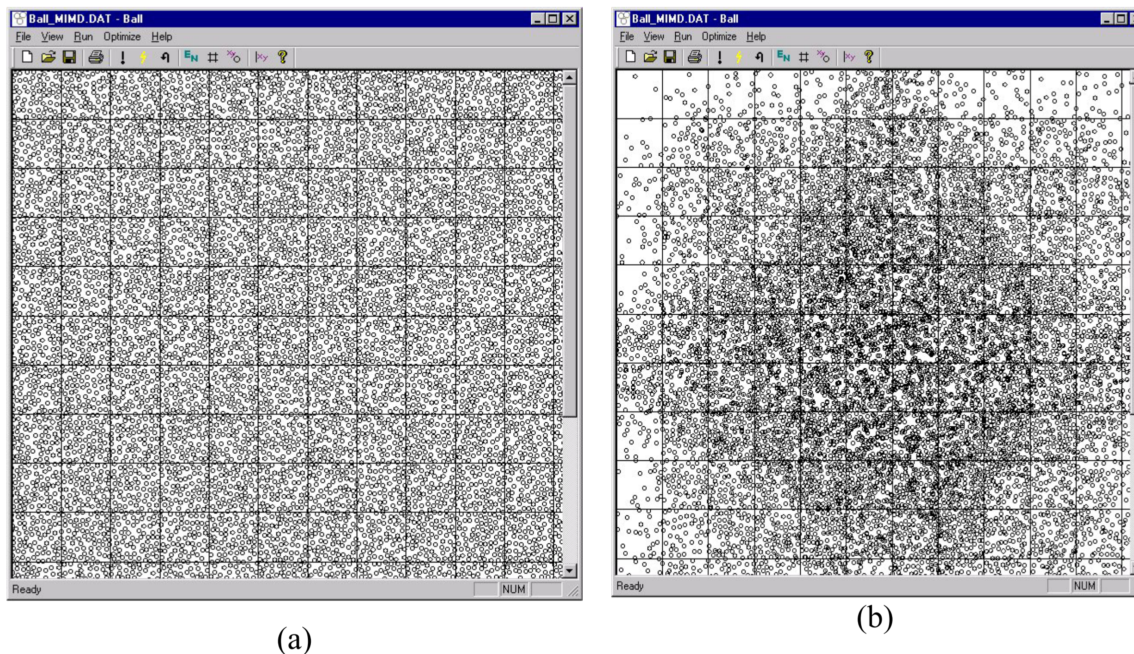
Table 6–1 *Simulation time for the SW and the HW implementations for 500 particles*

	T(SW)	T(HW) SIMD
Simulation time	0.73 s	0.13 s

6.2.2 High and Low Level Parallelism v. Software Implementation

An experiment was set up in order to measure the effectiveness of the high and low level parllelism design. Domains with 50,000, 75,000, 100,000, 125,000, 150,000, 175,000, and 200,000 particles were generated and simulated for 1,000 time steps. The performance of the software version was measured and compared with the results obtained by the hardware version.

Figure 6-2 shows one of the simulations with 50,000 particles. Figure 6-2 (a) shows the initial state of the randomly generated particles in the domain, with their initial velocities



(a) (b)

Figure 6-2 *(a) Initial state of the system of 50,00 particles*
(b) Final state after 1000 time steps (SW version).

pointing towards the centre of the domain. Figure 6-2 (b) shows the final state after 1000 cycles.

Table 6–2 shows a comparison between the speed-up achieved by the hardware simulation as compared to the software running on an 1GHz Intel™ Pentium© III based PC with 1.3 Gbytes of RAM.

Table 6–2 *Run time for the software and hardware simulation and speed-up results*

NO. OF PARTICLES	50,000	75,000	100,000	125,000	150,000	175,000	200,000
$Time_{software}$[s]	1800	2485	3120	3920	5156	6123	7423
$Time_{hardware}$[s]	51	80	103	130	175	196	245
Speed-$up_{measured}$	35.3	31.0	29.8	30.2	29.5	31.2	30.3

Figure 6-3 shows graphically the achieved speed-up. The slight variations between the results were due to load distribution differences between the simulations.

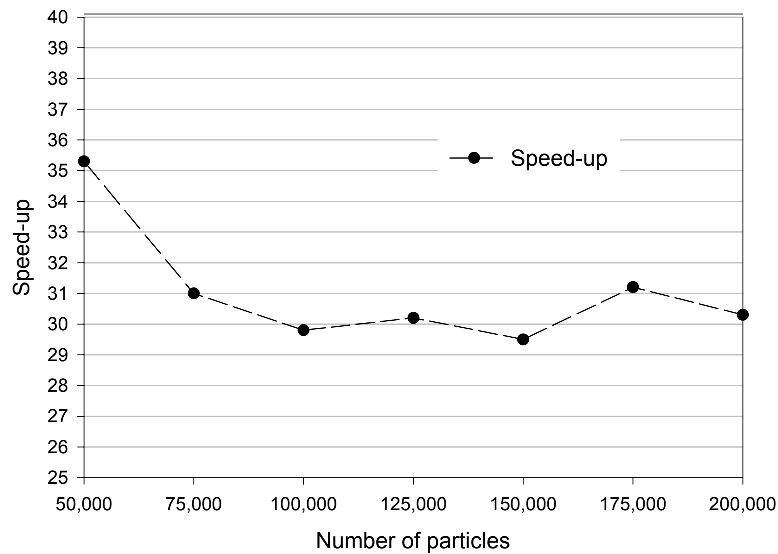


Figure 6-3 *Graphical representation of the measured and ideal speed-up*

The previous chapter showed that for a maximum of 125 balls/column (limited by the FPGA's internal RAM) the position update and the read and write operation of new data to the external memory is the slowest stage, and limits the overall speed of the hardware system.

6.2.3 Discussion of the Speed-up Results

Both HW implementations have accelerated the simulation, the first design by a factor of 5.6 and the second design by a factor of approximately 30. Given that the FPGA runs at a clock speed far lower than the PC microprocessor, this shows that the hardware implementations make very good use of the intrinsic parallelism of the DEM.

For a given number of particles within a subdomain, the hardware implementations will always take the same length of time to simulate a time step, because it will always treat each particle as if it had six contacts, irrespective of how many true contacts it actually has. Thus the computing time will grow linearly with the number of particles in the system. By contrast, the software implementations run time depends on the number of true contacts that the particles have. The system parameter that has the greatest influence on this is the stiffness. The lower the stiffness, the more contacts are generated, and the more often the

resultant forces need to be computed for each particle. If the stiffness is sufficiently large, the system will behave like a billiard table, thus contacts will last for only a very brief duration. In order to ensure that the estimates of speed-up were conservative, the stiffness value used for the simulations was set to the maximum allowed by the 16-bit data arithmetic used by the hardware implementation, in order to minimise the software SW runtime. Use of a lower stiffness value would give even better speed-up results, since the software simulator would take longer, due to having a larger number of contacts.

6.3 Data Precision

The previous section has shown that the hardware design is faster than the software version running on a fast computer, but this has to be examined in conjunction with consideration of the accuracy of the results. It does not make sense to have a fast system if the result obtained is wrong. This section and the next describe the estimation of the difference in precision between the hardware and the software implementation. Section 6.5 describes measurements on the hardware system to confirm the estimates.

Some basic concepts regarding this numerical analysis will be analysed in the following sub-sections.

6.3.1 Basic Concepts

The data format used for the hardware implementations was described in the previous chapter in detail and is reproduced in Figure 6-4 in order to define some key concepts given below [2]:

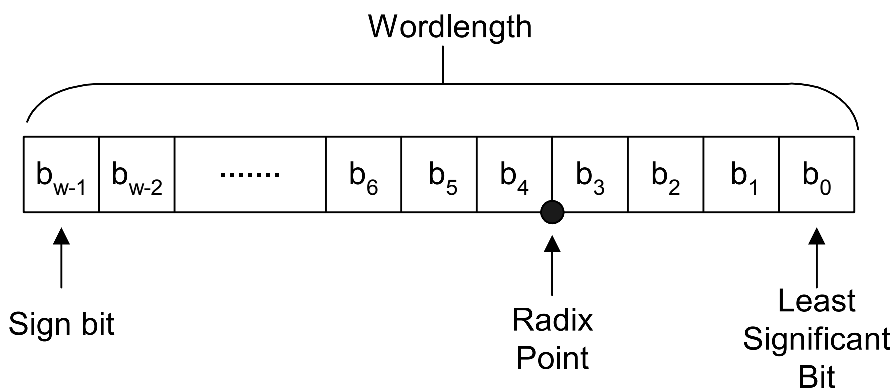


Figure 6-4 *16-bit data format*

Precision: Precision is the maximum number of non-zero bits representable. For fixed-point representations, precision is equal to the wordlength. For the 16-bit arithmetic, B(12,4) used, with 4-bits reserved for the fractional part: the precision is 16 bits.

Resolution: Resolution is the smallest non-zero magnitude representable. In this particular case: $resolution = 2^{-4} = 0.0625$

Unit in Least Significant Position (ulp) is the number that corresponds to a 1 in the least significant bit, and a zero in all other bits. In this particular case, 1 ulp = 0.0625.

Range: Range is the difference between the most negative number representable and the most positive number representable. In this particular case the range would go from 2047.9375 to –2048.

Accuracy: Accuracy is the magnitude of the maximum difference between a real value and its representation. Accuracy and resolution are related as follows:

$$Accuracy = \frac{Resolution}{2} \quad \textbf{Eq. 6-2}$$

For the 16-bit arithmetic with 4 bits reserved for the fraction part the accuracy would be $2^{-4}/2 = 0.03125$.

Absolute Error Δ: The absolute error is the *distance* between the number x and the estimate x'. $\Delta = |x - x'|$ [2].

Relative Error δ: The absolute error does not take into account the magnitude of the numbers involved. The relative error measures the error relative to the size of the number itself. $\delta = \frac{\Delta}{|x.|}$

6.3.2 Computation Errors

There can be several sources of computational errors. The most commons ones are:

- Errors in the original data
- Truncation errors
- Round-off errors
- Propagated error (in stable and unstable algorithms)
- Overflow errors

In this case, since the software and the hardware simulation start with the same data sets, it is assumed that there is no error in the original data. As the system parameters are chosen in a way that it is guaranteed that no overflows will occur, this source of error is also discarded.

The next sub-sections will describe in detail how computational errors appear as a result of the 16-bit data format used in the HW implementations.

6.3.2.1 Chopping Errors

The easiest way for the hardware system to store a value that has more than 16-bits is to chop (ignore) all the digits after the LSB (Least Significant Bit) of the 16-bits. Figure 6-5 shows an example of chopping. Two numbers are multiplied and the result cannot be represented using only four digits for the fractional part. In this example the absolute error is 0.015625 and the relative is $9.95 \times 10^{-4.}$

The worst cases error that can be introduced to a system due to chopping is equal to the resolution, which in this case would be 2^{-4}.

For our design, in the case where results are too large to represent with 16-bit arithmetic, where possible, a larger wordlength is used to represent the data until a bit reduction is naturally produced. For example, after a multiplication a value is obtained that may have 24 non-zero bits before the decimal point. These additional bits are retained until the value is representable within 16 bits.

$$3.75 \times 4.1875 = 15.703125$$

$$0011{\bullet}1100 \text{ x } 0111{\bullet}0011 = 1111{\bullet}1011\ \underbrace{0100}_{\text{chop}}$$

$$1111{\bullet}1011 = 15.6875$$

$$\Delta = 15.703125 - 15.6875 = 0.015625$$

$$\delta = \frac{0.015625}{15.703125} \approx 9.95 \times 10^{-4}$$

Figure 6-5 *Chopping example*

Much research into optimum wordlength allocation has been carried out in the last few years, especially for DSP systems implemented on FPGAs [3][4][5]. Some tools have been reported in the literature that allow the designer to perform estimations of a performance and area trade-off [6][7]. In this case in order to have a fully working prototype, a trade-off between optimum wordlength and the fitting of the system on the available FPGA had to be accepted.

6.3.2.2 Rounding Errors

A rounding error is introduced when a number's precision is reduced by rounding the bits to the right of the LSB, in order that the number can be represented with the available bits. Figure 6-6 shows an example of rounding error, and compares it with chopping error. The

$$4.375 \times 3.125 = 13.671875$$

$$0100{\bullet}0110 \text{ x } 0011{\bullet}0010 = 1101{\bullet}1010\ \underbrace{1100}_{\text{Rounding}}$$

$$1101{\bullet}1011 = 13.6875 \text{ Rounding result}$$

$$1101{\bullet}1010 = 13.625 \text{ Chopping result}$$

$$\Delta_{rounding} = \left|13.671875 - 13.6875\right| = 0.015625$$

$$\Delta_{chopping} = \left|13.671875 - 13.625\right| = 0.046875$$

$$\delta_{rounding} = \frac{0.15625}{13.671875} = 1.14 \times 10^{-3}$$

$$\delta_{chopping} = \frac{0.046875}{13.671875} = 3.43 \times 10^{-3}$$

Figure 6-6 *Rounding example and comparison with chopping*

worst-case error that can be introduced by rounding is $\frac{resolution}{2}$ and in the current hardware implementation $2^{-4}/2 = 0.03125$, which is equal to half the error introduced by chopping.

Rounding error is equal to only half the error of chopping, but this is achieved at the expense of additional hardware cost. This is because rounding is normally achieved by adding 0.1 ulp to the number before chopping, thus requiring an additional adder. An algorithm similar to that of Figure 6-7 has to be used.

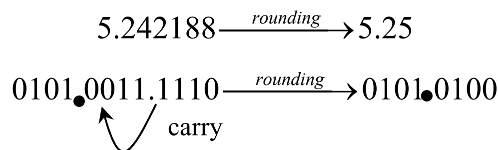


Figure 6-7 *Rounding carry example*

Due to the fact that the hardware design implemented consumed almost all of the usable FPGA, chopping rather than rounding was used.

6.4 Errors in arithmetic operations

The different arithmetic operations differ in how badly they are affected by the finite precision representation of the result. It is important to know whether the errors can grow so large as to cause all accuracy in the solution to be lost. This section attempts to estimate the effects of errors in the arithmetic operations on the results of the hardware DEM.

6.4.1 Worst Case analysis

Depending on the arithmetic operations used, there are various possibilities in terms of the error that may be introduced into the result. For example, in the case that two very close numbers are subtracted, almost all of the significant digits could be lost. If the resulting answer is then multiplied by a large number, this gives rise to a very inaccurate result. A number of different cases are described below for the most common arithmetic operations as a function of the operands.

MULTIPLICATION (a x b)

The following table shows the different errors introduced by a multiplication depending on the values of the input values.

Table 6–3 *Worst case analysis for the multiplication as a function of the input values*

$\lvert a \rvert < 1$ $\lvert b \rvert < 1$	$\lvert a \rvert < 1$ $\lvert b \rvert > 1$	$\lvert a \rvert > 1$ $\lvert b \rvert < 1$	$\lvert a \rvert > 1$ $\lvert b \rvert > 1$

If ($a \times b > 2^{-4}$)
Max. Error = 2^{-4}
else
All accuracy lost
	Max. Error = 2^{-4}	Max. Error = 2^{-4}	Max. Error = 2^{-4}
			(If no overflow occurs)

As seen above, only if both input values are smaller than unity and the result of the multiplication is smaller than the precision (2^{-4}) will all the digits be lost. An underflow condition would have happened.

DIVISION (a / b)

The following table shows the worst-case scenarios for a division of two 16-bit numbers with the previously described data format:

Table 6–4 *Worst case analysis for the division as a function of the input values*

$\lvert a \rvert < 1$ $\lvert b \rvert < 1$	$\lvert a \rvert > 1$ $\lvert b \rvert < 1$	$\lvert a \rvert < 1$ $\lvert b \rvert > 1$	$\lvert a \rvert > 1$ $\lvert b \rvert > 1$

If (a >> b)
2^{-4}
If (a≈b)
2^{-4}
If (b >> a)
All accuracy lost
	2^{-4}	If the difference is large enough:
		All accuracy lost
			2^{-4}

As seen in this table, there are two cases in which all accuracy can be lost.

- when the denominator b is bigger than the numerator a, and both are smaller than one
- when the numerator is smaller than 1 and the denominator is big

Therefore great care must be taken when divisions are used.

ADDITION (a + b)

The maximum error for the worst case in all the addition cases, independently of the values of a and b, is equal to the precision of the numerical representation. In this case 2^{-4}.

SUBTRACTION (a – b)

Subtraction, like the division, a very problematic operation, as two close numbers leads to the loss of all significant bits.

Table 6–5 *Worst case analysis for the subtraction as a function of the input values*

$\lvert a \rvert < 1$ $\lvert b \rvert < 1$	$\lvert a \rvert > 1$ $\lvert b \rvert < 1$	$\lvert a \rvert < 1$ $\lvert b \rvert > 1$	$\lvert a \rvert > 1$ $\lvert b \rvert > 1$

If ($a \approx b$)
Lose all accuracy
else
2^{-4}
	2^{-4}	2^{-4}	If ($a \approx b$)
			LOSE ALL ACCURACY
			else
			2^{-4}

6.5 Error Propagation in Computer Arithmetic

This section will summarise the basic ideas about how errors propagate through arithmetic operations. The error propagation model is presented first, with some basic ideas about its statistical analysis. The error propagation in each arithmetic operations is presented afterwards.

6.5.1 Error propagation model

The truncations/rounding after each arithmetic operation can be modelled by replacing each truncation/rounding by and addition with an error signal as shown in Figure 6-8 [8], where *Opr* stands for any arithmetic operations.

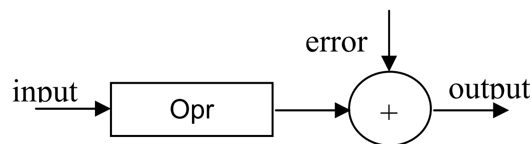


Figure 6-8 *Truncation/Round off propagation model*

Every time the signal is truncated/rounded, a noise source with variance σ^2 is constructed. The sources are assumed to be uncorrelated with each other and with themselves [9]. The noise model can therefore be described as a standard continuous rectangular distribution [8], as shown graphically in Figure 6-9 and analytical in Eq. 6-3, where the maximum error introduced in the system is equal to 2^{-4}, which is the equal to the resolution of the 16 bit data representation given in section 6.3.

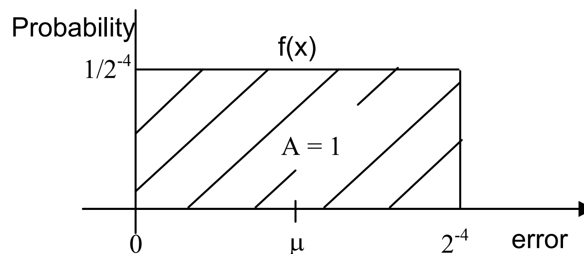


Figure 6-9 *Noise distribution model*

$$f(x) = \frac{1}{2^{-4}} = 2^4 \qquad \textbf{\textit{Eq. 6-3}}$$

μ stands for the mean error of the distribution and is given by Eq. 6-4.

$$Mean = \mu = \int_{0}^{2^{-4}} x \quad f(x)dx$$ **Eq. 6-4**

The variance of the distribution on the other hand is given by Eq. 6-5.

$$Variance = \sigma^2 = \int_{0}^{2^{-4}} x^2 \quad f(x)dx - \mu^2$$ **Eq. 6-5**

The mean and the variance of the truncation error distribution can be obtained using the distribution function shown in Eq. 6-3 and the expression for the mean (μ) and variance given in Eq. 6-4 and Eq. 6-5, as shown below.

$$\mu = \int_{0}^{2^{-4}} x \quad f(x)dx = \int_{0}^{2^{-4}} x \quad 2^4 dx = 2^4 \left. \frac{x^2}{2} \right|_0^{2^{-4}} = 2^4 \frac{\left(2^{-4}\right)^2}{2} - 0 = 0.03125 = \frac{ulp}{2}$$ **Eq. 6-6**

The mean of the error introduced after every arithmetic operation is as expected equal to half the resolution (*ulp*), which is half of the worst case value (*ulp*).

$$\sigma^2 = \int_{0}^{2^{-4}} x^2 \quad f(x)dx - \mu^2 = \int_{0}^{2^{-4}} x^2 \quad 2^4 dx - \mu^2 = 2^4 \left. \frac{x^3}{3} \right|_0^{2^{-4}} - \mu^2 = 2^4 \frac{\left(2^{-4}\right)^3}{3} - \left(\frac{ulp}{2}\right)^2 = 3.25 \times 10^{-4}$$ **Eq. 6-7**

The next sections will give an overview of two statistical concepts, the normal distribution and the central limit theorem, which will be used for the error introduction and propagation analysis that follows.

6.5.1.1 Normal Distribution

Also called a *Gaussian distribution*, this is in practice one of the most important distributions, since experimental errors are often normally distributed to a good approximation. The normal distribution describes many situations where observations are distributed symmetrically around the mean. 68% of all values under the curve lie within one standard deviation of the mean and 95% lie within two standard deviations. Its density function expressioni if given by Eq. 6-8 and shown in Figure 6-10.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left[-\frac{1}{2\sigma^2}(x-\mu)^2\right]}$$ ***Eq. 6-8***

The function depends on two parameters: μ, which the mean and σ, which is the standard deviation.

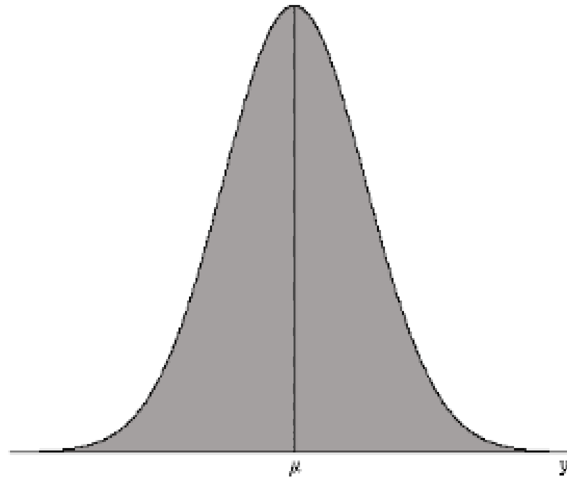


Figure 6-10 *Normal distribution function*

The normal distribution has the following characteristics:

- The graph has a single peak at the centre, this peak occurs at the mean.
- The graph is symmetrical about the mean.
- The graph never touches the horizontal axis.
- The area under the graph is equal to 1.

The normal distribution with μ = 0 and σ = 1 is called standard normal distribution and its distribution is given in predefined tables, which can be looked at in any statistic book. An arbitrary normal distribution can be converted to a standard normal distribution by making a variable change in Eq. 6-8 (for more about these concepts please refer to any statistic book).

6.5.1.2 Central Limit Theorem

Central Limit Theorem states the following:

"The average of the sum of a large number of independent, identically distributed random variables with finite means and variances converges "in distribution" to a normal random variable"

This definition implies that if a variable is the sum of others:

- The mean (μ) of the sum is equal to the sums of the means
- The variance (σ) of the sum is equal to the sum of the variances

Having the distribution of the truncation error introduced in each stage and applying the central limit theorem as the sum of the errors introduced in each stage after each truncation a global expression for the distribution of the error introduced in the position and forces update unit can be obtained.

6.5.2 Arithmetic Operations Error Propagation

The following examples show the error propagation in the arithmetic operations. The examples use the approximation a' as the finite precision representation of the number a. Let a' and b' be the estimate of numbers a and b, so that $a = a' \pm \Delta_a$ and $b = b' \pm \Delta_b$ (with Δ_a and Δ_b being the absolute error). The effect of most important arithmetic operations are presented below.

6.5.3 Addition

The total error of an addition is:

$$\begin{aligned} \Delta_{a+b} &= |(a+b) - (a'+b')| \\ &= |a-a' + b-b'| \\ &\leq |a-a'| + |b-b'| \\ &= \Delta_a + \Delta_b \end{aligned} \qquad \textit{Eq. 6-9}$$

where the triangle inequality has been used. So the maximum total absolute error of an addition is bounded above by the sum of the individual absolute errors.

On the other hand, the relative error is:

$$\delta_{a+b} = \frac{\Delta_{a+b}}{|a+b|} = \frac{\Delta_a + \Delta_b}{|a+b|} \approx \frac{\Delta_a + \Delta_b}{|a'+b'|} \qquad \textbf{\textit{Eq. 6-10}}$$

6.5.4 Subtraction

For subtraction, the total error is given by:

$$\begin{aligned} \Delta_{a-b} &= |\,(a-b) - (a'-b')\,| \\ &= |\,a-a' - (b-b')\,| \\ &\leq |a-a'| + |b-b'| \\ &= \Delta_a + \Delta_b \end{aligned} \qquad \textbf{\textit{Eq. 6-11}}$$

This expression is the same as for addition. The relative error is similarly the same as that of addition.

$$\delta_{a-b} = \frac{\Delta_{a-b}}{|a-b|} = \frac{\Delta_a - \Delta_b}{|a-b|} \approx \frac{\Delta_a - \Delta_b}{|a'-b'|} \qquad \textbf{\textit{Eq. 6-12}}$$

As mentioned in the previous section, subtraction (or equivalently addition of two numbers of opposite sign) can be problematic when the numbers are close in magnitude, because the relative error becomes very large.

6.5.5 Multiplication

The mathematical derivation of the propagation of error in multiplication is shown in the following expressions:

$$\begin{aligned} \Delta_{ab} &= |\,(ab) - (a'b')\,| \\ &= |(a' + \Delta_a)(b' + \Delta_b) - (a'b')\,| \\ &= |a'\Delta_b + b'\Delta_a + \Delta_a \Delta_b\,| \end{aligned} \qquad \textbf{\textit{Eq. 6-13}}$$

$$\leq |a'\Delta_b| + |b'\Delta_a| + |\Delta_a\Delta_b|$$

$$\approx |a'|\Delta_b + |b'|\Delta_a \qquad \text{with } |\Delta_a\Delta_b| \approx 0$$

On the other hand, the relative error is equal to the sum of the individual relative errors.

$$\delta_{ab} = \frac{\Delta_{ab}}{|ab|} \approx \frac{\Delta_{ab}}{|a'b'|} = \delta_a + \delta_b \qquad \textbf{\textit{Eq. 6-14}}$$

6.5.6 Division

The analysis for division analysis is very similar to that of multiplication. The absolute error expression is given below:

$$\Delta_{a/b} = |\,(a/b) - (a'/b')\,|$$

$$\leq \frac{(\,|a'|\,\Delta_b + |b'|\,\Delta_a)}{(b')^2} \qquad \textbf{\textit{Eq. 6-15}}$$

And the relative error is:

$$\delta_{ab} = \frac{\Delta_{a/b}}{|a/b|} \approx \frac{\Delta_{a/b}}{|a'/b'|} = \delta_a + \delta_b \qquad \textbf{\textit{Eq. 6-16}}$$

6.6 Arithmetic Error Analysis of the Hardware Implementation

This section will analyse the introduction of the error after each arithmetic operation in the forces and position update unit.

6.6.1 Discussion of the Worst Case Analysis

As explained is section 6.3.3, there can arise situations where all significant figures output by finite precision arithmetic operators are wrong. This normally occurs where the result of an underflow is multiplied by a large number.

Figure 6-11 repeats the internal architecture of the velocity and position update pipeline (this was originally explained in section 5.8). Whether or not catastrophic loss of accuracy will occur is dictated by the nature of the pipelines, and by the data fed to them.

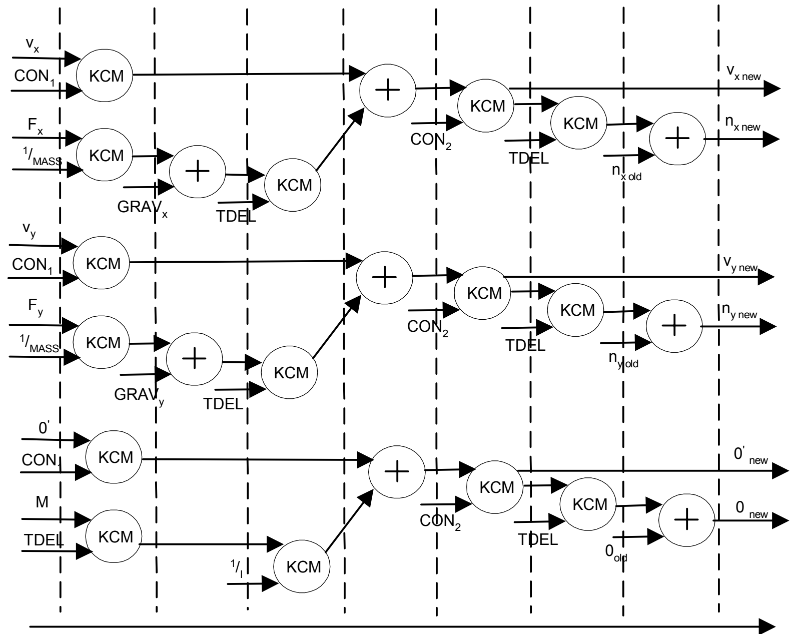


Figure 6-11 *Velocity and Position update unit internal structure*

Consideration of Figure 6-11 shows that in this pipeline catastrophic loss of accuracy will not occur. One situation that might tend to create an underflow is the multiplication of two small numbers. If this underflow were then multiplied by a large number, this would cause a serious loss of accuracy. However, the paths through multipliers terminate on adders; if one of the inputs is an underflow, then the impact on accuracy is minimal.

Another situation that might give rise to underflow is the addition of two numbers that are very close in magnitude, but opposite in sign. There are a few places where this could happen, e.g. if F_x/mass ≈ $Grav_x$; however, these situations would crop up very rarely in practice, and in any case the result is multiplied by the time step, which is always less than 1.

Figure 6-12 repeats the internal structure of the force update unit. There are several areas of the pipeline that could present a problem. The first of these is the initial stage that computes the argument for the sin/cos look-up table.

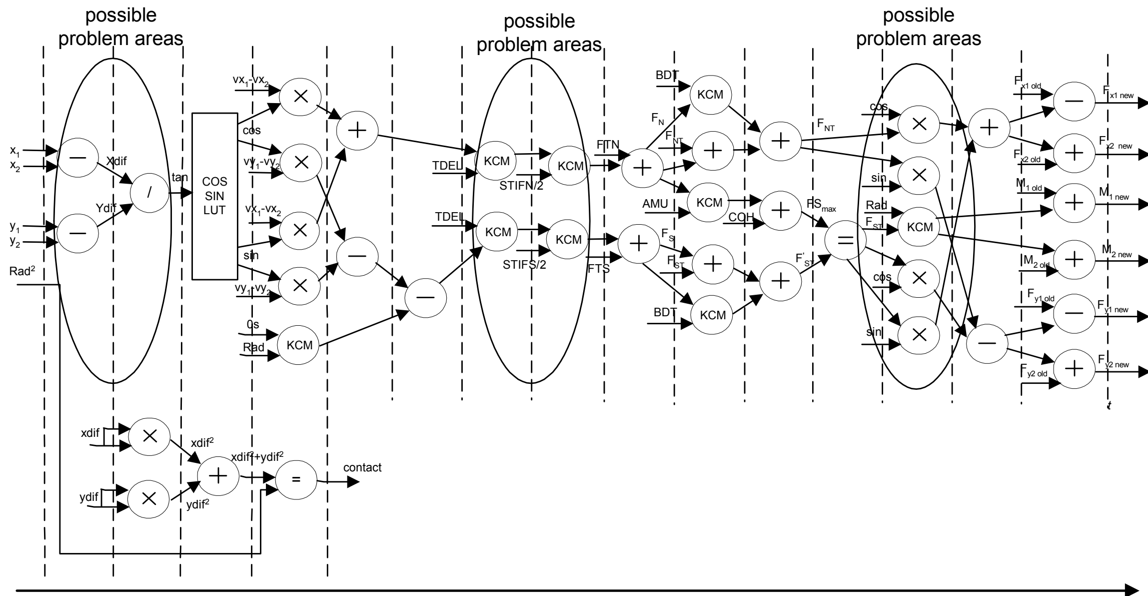


***Figure 6-12** Forces update unit internal structure*

This contains two subtractions and a division, all of which are potentially problematic operations, and could lead to the introduction of a large error. Consideration of the meaning of the operands shows that the division is in fact not problematic, since x_{dif} and

$$x_{dif} = x_1 - x_2 \quad \textbf{Eq. 6-17}$$

$$y_{dif} = y_1 - y_2 \quad \textbf{Eq. 6-18}$$

$$\tan = \frac{y_{dif}}{x_{dif}} \quad \textbf{Eq. 6-19}$$

y_{dif} (shown in Eq. 6-17 and Eq. 6-18) will always be of similar size, and are smaller than twice the radius. (x_{dif} and y_{dif} represent the differences between the centroid co-ordinates of two particles in contact). However, the subtractions will cause a problem if the two particles are closely aligned either horizontally or vertically (see Eq. 6-19 and Figure 6-13). In order to handle this, the hardware checks for the two special cases x_{dif} = 0 or y_{dif} = 0 and handles them separately. If x_{dif} = 0 then the LUT outputs cos = 0 and sin =1; if y_{dif} = 0 the LUT outputs cos = 1 and sin = 0.

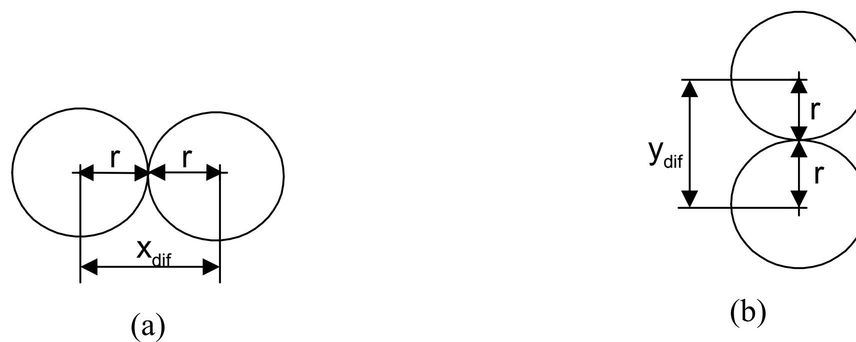


Figure 6-13 *Special cases for cosine and sine*

A second potential problem area is at the 8th stage when the incoming number is multiplied by the time step (which is normally small) and this result is then multiplied by the stiffness (which is normally large). This problem is resolved by adding 4 additional bits to the fractional part of these paths, so in this region the number representation is B(12,8) rather than B(12,4).

After this stage, there are no further problems areas in the pipeline, as all of the multiplications involve values larger than the unity, except for the sine and cosine multiplications at stage 12. The outcome of the sine and cosine multiplications is added onto the total force, so they do not contribute a large error.

Having established that the pipelines are unlikely to be affected by catastrophic worst-case error scenarios, in the next sub-sections an estimate of typical error build up is formulated.

6.6.2 Forces update unit

This unit is the most computationally expensive, with the most pipelined arithmetic stages (15 in total). This gives rise to a higher possibility of error generation and propagation.

The truncation after each arithmetic operation adds a noise and a bias component to the result. These effects are carried over from one pipeline stage to another so that at the end of the pipeline the result obtained using this data format will differ from the one obtained using floating point numbers by a certain amount.

In order to compute this amount Figure 6-12 has been expanded to include the noise and bias introduction after each arithmetic operation as shown in Figure 6-14.

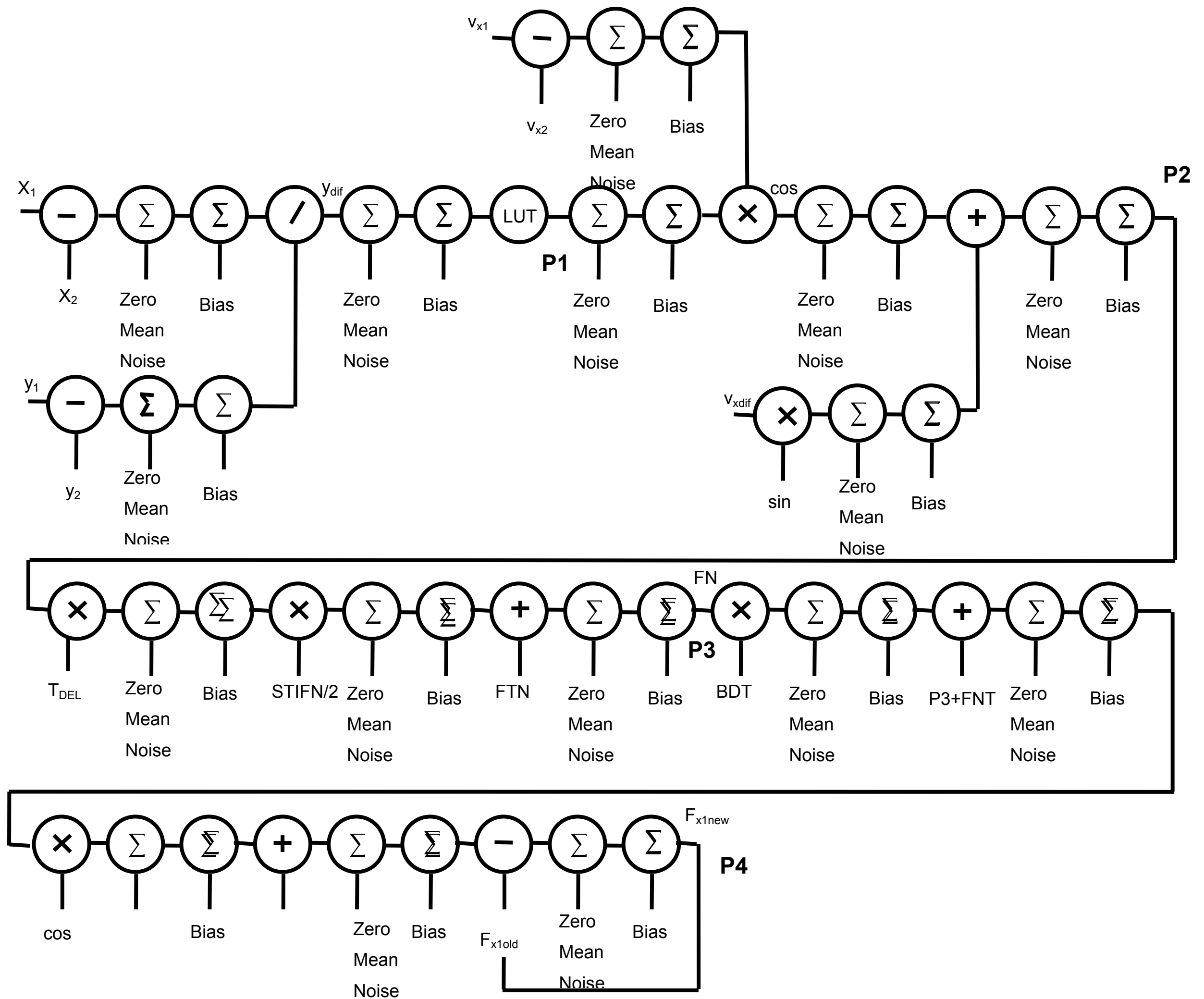


***Figure 6-14** Expanded flow graph for the F_x force pipeline in the forces update unit*

Every stage in the pipeline as shown in this figure introduces a noise and a bias due to truncation.

Considering N as the noise and B as the bias, than the error introduced in the system due to truncation in each arithmetic operation is given by the difference between the global expression for the forces without the noise and bias introduction and the expression including these factors.

The expanded flow diagram given Figure 6-14 shows how the noise and bias stages are introduced in the system due to the chosen fixed point format and the truncations at the end of each arithmetic operation (for the x-coordinate only, as the y coordinate is analogous). E.g. after the first subtraction the result will look as given in Eq. 6-20.

$$E\{1^{st}\ stage\} = x_1 - x_2 + N + B \qquad \textit{Eq. 6-20}$$

If all the stages are considered and the expression for the forces without error introduction and the expression of the forces with noise and bias are subtracted than an expression for the error introduced in the forces update unit can be found. This expression, considering only the noise (N) is given in Eq. 6-21.

$$\begin{aligned} E\{P4\} = & \left(N\ \cos(\theta)^2 TDELSTIFN\right) + \left(N\ \sin(\theta) TDELSTIFN\ \cos(\theta)\right) + \left(2N\ TDEL STIFN\ \cos(\theta)\right) \\ & + \left(N\ STIFN\ \cos(\theta)\right) + \left(N\ \cos(\theta)\right) + \left(N\ \sin(\theta)^2 TDEL\ STIFS\right) + \left(N\ \cos(\theta) TDEL\ STIFS\ \sin(\theta)\right) \\ & + \left(5N\ TDEL\ STIFS\ \sin(\theta)\right) + \left(N\ STIFS\ \sin(\theta)\right) + \left(N\ STIFS\ \sin(\theta)\right) + \left(N\ \sin(\theta)\right) + 4N \end{aligned}$$

Eq. 6-21

It is interesting to observe that the error introduced in the system is almost halved in the case that the particles are in contact completely horizontally or vertically, as the sine or cosine would than be respectively 1 or 0. (Note that the whole process of how this expression has been derived has not been included as it involves a large amount of mathematical derivations)

Eq. 6-22 shows the expression if only the bias is considered.

$$\begin{aligned} E\{P4\} = & \left(B\ \cos(\theta)^2 TDEL\ STIFN\right) + \left(B\ \sin(\theta) TDEL\ STIFN\ \cos(\theta)\right) + \left(2B\ TDEL\ STIFN\ \cos(\theta)\right) \\ & + \left(B\ STIFN\ \cos(\theta)\right) + \left(B\ \cos(\theta)\right) + \left(B\ \sin(\theta)^2 TDEL\ STIFS\right) + \left(B\ \cos(\theta)\ TDEL\ STIFS\ \sin(\theta)\right) \\ & + \left(5B\ TDEL\ STIFS\ \sin(\theta)\right) + \left(B\ STIFS\ \sin(\theta)\right) + \left(B\ STIFS\ \sin(\theta)\right) + \left(B\ \sin(\theta)\right) + 4B \end{aligned}$$

Eq. 6-22

These errors are introduced to the system in every cycle so that the total error after n cycles will be equal to the sum of these errors after each cycle, as shown in Eq. 6-23.

$$E\{P4\} = \sum \left(N \ \cos(\theta)^2 TDELSTIFN\right) + \left(N \ \sin(\theta) TDELSTIFN \cos(\theta)\right) + \left(2N \ TDELSTIFN \cos(\theta)\right)$$
$$+ \left(N \ STIFN \ \cos(\theta)\right) + \left(N \ \cos(\theta)\right) + \left(N \ \sin(\theta)^2 TDEL \ STIFS\right) + \left(N \ \cos(\theta) TDEL \ STIFS \ \sin(\theta)\right)$$
$$+ \left(5N \ TDEL \ STIFS \ \sin(\theta)\right) + \left(N \ STIFS \ \sin(\theta)\right) + \left(N \ STIFS \ \sin(\theta)\right) + \left(N \ \sin(\theta)\right) + 4N$$

Eq. 6-23

These two expressions give a mathematical expression of the error introduced in the system due to truncation after each arithmetic operation and will be used in the next section to compare some real results with this model.

6.6.3 Position Update unit

The position update unit involves a smaller amount of arithmetic operations and a much smaller number of pipeline stages (only 7), which reduces the possible error generation and its propagation in the system.

Figure 6-15 shows the expanded flow diagram of the pipeline in the position update unit that computes the new x coordinate of the particles. Again as show in the previous section, each arithmetic operation in the pipeline inserts a noise and bias error in the system.

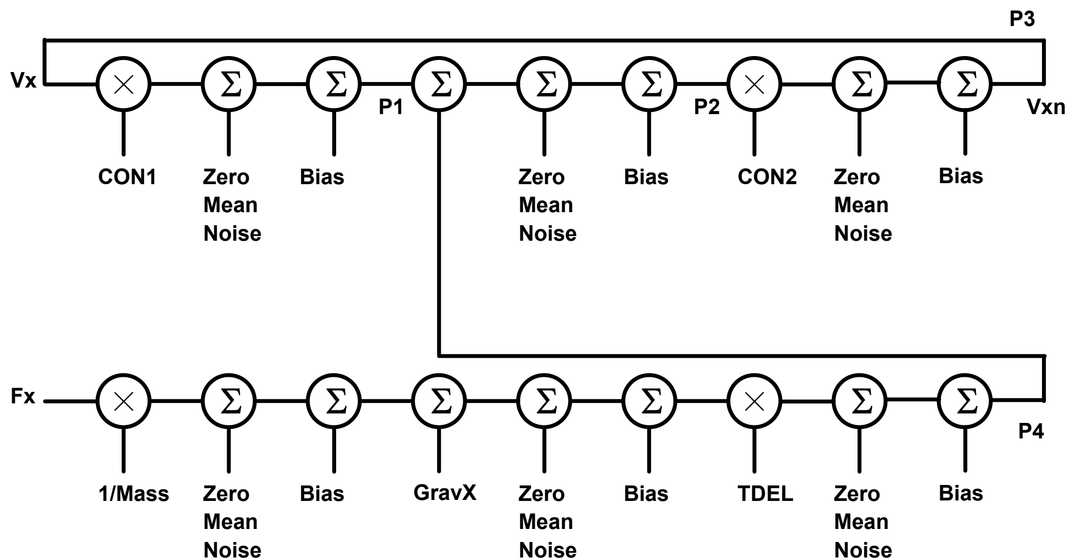


Figure 6-15 *Expanded flow graph for the x coordinate pipeline in the position update unit*

Considering *N* as the added noise and *B* as the bias, then the value obtained for v_x will be given by Eq. 6-24 and Eq. 6-25, where *E{P4}* is the expression of the value in point P4 of

Figure 6-15 and *E{P3}* the value at the end of the pipeline (which correspond to the value of the x-coordinate).

$$E\{P4\} = \left(\left(\frac{Fx}{mass} + N + B\right) + GravX + N + B\right)TDEL + N + B$$ **Eq. 6-24**

$$E\{P3\} = ((VxCON1 + N + B) + P4 + N + B)CON2 + N + B$$ **Eq. 6-25**

Eq. 6-26 shows the result of inserting Eq. 6-24 in Eq. 6-25.

$$E\{P3\} = \left((VxCON1+N+B)+\left(\left(\frac{Fx}{mass}+N+B\right)+GravX+N+B\right)TDEL+N+B+N+B\right)CON2+N+B$$

Eq. 6-26

As from a time instant viewpoint $Vx(t)$ is used to compute $Vx(t+1)$ the central limit theorem, explained in section 6.5.1.2, will start to apply for Eq. 6-4. So stripping out the noise component to just look at the mean value will derive in Eq. 6-27.

$$Vx(t+1) = \left((Vx(t)CON1+N+B)+\left(\left(\frac{Fx}{mass}+N+B\right)+GravX+N+B\right).TDEL+N+B+N+B\right)CON2+N+B$$

Eq. 6-27

On the other hand stripping out the noise component to just look at the mean value will derive in Eq.6-28.

$$Vx(t+1) = \left((Vx(t)CON1 + B) + \left(\left(\frac{Fx}{mass} + B\right) + GravX + B\right).TDEL + B + B\right)CON2 + B$$ **Eq.6-28**

Obviously for the error free value should be:

$$Vx(t+1) = \left(\left(Vx(t)CON1 \right) + \left(\left(\frac{Fx}{mass} \right) + GravX \right) .TDEL \right) CON2 \qquad \textbf{\textit{Eq. 6-29}}$$

As the F_x is also computed in the same way, as explained in the previous section, it also contributes to the error introduced when computing the new x coordinate as shown in Eq. 6-21. Thus the magnitude of the error for the new coordinate is equal to the difference between the error free and the expression which considers the error ($x_{error\ free} - x_{error}$), as shown in Eq. 6-30.

$$x_{Error} = \left(\frac{F_x + Error_{Fx}}{m} TDEL^2 CON2 \right) + \left(2N\ TDEL^2 CON2 \right) + \left(3N\ CON2\ TDEL \right) + \left(N\ TDEL \right) + \left(2N \right) \qquad \textbf{\textit{Eq. 6-30}}$$

The first term that includes the error due to the error induced in the forces update unit, and will be equal to 0 if the particle has no other particles in contact with it.

Considering the variance alone and taken the square root of the expression given in Eq. 6-31, would give a 65 % in confidence levels, as we are assuming a normal distribution.

$$x_{noise} = \left(\frac{F_x + Error_{Fx}}{m} TDEL^2 CON2 \right) + \left(2B\ TDEL^2 CON2 \right) + \left(3B\ CON2\ TDEL \right) + \left(B\ TDEL \right) + \left(2B \right) \qquad \textbf{\textit{Eq. 6-31}}$$

6.6.4 System error accumulation

In order to validate the previously discussed error analysis, a simulation was generated and run in software and hardware. This system was simulated one time step at a time and the differences between the software and the hardware results were monitored and stored. Figure 6-16 shows the simulator window that reports the values of the hardware and

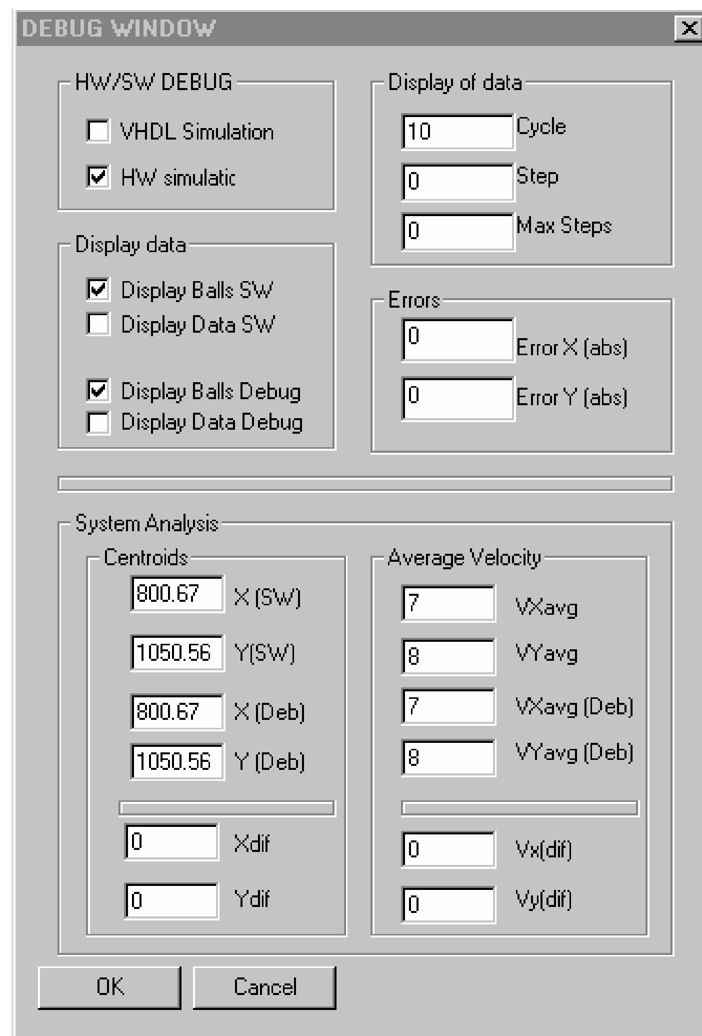


Figure 6-16 *SW window to measure the difference between the SW and the HW simulation*

software centroid, average velocity and their x and y-coordinate deviation.

Figure 6-17 shows how the mean absolute error in particle position grows with each time step. The error is taken to be the difference between the particle positions produced by the hardware and software simulations. The blue line corresponds to the theoretical error

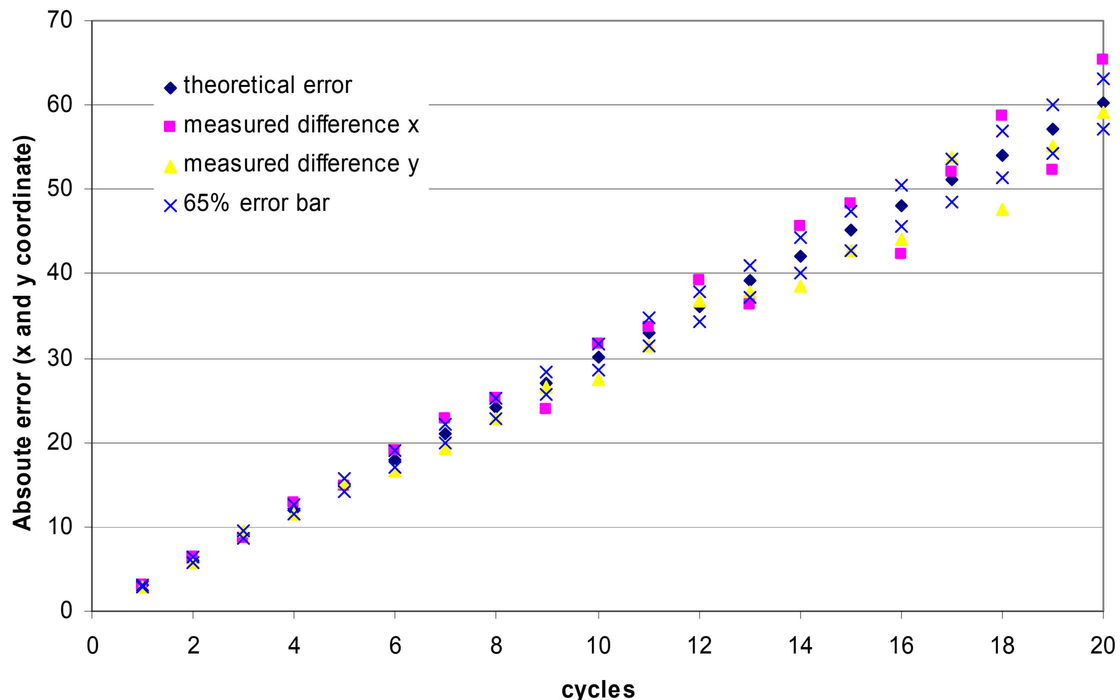


Figure 6-17 *Error accumulation in the hardware implementation compared to the software implementation for the x and y coordinates*

introduction and predicts a linear error growth. The purple and black lines show the real error introduction, which is distributed around the predicted error values and grows in a similar linear way. As we assumed a normal distribution 65 % of the error introduction resides between one standard deviation of the mean error, which is shown in the graph as blue crosses.

This result shows that the predicted analytical error study matches the results obtained in the arithmetic operation units (position and forces units).

6.7 Comparison of Bulk Errors in Software and Hardware

The previous section discussed the build up of errors in the positions of individual particles. In fact, the behaviour of individual particles is rarely of interest in DEM

simulations. (If it were, then the standard practice of initialising particles with a randomised initial position and velocity would be unacceptable.) Instead, the macroscopic properties of the bulk are the issue of interest. If the errors in particle positions are not systematically biased, then the error incurred in each particle will in general be cancelled out by error in other particles, thus giving an overall behaviour of the bulk that is accurately modelled, even when the behaviour of the individual particles is not.

The statistical analysis and its experimental verification in the previous section has shown on the other hand that there is and biased error introduction in the system due to truncation, which makes the hardware values be always smaller than expected. This is a trade-off that had to be assumed when designing this prototype and newer FPGAs that allow the implementation of this hardware architecture with higher wordlength will minimize this effect. The next sections will also discuss and show that this biased error introduction does not have any effect on the stability of the system, which behaves in the same qualitative manner as the software simulator.

In order to check if the bulk behaviour of a particle assembly behaves equally in the software and in the hardware implementations, a set of bulk measures computed. These were:

- The system energy
- The average particle speed
- The location of system centroid

A simulation was set up in order to compare the behaviour of these bulk measures between software and hardware. The initial parameters are shown in Figure 6-18. The results are presented in the following sub-sections.

```
START 150.00 150.00 7 0
RADIUS 2.00
AUTO 0.00 150.00 0.00 150.00 50 100 0 1
SHEARSTIFF 2.00
NORMSTIFF 500.00
```

```
DENSITY 1.00
    FRICTION 0.00
    DAMPING 0.00 0.00 0.00 0.00
    COHESION 500.00
    XGRAVITY 0.00
    YGRAVITY 0.00
    FRACTION 0.08
    CYCLE 1
```

Figure 6-18 *Initialisation file for the simulations*

6.7.1 System Energy

The contributions to system energy are listed below:

1. **Kinetic Energy:** Eq. 6-32 gives the expression to compute the kinetic energy of a single particle.

$$E_{kinetic} = \frac{1}{2}mv^2 + \frac{1}{2}I\theta^2 \qquad \textbf{Eq. 6-32}$$

This is summed across the entire assembly to give the total kinetic energy

2. **Potential Energy:** When two particles come into contact, their velocities diminish and they overlap. The repulsive force F between them is proportional to the overlap Δn. In terms of energy, this means that part of the kinetic energy of the system is converted to potential energy stored in the contact. When the contact between the particles ceases, the potential energy is converted back to kinetic energy again. Eq. 6-33 shows the expression needed to compute the potential energy of one particle, where k represents the stiffness.

$$E_{Potential} = \frac{1}{2}k\Delta n^2 = \frac{1}{2}F\Delta n \qquad \textbf{Eq. 6-33}$$

with $F = k\Delta n$

Figure 6-19 shows how the contacts between particles are modelled by a normally directed spring and dashpot, and by a spring–dashpot-slider assembly in the tangential direction. This model is explained further in [10].

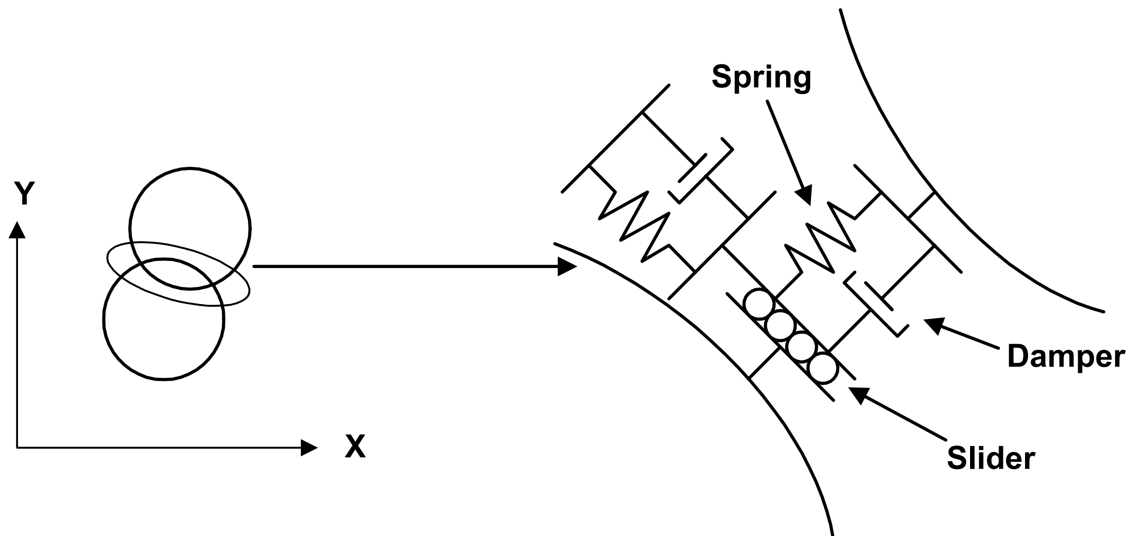


Figure 6-19 *Balls' contact model*

The total energy (neglecting potential energy due to gravity) of the system at every instant of time is given by the sum of both energies, as shown in Eq. 6-34.

$$E_{Total}(t) = E_{Kinetic}(t) + E_{potential}(t) \qquad \textbf{\textit{Eq. 6-34}}$$

In order to monitor the energy in the system after every step, an option was built into the software simulator to trace the evolution of the energy components in the software and hardware simulations. In debugging mode, both hardware and software simulations are run in parallel, thus providing the instantaneous values of the various energies for each time step, as shown in Figure 6-20.

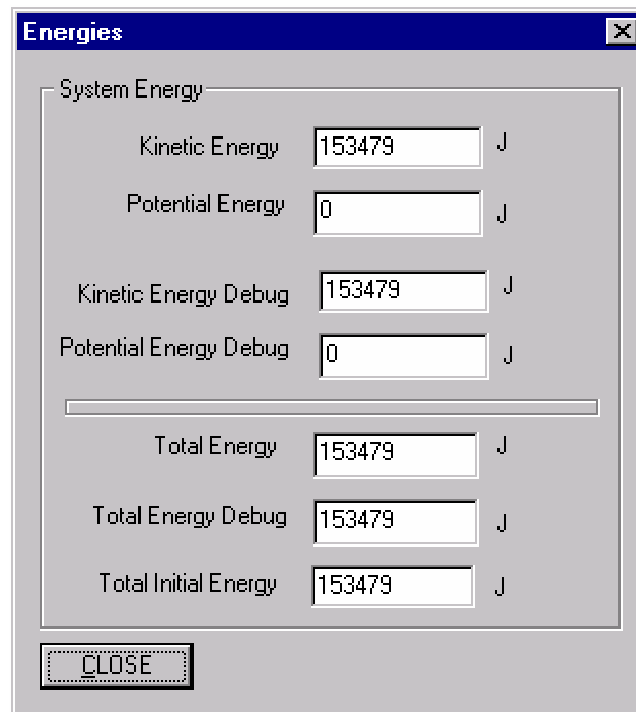


Figure 6-20 *Energy window once the software and hardware system have been generated*

Figure 6-21 shows how a DEM system with damping dissipates energy in the software and in the hardware version. The hardware version dissipates the energy a little faster, but its qualitative behaviour is exactly the same as the software version.

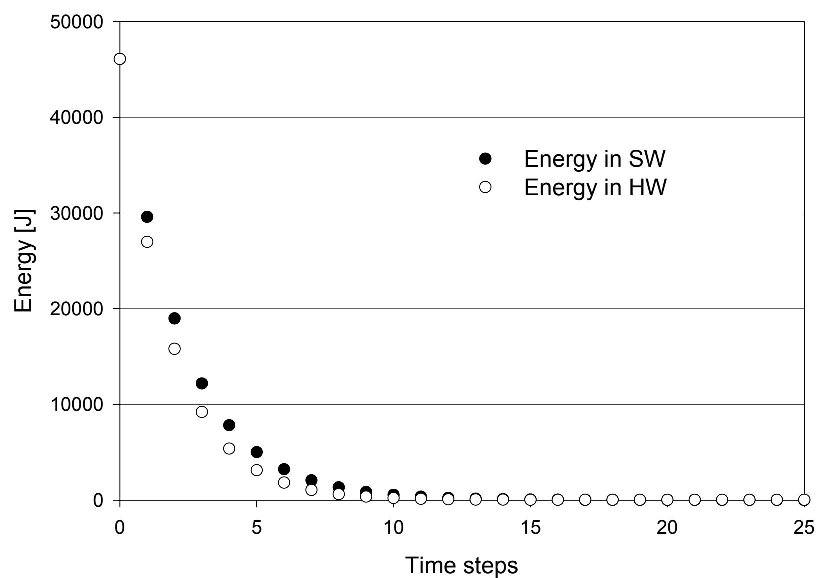


Figure 6-21 *System's Energy progression with damping for the software and the hardware implementation*

During testing, it was discovered that the numerical integration scheme used in the DEM is intrinsically energy dissipating if any contacts are established between particles. By contrast, a real system should not lose any energy if there is no damping and no sliding in the system. However, this is not the case for the DEM using the trapezoidal time integration scheme; the system loses energy each time a contact is created and released. This result was unexpected, so the software was checked against a standard DEM code, Cundall and Strack's BALL [11], to check whether this was due to a bug in the software developed for this project, or whether this is a general property of DEM simulators. The behaviour of the standard code was identical to the behaviour of our software.

Figure 6-22 shows how the total system energy evolves without damping for the software and the hardware implementations. In this case, it can be seen that the energy dissipation is much slower. Once again, the hardware implementation dissipates energy slightly faster then the software, but the qualitative behaviour is similar.

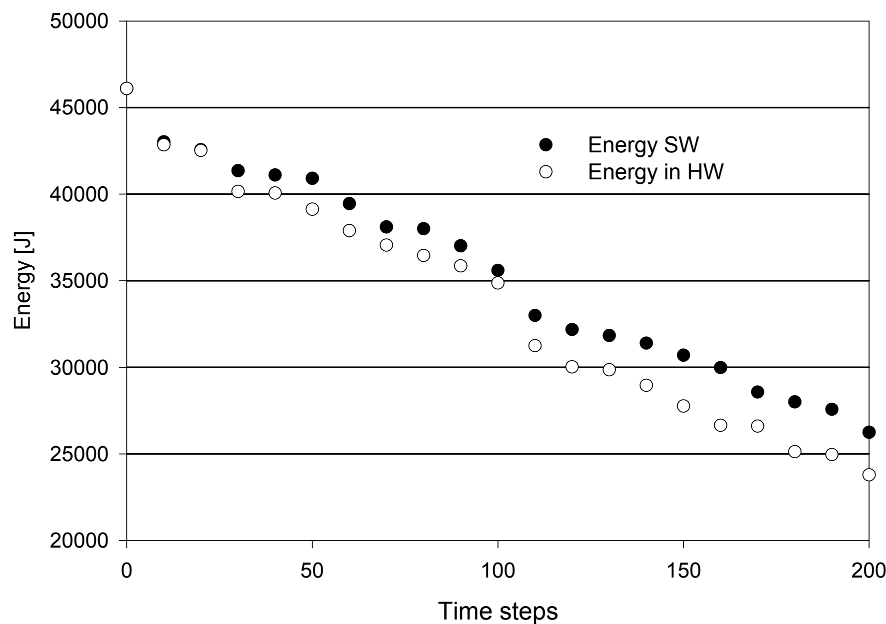


Figure 6-22 *Systems Energy progression without damping for the software and hardware implementation*

In order to test out the behaviour of a numerically unstable system, a simulation was performed where the time step was larger than the critical time step. For the software

simulator, the energy of the system increased rather than decreased. This is a sign of numerical instability. For the hardware version, there was an exponential increase in the number of overflows within the computational pipelines, and many of the results produced were nonsensical. These are symptoms of numerical instability; however, for the hardware version, the total energy of the system did not increase.

Another factor that affects the stability of the system are the overflows. As the parameters in the system were chosen in a way that overflows did appear extremely rarely this source of instability was not considered.

6.7.2 Assembly Centroid

Another measure of the bulk behaviour of the assembly is the centroid of the system. After every time step, the centroid of the system is computed and stored. The way the centroid is computed is given in Eq. 6-35 and Eq. 6-15.

$$x_{centroid} = \frac{\sum_{i=1}^{nr\ \ of\ \ balls} x_i}{nr\ \ of\ \ balls} \qquad \textbf{\textit{Eq. 6-35}}$$

$$y_{centroid} = \frac{\sum_{i=1}^{nr\ \ of\ \ balls} y_i}{nr\ \ of\ \ balls} \qquad \textbf{\textit{Eq. 6-36}}$$

The value of the centroid was computed both for the software and the hardware versions. The differences between the software and the hardware results are plotted in Figure 6-23 for a system using a ball radius of 2 units.

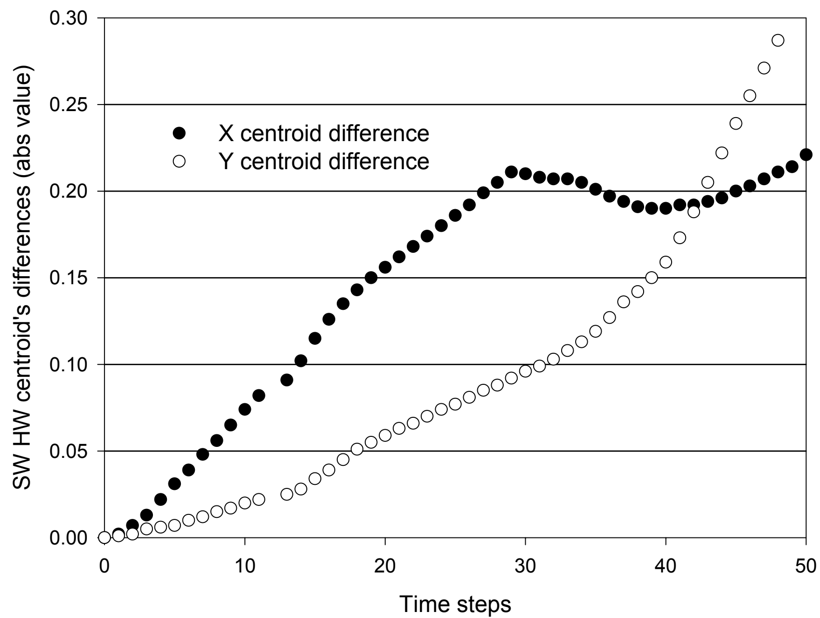


Figure 6-23 *Behaviour of the centroids difference between the software and the hardware implementation*

It can be seen from Figure 6-23 that the discrepancy steadily grows for about the first 30 time steps, but the size of the discrepancy is a small fraction of a ball radius. After about the 30th time step, a point of inflection occurs. This is because the simulator used a periodic boundary condition; when a ball exits the domain at one edge, it re-enters the domain at the opposite edge, as shown in Figure 6-24. The periodic boundary condition is used in order to prevent loss of particles from a finite sized domain.

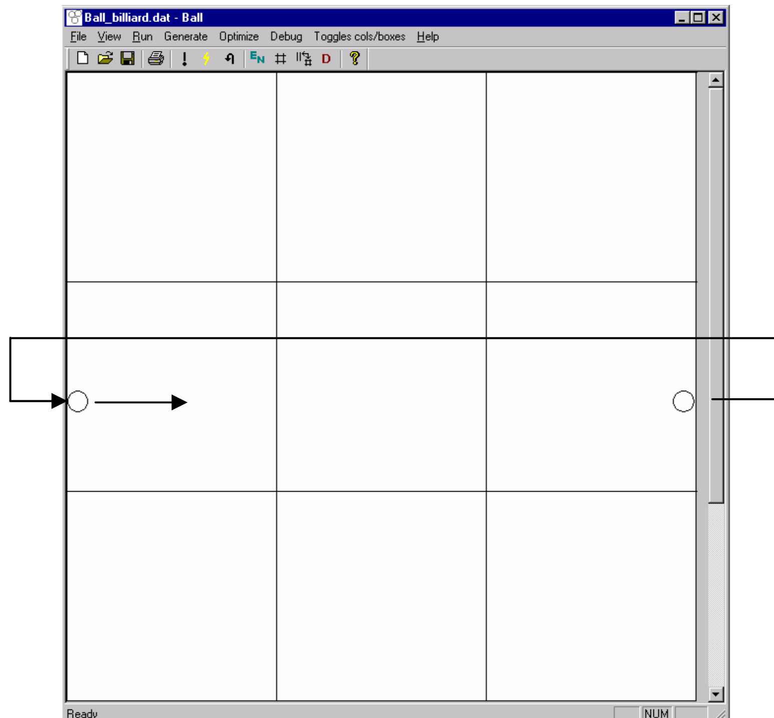


Figure 6-24 *Periodic domain example*

Two effects appear here due to the effect of chopping errors:

- The ball positions in the hardware version will lag behind those of the software version for positive displacements
- The ball position in the software version will lag behind those of the hardware version for negative displacements (this is due to the fact that chopping negative numbers given two's complement will generate a larger negative number)

This means that the particles simulated in software will wrap around the domain earlier than they will in hardware for positive displacements and will wrap around the domain later for negative displacements. This is the cause of the point of inflection seen in Figure 6-23. On average and for a large amount of particles both effects should cancel out.

6.7.3 Average Assembly Velocity

The last bulk measure to be considered is the average particle velocity of the assembly. The average velocity of both assemblies is monitored in the same way as the energies and

the location of the centroid. Figure 6-25 shows how the difference of average velocities in the x and y direction changes in every time step for a simulation where the initial velocities were initialised to a random number in the range –20 to +20 units. The error grows almost linearly, in a manner similar to the behaviour of the centroid before the period boundary problem appears.

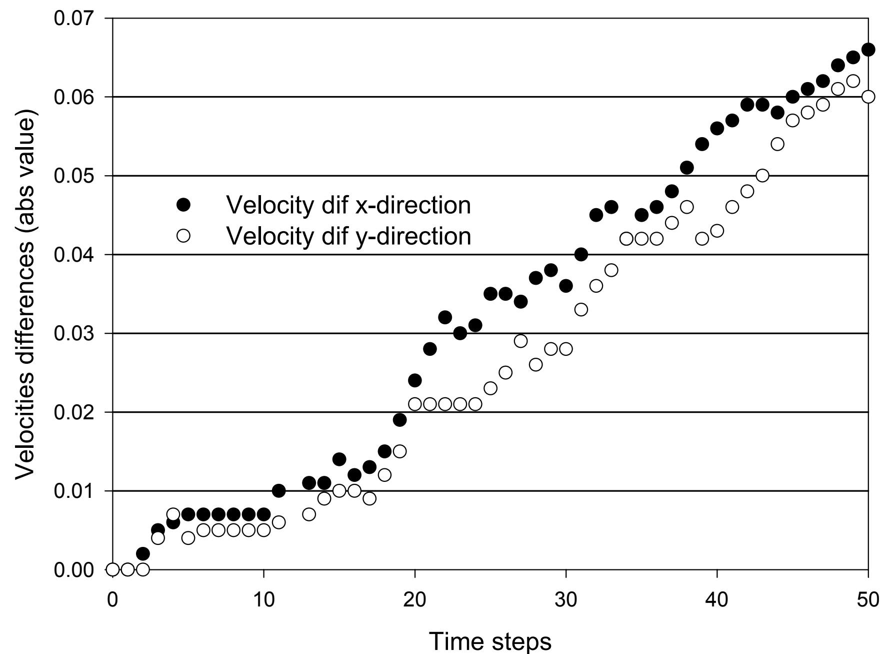


Figure 6-25 *Average velocity progression*

A similar conclusion as in the previous cases (sections 6.7.1 and 6.7.2) can be drawn. The chopping errors, introduced in the hardware version, are responsible for this growth in the difference. The error remains quite small, and the qualitative behaviour of the hardware and software simulators is close.

6.7.4 Comparison of Single Precision and Double Precision Software

So far it has been assumed that the 32-bit floating point software can be taken as a “correct” reference implementation against which the hardware results can be evaluated. In order to establish whether the software results are affected by the precision used, the simulator was modified to use double precision, and its results compared against the single precision results.

Two simulations were run, one for a 10,000 particle assembly and one for 20,000. Both were generated with random initial configurations, and ran for 1000 time steps. After 1000 time steps, the results of both simulation were compared. The discrepancies in particle positions were typically 1 part in a billion. Figure 6-26 the behaviour of the energy over the first 200 time steps.

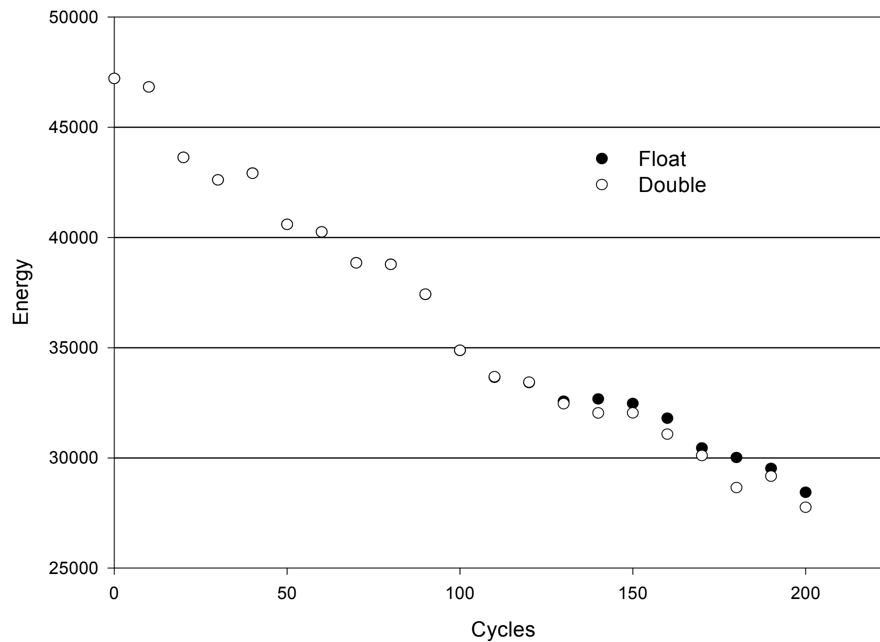


Figure 6-26 System energy for float and double SW simulation

As the importance of the DEM is the bulk behaviour of the assembly, and this was found to be almost identical, it can be concluded that in both simulations assemblies behaved equivalently.

The major difference was found in the simulation time needed. As the Pentium processor on which the software simulator runs is a 32-bit processor, use of double floating-point arithmetic increased simulation time by about 20%, as shown in Table 6–6.

***Table 6–6** Simulation time for 10 000 and 20 000 particles assemblies for 1000 cycles in the single and double precision floating-point arithmetic*

	t(single precision)	t(double precision)
10 000 particles	9 min 33 s	11 min 28 s
20 000 particles	21 min 17 s	25 min 51 s

6.8 Discussion

Two main aspects have to be considered when comparing the DEM implemented in software and hardware. The *effectiveness* of the design, i.e. the time it takes to run a simulation, and the *correctness* of the results.

The efficiency has been demonstrated for both hardware implementations (SIMD and MIMD) running a range of simulations. The SIMD design gave a speed-up of 5.6, and the MIMD design gave a speed-up of about a factor of 30. In order to give a conservative estimate of the speed-up, the highest possible stiffness that could be represented in 16 bits was selected. This has no effect on the speed of the hardware, but makes the software version very fast, because the contacts between particles are very brief, so at any one time the number of contacts is low. Also, the software did not use the same domain decomposition as the hardware; instead its domain decomposition was independently optimised to get the shortest possible run time for the software.

It could be objected that the PC used was rather old and does not represent the state of the art: it used a 1 GHz processor, whereas 3 GHz processors are appearing nowadays. However, the same argument can be made about the hardware board. It used a Virtex I FPGA; the same design implemented on a modern Virtex II would run at a clock speed about three times faster. The nature of the hardware pipelines means that trebling the FPGA clock speed really would cause the results to be produced three times faster. By contrast, trebling the clock speed of a microprocessor may not give a factor of three speed-up, due to the effect of cache misses and pipeline stalls. Also, the RC1000 board uses very slow memory (and this was the bottleneck limiting the speed of the MIMD design). The

static RAM runs at 40 MHz, and requires two clock cycles to perform a write cycle. Use of more modern memory could increase the I/O bandwidth by up to a factor of 10.

The parameters of the simulations were scaled so as to minimise the number of overflows that occurred. The only important errors introduced therefore into the system were those introduced by the chopping after each arithmetic operation. The error distribution functions, means and variances for the forces and position update units where obtained using the central limit theorem. The mean error would have been halved if a rounding scheme is used rather than chopping, however this would increase the expense of hardware.

The introduction of the chopping error was demonstrated by setting-up simulations and monitoring how the particle positions behaved. A 0.2% deviation was observed after each time step for each particle, which overlays with the predicted error introduction in the statistical analysis.

Usually when running a DEM simulation, the result of interest is the behaviour of the bulk, not the behaviour of every single particle. Some bulk measures were evaluated to analyse whether a particle assembly behaves equally in software and in hardware simulation:

- The total energy in the system showed a good behaviour, as both systems lost energy at a similar pace. The hardware system energy at a rate slightly higher than the software system due to the chopping error.
- The second bulk measure investigated was the centroid of the particle assembly. Chopping makes particles simulated in software wrap around the domain faster for when becoming larger and wrapping around the domain slower when exiting the domain from the lower end of the domain. These two effects cancel out when having a large enough number of particles.
- Lastly, the average system velocity was investigated. It shows a linear growth in the discrepancy, but gave reasonable agreement between the hardware and software simulations.

6.9 Summary and Conclusions

This chapter has presented an in depth analysis of the *effectiveness* and the *correctness* of the hardware and software implementations of the DEM.

The hardware implementations has shown a very good *effectiveness*, requiring 30 times less computation time to complete the same task as the optimised software version running on a fast PC.

The *correctness* of the results was also investigated. In order to fit the design on the available departmental FPGAs, which is a Xilinx™ XCV2000E, some compromises had to be made. 16-bit arithmetic with chopping (rather than rounding) was used. As a result of the use of 16-bit arithmetic, a linear error introduction was observed due to the loss of precision in the arithmetic operations. Applying a rounding scheme instead of truncating could halve the worst case error, but at some hardware costs, which in this case could not be afforded, as the design could only just fit onto the available FPGA. Introducing higher precision arithmetic (24 or 32-bits) would further reduce the error.

This chapter has therefore demonstrated the validity of the hardware implementation, its efficiency and correctness.

6.10 References

[1] Yates, R *,"Fixed-Point Arithmetic: An Introduction"*, Digital Audio Signal Procession, March 3, 2001.

[2] Wood, Alastair, *" Introduction to Numerical Analysis",* Addison-Wesley, Harlow, 1999

[3] Constantinidis, G.A., Cheung, P.Y.K, Luk, W., *"Optimum Wordlength Allocation"*, Field Programmable Custom Computing Machines (FCCM'02), Napa, California. 2002.

[4] Wadeker, S.A., Parker, A.C., *"Accuracy sensitive word-length selection for algorithm optimisation"*, Proc. International Conference on Computer Design, Austin, Texas, pp 54-61, October 1998

[5] Nayak, A., Haldar, M, Choudhary, A, Banerjee, P, *“Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs”*, Proc. Design Automation and Test in Europe, Munich, Germany, pp 722-728, 2001

[6] Constantinidis, G.A., Cheung, P.Y.K, Luk, W., *“The Multiple Wordlength Paradigm”*, Field Programmable Custom Computing Machines (FCCM’01), Napa, California. 2001

[7] Chang, M.L., Hauck, S.*”Précis: A Design-Time Precision Analysis Tool”,* Field Programmable Custom Computing Machines (FCCM’02), Napa, California. 2002

[8] Liu, B. *“Effect of finite word length on the accuracy of digital filters – a review”*, IEEE Trans. Circuit Theory, vol. CT-18, no6, pp. 670-677, 1971

[9] Constantinidis, G.A., Cheung, P.Y.K, Luk, W., *“The Multiple Wordlength Paradigm”*, Field Programmable Custom Computing Machines (FCCM’00), Napa, California. 2000.

[10] Rong, G.H., Negi, S. C, *“Simulation of Flow of Bulk Solids in Bins. Part 1: Model Development and Validation”,* Journal of agricultural Engineering 1995 62, 247-256

[11] Cundall P.A, O.D.L. Strack, “*A discrete numerical model for granular assemblies”*. Geotechnique 29, pp. 1-8, 1979.

CHAPTER 7

SUITABILITY OF THE HARDWARE DESIGN FOR A MORE COMPLEX DEM

7.1 Introduction

The hardware implementation described and analysed in the previous chapters worked only for a simplified version of the DEM, in order to allow the design fit onto the available FPGAs. The simplifications used were: domains contained no walls, all particles had the same radius and density, and all simulations were 2-dimensional. This chapter will examine how well a more sophisticated DEM implementation could be mapped on the high and low level parallelism hardware design.

In the real world, particle assemblies are kept in containers, and move around within containers and between them. These containers are represented in the DEM simulators [1][3] as walls with which the particles interact. These particles, in most cases, also have different radii and are 3-dimensional bodies.

These three factors were not considered when the hardware implementations were designed, as these factors would cost precious hardware resources and would result in a design that was too large to map onto the available FPGAs.

This chapter will analyse the alterations to the hardware implementation that would be necessary to treat these factors, and will estimate the hardware resources required.

7.2 Insertion of Walls

When walls are included in the system, the data flow of the DEM changes slightly:

- The walls can also move, thus their positions have to be updated
- Contact checking must also be performed between balls and walls
- Forces between balls and walls need to be computed.

Figure 7–1 shows the new flow diagram. Contact checking would still need to be performed for the balls in each domain, but the walls need to be considered too. Once the contact list has been established, then the inter-particle forces must be computed as well as the forces between particles and walls.

If the walls are not static, than their new positions must also be computed for the balls and the walls.

Finally the balls and walls need to be re-boxed if they have moved to a different domain.

As can be seen, three new sub-steps have been introduced into the computation path.

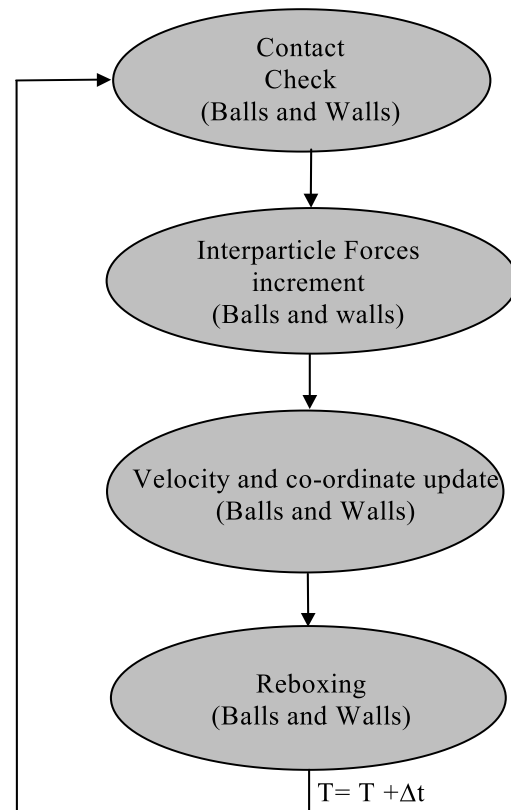


Figure 7–1 *Dataflow diagram for the DEM with walls*

This section shows how these steps can be incorporated into the design.

A wall can be defined as shown in Figure 7–2. The active side of the wall means the side facing the balls. The so-called shadowed part is the side of the wall where no balls should be found.

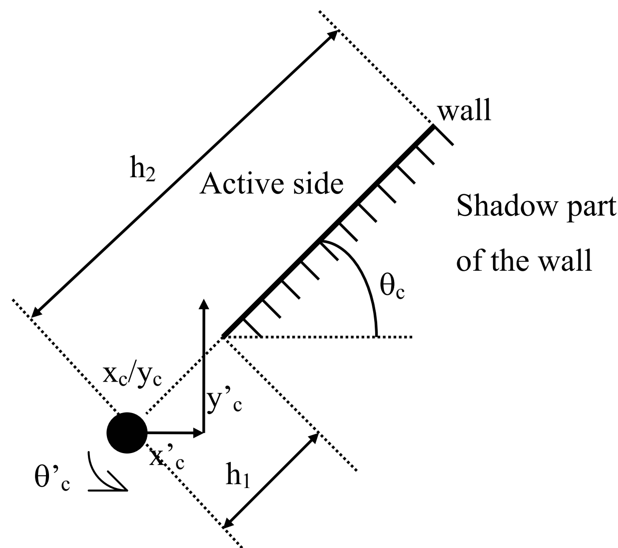


Figure 7–2 *Wall description*

The data needed to completely describe a wall is given below Table 7–1 [2].

Table 7–1 *Minimum set of parameters to describe a wall*

PARAMETER	DESCRIPTION
x_c	Initial x-coordinate of the wall
y_c	Initial y-coordinate of the wall
h_1	Distance from the initial coordinates to the beginning of the wall
h_2	Distance from the initial coordinate to the end of the wall
θ_c	Angle of the wall (in degrees) from the x-axis
$\dot{\theta}_c$	Angular velocity of the wall (in degrees per time unit)
$\dot{x}_c$	Velocity of the wall in the x-direction
$\dot{y}_c$	Velocity of the wall in the y-direction

Once the wall has been generated, a new set of data items are needed in order to describe its dynamic behaviour as shown in Table 7–2. These parameters can be deduced from the parameters given in Table 7–1, and are computed in order to facilitate the remainder of the computations involved with the walls. These parameters are also sufficient to completely describe a wall. The only difference between these and the previous parameters is that in Table 7–2 the wall can rotate from a different reference point.

Table 7–2 *Rest of parameters needed once the wall is interacting will the balls.*

x_1	Initial x-coordinate of the wall
y_1	Initial y-coordinate of the wall
x_2	Final x-coordinate of the wall
y_2	Final y-coordinated of the wall
F_{xsum}	Force on the wall in the x-direction
F_{ysum}	Force on the wall in the y-direction
M_{sum}	Moment of the wall
$\sin\theta_c$	Sine of the wall angle
$\cos\theta_c$	Cosine of the wall angle

7.2.1 Contact checking between Balls and Walls

The first step of the DEM that has been modified to treat walls is to check if there are any balls in contact with the walls. The following calculations have to be performed for that purpose. Based on Figure 7–3 the following equations are needed to check if a ball is in contact with a wall, assuming that the data shown in Figure 7–2 is available:

$$\frac{(x_2 - x_1)(y_b - y_1) - (y_2 - y_1)(x_b - x_1)}{L} < R \qquad \textbf{\textit{Eq. 7–1}}$$

$$where \quad L = h_2 - h_1 \qquad \textbf{\textit{Eq. 7–2}}$$

In order to check if the ball is in contact with the wall, the distance d (see Figure 7–3) needs to be computed and checked against the radius of the particle (R). The simplest way

to achieve this is to compute the area described by the parallelogram formed by the wall and a parallel line passing through the ball's centroid.

Knowing that the area of a parallelogram is given by Eq. 7–3, the distance between the

$$A = L \quad d$$ ***Eq. 7–3***

ball's centroid and the wall can be compared with the ball's radius in order to check if the ball and the wall are in contact.

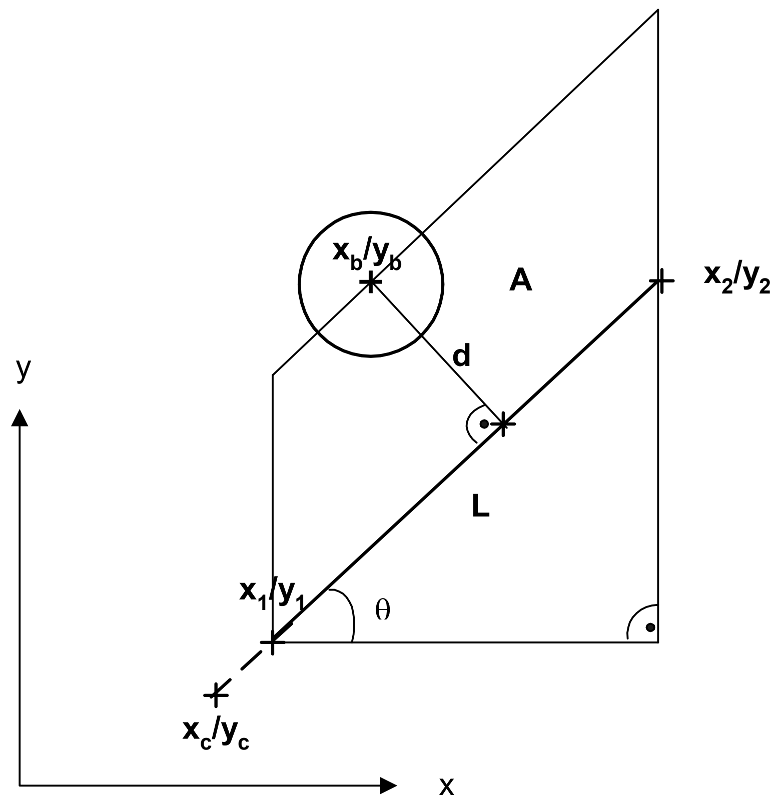


Figure 7–3 *Ball-Wall contact detection*

If d is smaller then the ball's radius, then the ball and the wall are in contact. Table 7–3 shows the number of arithmetic operations needed to solve Eq. 7–1, and Eq. 7–2.

Table 7–3 *Arithmetic operations needed the check for contacts between walls and balls*

ADD/SUB	MULTIPLICATION	DIVISION
6	2	1

The method described above requires many arithmetic operations, including one division, which do not map well to hardware.

The contact check method could be simplified if the walls were only allowed to be horizontal or vertical, as only a simple addition or subtraction would be needed. As shown in Figure 7–4 for a horizontal wall below, only one operation is needed.

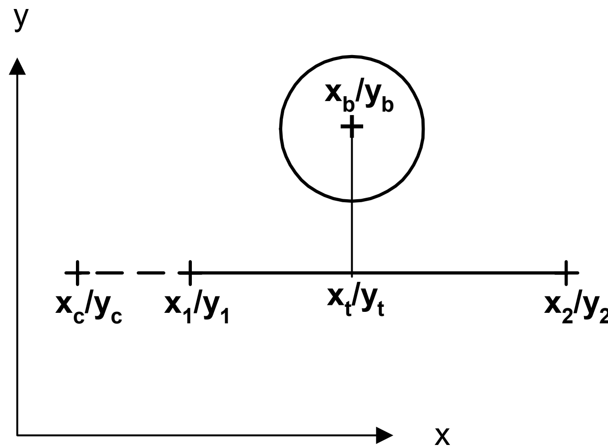


Figure 7–4 *Ball-Wall contact for Vertical/Horizontal walls*

For the case that the ball is being checked for contact with a horizontal wall below, only the comparison shown in Eq. 7–4 is needed.

$$if\,(y_b - rad \le y_1) \qquad \textbf{\textit{Eq. 7–4}}$$

In this case the number of arithmetic operations needed would decrease significantly: only 2 additions and 2 subtractions are needed, as shown in Table 7–4.

Table 7–4 *Number and types of arithmetic operations to check for contacts between balls and walls, only for vertical and horizontal walls.*

ADDITIONS	SUBTRACTIONS
2	2

7.2.2 Forces between Balls and Walls

In order to calculate the forces resulting from a contact between a ball and a wall, the equations shown in Figure 7–5 must be solved. The equations that are different from the simple DEM implemented in chapter 5 are highlighted.

$$X_{dif} = x_1 - x_t$$
$$Y_{dif} = y_1 - y_t$$
$$X_R = X_{dif} \times \cos(\theta) + Y_{dif} \sin(\theta)$$
$$Y_R = Y_{dif} \times \cos(\theta) + X_{dif} \sin(\theta)$$

$$XDR = v_{xwall} - v_{xball}$$
$$YDR = v_{ywall} - v_{yball}$$
$$DN = (YDR\cos(\theta) - XDR\sin(\theta) + X_R ang_{wall})\Delta t$$
$$DS = (XDR\cos(\theta) + YDR\sin(\theta) - radY)\Delta t$$

Different

$$DFN = DN \times k_{normstiff}$$
$$DFS = DS \times k_{shearstiff}$$
$$FNT = DFN + DFN \times BDT$$
$$FST = DFS + DFS \times BDT$$

$$FX = FNT\sin(\theta) - FST\cos(\theta)$$
$$FY = FST\sin(\theta) + FNT\cos(\theta)$$
$$M_{ball} = M_{oldwall} + FST \times rad$$
$$M_{wall} = M_{oldwall} - FN \times XR$$
$$F_{xball} = F_{xold} - F_x$$
$$F_{xwall} = F_{xwallold} + F_x$$
$$F_{yball} = F_{yballold} + F_y$$
$$F_{ywall} = F_{ywallold} - F_y$$

Different

Figure 7–5 *Equations to compute the forces between a wall and a ball*

Thankfully, most of these equations are the same as those needed to calculate the forces between two balls in contact. This means that only a small number of additional operations have to be implemented when walls are included.

Table 7–5 *Additional arithmetic operations needed to calculate the forces between a wall and a ball, compared to the arithmetic operations needed for forces between two balls.*

ADDITIONS/ SUBTRACTIONS	MULTIPLICATIONS	
11	Multipliers	KCM
	10	4

In total, only 11 extra additions, 10 extra multiplications and 4 extra constant coefficient multiplications are needed.

The next sub-section will analyse the influence of having a system with walls that move.

7.2.3 Wall Movement

As walls can also move, their position needs to be updated after every time step. This is done using the equations shown in Figure 7–6.

$$x_c = x_{oldc} + x'_c \Delta t$$
$$y_c = y_{oldc} + y'_c \Delta t$$
$$\theta_c = \theta_c + \theta'_c \Delta t$$
$$F_{xsum} = 0$$
$$F_{ysum} = 0$$
$$M_{sum} = 0$$
$$x_1 = x_c + h_1 \cos(\theta_c)$$
$$y_1 = y_c + h_1 \sin(\theta_c)$$
$$x_2 = x_c + h_2 \cos(\theta_c)$$
$$y_2 = y_c + h_2 \sin(\theta_c)$$

Figure 7–6 *Equations to compute the new position of the wall*

The total number of arithmetic operations needed to compute the new position of the wall is shown in Table 7–6. These arithmetic operations are not too hardware consuming, thus there should not be a problem implementing this on the FPGA.

***Table 7–6** Arithmetic operations needed to compute the new position of a wall*

<table>
<tr><td>ADDITIONS</td><td colspan="2">MULTIPLICATIONS</td><td>LUT (to compute the sin and cosine)</td></tr>
<tr><td rowspan="2">7</td><td>MUL</td><td>KCM</td><td rowspan="2">1</td></tr>
<tr><td>4</td><td>3</td></tr>
</table>

The cosine and sine can be computed using a look up table, once the wall's new angle is known.

7.2.4 Hardware Resources needed to accommodate Walls

The previous sections showed the number of additional arithmetic operations needed in order to simulate a system with walls. The question that arises is whether this could have been implemented with a state of the art FPGA, and if so, how many resources would this need.

The design implemented in chapter 5 took up 80 % of a Xilinx™ XCV2000E FPGA, which consists of 19 200 slices (each slice consists of two logic elements). This means that 15 360 slices were needed for this design. An estimate of the additional hardware needed in order to include walls in the design can be formed based on an estimate of the resource requirements of each of the constituent operations, shown in table Table 7–7.

***Table 7–7** Number of Xilinx slices needed to perform different arithmetic operations*

	ADD	MUL	KCM	DIV	SQRT
Slices	8	140	110	527	460
	Not pipelined	Core gen Pipelined	Core gen Pipelined	Core gen Pipelined	Pipelined

From this table, an estimate can be formed of the number of slices needed in order to deal with the walls. This is summarised in Table 7–8.

Table 7–8 *Number of Xilinx FPGA slices needed for the additional arithmetic operations to include walls*

	ADD	MUL	KCM	DIV
Operations	24	17	7	1
Slices	192	2380	770	527

The total additional number of slices needed for a design that deals with walls is 3 869, thus the overall number of slices is as given in Eq. 7–5 and Eq. 7–6.

$$slices_{total} = slices_{previous} + slices_{walls}$$ ***Eq. 7–5***

$$slices_{total} = 15360 + 3869 = 19220 \quad slices$$ ***Eq. 7–6***

Consideration of the Xilinx™ datasheets (assuming a maximum achievable utilisation of 80%), shows that a Virtex E FPGA (XCV 26000E) has the enough resources to deal with walls. It has 25 392 slices, while only 19 220 + 20% = 23 064 slices are needed.

In terms of extra memory resources needed to hold the walls in the internal FPGA memory, not much is needed, as every sub-domain will hold a maximum of three walls. The easiest way to describe a wall is to use the parameters given in Table 7–2. Therefore if only x_1, y_1, x_2, y_2, F_x, F_y, M, θ_c and $\dot{\theta}_c$ are considered to describe a wall ($\sin\theta_c$ and $\cos\theta_c$ are deduced from θ_c), Eq. 7–7 shows the total number of bits needed to describe a wall using 16 bit fixed point arithmetic.

$$Mem_{wall} = 9 \quad items \quad to \quad describe \quad a \quad wall \times 16 \quad bits = 144 \quad bits$$ ***Eq. 7–7***

As there can be a maximum of 3 walls in every column, therefore a maximum of 3x144 = 432 bits will be required to store three walls in the FPGAs internal memory. This should

not be challenging, since the XCV2600E has also 25 % more Block RAM than the XCV2000E used throughout this work.

7.2.5 Influence of the inclusion of Walls in the Overall Computing Time

The previous section dealt with the hardware resources needed to compute the different steps of the DEM with walls. This section will discuss how the inclusion of the walls would influence the computation time of the system. The three stages that must be considered are:

1. Contact check detection
2. Forces Update
3. Position update

7.2.5.1 Contact Detection

The contact detection between a ball and a wall could be computed in parallel with the contact detection between balls. Ball data is streamed through the contact check unit and this could be compared with the wall data applicable to this cell as well. This means that this unit would not cause any delay in terms of contact detection time.

7.2.5.2 Forces Update

After the forces due to contacts between the balls have been computed, the forces due to contacts between balls and walls are calculated. In order to save hardware resources, some of the arithmetic operations needed to compute the forces between balls can be re-used to compute the forces between balls and walls. This will of course have an influence in the computational time.

The time needed to compute these forces will depend on the number of particles that are in contact with the wall(s) in that sub-domain (see Eq. 7–8).

$$t(forces\quad walls) = \frac{wall\quad length\quad of\quad the\quad domain}{ball\quad diameter} + pipeline\quad latency \qquad \textbf{Eq. 7–8}$$

Depending on the orientation of the walls in the domain, there could be more balls in contact with the walls in one sub-domain than in others.

The depth of the pipeline for ball-wall calculations is the same as the one to compute the forces between balls, i.e. about 50 stages. The time needed to compute the forces between walls and balls will therefore not be significant in comparison with the time taken for ball-ball forces, since ball-ball contacts will be much more numerous than ball-wall contacts.

7.2.5.3 Position Update

The time needed to compute the new position of the wall will depend on the number of walls in the system and the depth of the pipeline, which in this case consists of only 6 stages.

$$t(position \quad walls) = number \quad of \quad walls + pipeline \quad latency \qquad \textbf{Eq. 7–9}$$

In a normal system the number of walls would be very small compared to the number of particles, which means that the total time required to update their positions will be minimal.

7.2.6 Discussion of the Inclusion of Walls on the Implementation

The inclusion of walls in the DEM simulation involves the need to re-design the hardware implementation. Sections 7.2.1, 7.2.2 and 7.2.3 show the amount of extra arithmetic operations needed to perform the contact check between balls and walls, forces between balls and walls, and to compute the new position of the moving wall respectively.

The time needed to compute these operations is given in sections 7.2.5.1, 7.2.5.2 and 7.2.5.3. The timing considerations of these operations are based on the assumption that large pipelines can be used in order to achieve one result every clock cycle. If the hardware resources are insufficient to allow this type of configuration, then other slower configurations can be used; the additional operations needed to compute the effects of the walls is small compared to the operations required to compute ball-ball interactions, so it is not critical to use a fast method for wall computations.

The conclusion of this section is that the insertion of walls would add some extra costs in terms of computing time and hardware resources, but this could easily be handled by the implementation. However, it should also be mentioned that the additional complexity of task scheduling makes the control unit more complex

7.3 Multiple Radii

The second aspect to take into account is the influence of particles with different radii. In principle, there could be some very large particles and some very small ones, with the result that a ball could have a potentially unlimited number of other balls in contact (see Figure 7–7). Therefore the software frontend to the hardware simulator should impose a maximum allowable ratio between the maximum and the minimum valid radii in order to avoid having a very large number of possible balls in contact, which could exhaust the FPGA's internal memory.

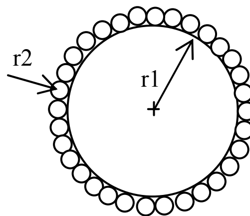


***Figure** 7–7 Balls in contact with different radii*

The changes to the arithmetic needed as a result of having balls of different radii will be discussed next.

7.3.1 Arithmetic Changes:

The arithmetic changes needed for this case are given below:

1. <u>Contact checks</u>

The contact check has to change as well. Instead of checking for a ball in the surrounding box of 4 x radius, contacts have to be checked for contacts in the surrounding box of

2 rad_1 +2 rad_2 (see Figure 7–8), and the box size has to be changed for each individual ball in the same cell.

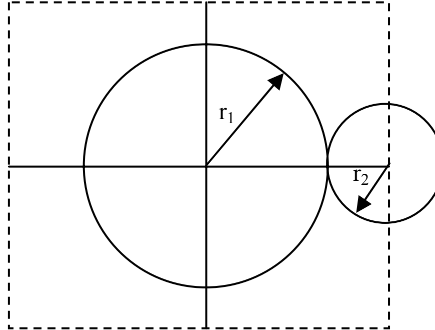


Figure 7–8 *Contact detection for particles of different radius*

2. **Force update:**

Multiplications are performed using the radii of the balls. These multipliers have to be changed to generic multipliers instead of *cheaper* constant coefficient multipliers. In total only two KCMs need to be changed to normal multipliers. As the force update unit is implemented as a large pipeline, the value of this radius needs to be stored along the pipeline in order to be applied at the correct time. Therefore a FIFO of 50 stages needs to be added as well, as the radius has to be available at the end to the forces pipeline, when it is requested again

3. **Position update:**

No multiplications by the radius are involved in the position update unit, but the masses and the moment of inertia of the different particles would depend on the radius. Therefore the 4 divisions involved in the positions update unit (which were transformed to constant coefficient multiplications by $1/mass$ and $1/I$) have to be computed in full. This means that 3 KCM must be transformed into dividers, which consume far more hardware resources than KCMs.

7.3.2 Data Format

Having balls of different radii also makes the fixed contact list scheme invalid, as there can be more than six balls in contact now. An alternative data structure to hold the contact list

could be to allocate a variable number of entire (256 bit) memory slots within the FPGA block RAM for contacts. Within a single memory location, a maximum of 16 contacts can be stored. Initially one memory slot is allocated to each ball. If during processing the number of contacts for a ball rises above a certain threshold x, then an additional memory slot will be assigned to this ball as the data is written back to the external memory (see Figure 7–9). This increases the amount of FPGA internal memory required in order to avoid overflow and loss of data.

This is one of many possible solutions to this problem, and is used to demonstrate that the proposed hardware implementation described in the previous chapters can cope with more complex DEM problems.

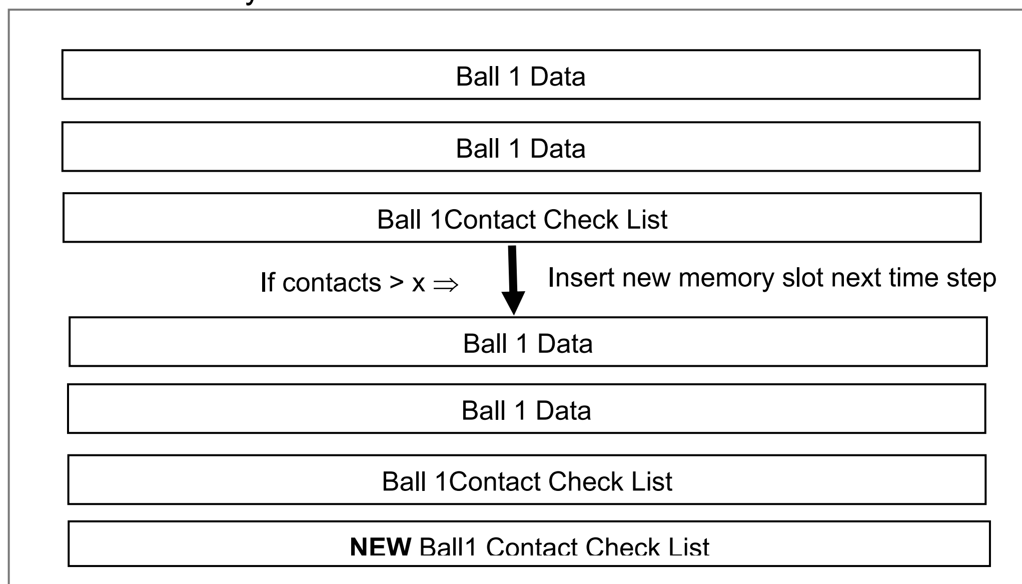


Figure 7–9 *Suggested Contact balls' data structure for systems with balls of different radii in the FPGA*

7.3.3 Discussion of the Use of Balls with different Radii

The use of balls of varying radii has a relatively modest effect on the contact check, force and position units. The main effect is to cause the replacement of the constant coefficient multipliers by normal multipliers or dividers.

A new data format is needed to allow a variable number of particle contacts. The fact that data is written back to the external memory allows the allocation and de-allocation of more memory space to each particle every time data is read in and out, thus allowing the use of a flexible data format.

The main impact of use of differing radii is felt in the increased complexity of the control unit and the interface unit that is needed to deal with the new data format, and also in the increased memory bandwidth requirements that are needed to handle the additional data.

7.4 3-Dimensions

Real world particles are 3-D bodies. In some cases, a 2-D simulation gives sufficient practical information for a specific problem, but in many other cases 3-D simulations are needed.

This section will discuss the changes needed in the hardware implementation in order to allow 3-D simulations to be performed. The discussion starts with a summary of the considerations involved in 3-D representation of particles and their memory requirements. This is followed by a discussion of the effect of using three dimensions for the main design units (contact check and the forces and position update units)

7.4.1 Ball description in 3-D

The first implication of a 3-D particle model is that more variables are required to describe each particle, since there will be six degrees of freedom rather than three. Table 7–9 shows the variables needed to describe a single ball for the 3-D case. All the variables in the 2-D case are needed, plus the variables for the z-axis and rotation in y- and z-plane.

Table 7–9 *Variables to describe a 3-D particle*

X	x-coordinate
Y	y-coordinate
Z	z-coordinate
V_x	Velocity in the x-direction
V_y	Velocity in the y-direction

V_z	Velocity in the z-direction
θ'_x	Angular velocity in the x-direction
θ'_y	Angular velocity in the y-direction
θ'_z	Angular velocity in the z-direction
F_x	Force in the x-direction
F_y	Force in the y-direction
F_z	Force in the z-direction
M_x	Moment in the x-direction
M_y	Moment in the y-direction
M_z	Moment in the z-direction
θ_x	Rotation in the x-direction
θ_y	Rotation in the y-direction
θ_z	Rotation in the z-direction

This means that by comparison with the 2-D case, nine further variables are needed. The maximum number of particles in contact also changes. If all particles have the same radius, then a maximum of twelve balls can be in contact with a ball. If particles of different radii are considered, then the maximum number of particles in contact with one will be determined by the ratio between the biggest and the smallest ball as in the 2-D case.

To hold the data to describe one single particle in 3-D will therefore need 512 bits (32 x 16 bits) instead of the 256 bits (16 x 16 bits) needed to describe a 2-D particles using 16-bit arithmetic, which involves twice as many memory bits per ball.

The next sub-sections will describe how each of the main units (contact checking, forces and position update) must be adapted to solve the 3-D problem.

7.4.2 Contact checking in 3-D

The contact checks for the 3-D case are similar to the 2-D case, but the z-component must now be included in the check. Instead of having to solve the computationally expensive equation of Eq. 7–10, it is sufficient to check for balls in the bounding cube, as with the 2-D case. Figure 7–10 shows the cube that must be checked for balls in contact.

$$\Delta n = R_1 + R_2 - \sqrt{x_{dif}^2 + y_{dif}^2 + z_{dif}^2} \geq 0 \qquad \textbf{\textit{Eq. 7–10}}$$

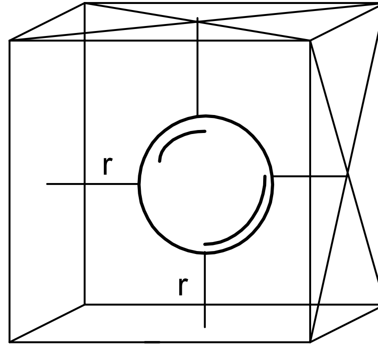


Figure 7–10 *3-D Contact checks*

The number of arithmetic operations needed to check for balls in the surrounding box is shown in Table 7–10**.**

Table 7–10 *Number of arithmetic operations needed for the contact check in 3-D*

	ADDITIONS	SUBTRACTIONS	MUL
Number of operations	**3**	**3**	**0**

If the unit is implemented as a pipeline, the time needed to perform this contact check would be exactly the same as for the 2-D case.

7.4.3 Forces Update Unit

The forces update unit for the 3-D case is the unit that requires the most arithmetic operations. It is also the unit that differs most from the 2-D case.

The equations needed to compute the interaction forces between two particles in contact in 3-D are shown in Figure 7–11.

$$x_{dif} = x_1 - x_2$$
$$y_{dif} = y_1 - y_2$$
$$z_{dif} = z_1 - z_2$$

$$D = \sqrt{x_{dif}^{2} + y_{dif}^{2} + z_{dif}^{2}}$$

$$\left.\begin{aligned} unity_x &= \frac{x_{dif}}{D} \\ unity_y &= \frac{y_{dif}}{D} \\ unity_z &= \frac{z_{dif}}{D} \end{aligned}\right\} \text{Unity vector}$$

$$ZMT = \frac{\Delta t}{x_{dif}^{2} + y_{dif}^{2} + z_{dif}^{2}}$$

$$v_{xdif} = v_{x2} - v_{x1}$$
$$v_{ydif} = v_{y2} - v_{y1}$$
$$v_{zdif} = v_{z2} - v_{z1}$$

$$UNDM = (v_{xdif} \times unity_x) + (v_{ydif} \times unity_y) + (v_{zdif} \times unity_z)$$

$$v_{n_x} = UNDM \times unity_x$$
$$v_{n_y} = UNDM \times unity_y$$
$$v_{n_z} = UNDM \times unity_z$$

$$v_{t_x} = v_{xdif} - v_{n_x}$$
$$v_{t_y} = v_{ydif} - v_{n_y}$$
$$v_{t_z} = v_{zdif} - v_{n_z}$$

$$\left.\begin{aligned} THDR_x &= (y_{dif} \times v_{t_z} - z_{dif} \times v_{t_y}) \times \Delta t \\ THDR_y &= (z_{dif} \times v_{t_x} - x_{dif} \times v_{t_z}) \times \Delta t \\ THDR_z &= (y_{dif} \times v_{t_y} - y_{dif} \times v_{t_x}) \times \Delta t \end{aligned}\right\} \text{Cross product}$$

$$\left.\begin{aligned} F_{s_x} &= (THDR_y \times F_{s_zold} + THDR_z \times F_{yold}) + F_{s_xold} \\ F_{s_y} &= (THDR_z \times F_{s_xold} + THDR_x \times F_{zold}) + F_{s_yold} \\ F_{s_z} &= (THDR_x \times F_{s_yold} + THDR_y \times F_{xold}) + F_{s_zold} \end{aligned}\right\} \text{Cross product}$$

$$F_x = \frac{1}{2}(F_{s_xold} \times F_{s_x})$$

$$F_y = \frac{1}{2}(F_{s_yold} \times F_{s_y})$$

$$F_z = \frac{1}{2}(F_{s_zold} \times F_{s_z})$$

$$ang_x = \theta_{xs2} \times r_2 + \theta_{xs1} \times r1$$
$$ang_y = \theta_{ys2} \times r_2 + \theta_{ys1} \times r1$$
$$ang_z = \theta_{zs2} \times r_2 + \theta_{zs1} \times r1$$

$$v_{s_x} = ang_y \times unity_z - ang_z \times unity_y$$
$$v_{s_y} = ang_z \times unity_x - ang_x \times unity_z$$
$$v_{s_z} = ang_x \times unity_y - ang_y \times unity_x$$

$$F_{s_x} = F_{s_x} - \left(v_{s_x} \times k_{shear.stiff} \times \Delta t\right)$$
$$F_{s_y} = F_{s_z} - \left(v_{s_y} \times k_{shear.stiff} \times \Delta t\right)$$
$$F_{s_z} = F_{s_z} - \left(v_{s_z} \times k_{shear.stiff} \times \Delta t\right)$$

$$FSM = \sqrt{F_{s_x}{}^2 + F_{s_y}{}^2 + F_{s_z}{}^2}$$

$$F_N = F_{Nold} - \left(UNDM \times k_{norm.stiff} \times \Delta t\right)$$
$$F_{s\max} = F_N \times FRICTION + COH$$

$$if\,(FSM > F_{s\max});balls_are_sliding$$

Shear Forces:

$$F_{s_x} = F_{s_xold} \times \frac{F_{s\max}}{FSM}$$
$$F_{s_y} = F_{s_yold} \times \frac{F_{s\max}}{FSM}$$
$$F_{s_z} = F_{s_zold} \times \frac{F_{s\max}}{FSM}$$

Moments:

$$M_x = F_{s_y} \times unity_z - F_{s_z} \times unity_y$$
$$M_y = F_{s_z} \times unity_x - F_{s_x} \times unity_z$$
$$M_z = F_{s_x} \times unity_y - F_{s_y} \times unity_x$$

Normal forces:

$$F_{NR_X} = F_N \times unity_x + F_{s_x}$$
$$F_{NR_Y} = F_N \times unity_y + F_{s_y}$$
$$F_{NR_Z} = F_N \times unity_z + F_{s_z}$$

New F_x, F_y, F_z forces:

$$F_{x1} = F_{x1old} + F_{NR_X}$$
$$F_{y1} = F_{y1old} + F_{NR_Y}$$
$$F_{z1} = F_{z1old} + F_{NR_Z}$$
$$F_{x2} = F_{x2old} + F_{NR_X}$$
$$F_{y2} = F_{y21old} + F_{NR_Y}$$
$$F_{z2} = F_{z21old} + F_{NR_Z}$$

$$
\left.\begin{aligned}
M_{x1} &= M_{x1old} + M_x \times rad_1 \\
M_{y1} &= M_{y1old} + M_y \times rad_1 \\
M_{z1} &= M_{z1old} + M_z \times rad_1 \\
M_{x2} &= M_{x2old} + M_x \times rad_2 \\
M_{y2} &= M_{y2old} + M_y \times rad_2 \\
M_{z2} &= M_{z2old} + M_z \times rad_2
\end{aligned}\right\} \text{New } M_x, M_y, M_z \text{ moments}
$$

Figure 7–11 *Equations to compute the forces between two particles in 3-D*

The equations used to compute the forces between two particles are completely different from the 2-D equations (given in section 2.5.2). The number of equations is over three times greater than for the 2D case. The reason for this is that in order to decompose the interparticle forces into normal and shear forces at the contact point and than convert these back to x, y and z components the cross product needs to be performed.

Table 7–11 shows the type and total number of arithmetic operations that are needed to compute the inter-particle forces in the 3-D case with particles having the same radius.

Table 7–11 *Number of Arithmetic operations involved in the3-D forces update units for particles of the same radius.*

	ADD/SUB	MUL		DIV	SQRT
		MUL	KCM		
Forces	**55**	**48**	**21**	**5**	**2**

If all the arithmetic operations are organized as a large pipeline, one result will be obtained after every clock cycle. The time needed to compute the inter-particle forces will therefore be almost the same as for the 2-D case. (The only difference will be that the latency associated with filling the pipeline will increase from about 50 clock cycles to 70; this can be neglected if a large number of particles are used.)

7.4.4 Position Update unit

The equations required the new position of a particles in 3-D are given in Figure 7–12.

$$v_x = \left[v_{xold} \times CON1 + \left(\frac{F_x}{m} + GRAVX \right) \times \Delta t \right] \times CON2$$

$$v_y = \left[v_{yold} \times CON1 + \left(\frac{F_y}{m} + GRAVY \right) \times \Delta t \right] \times CON2$$

$$v_z = \left[v_{zold} \times CON1 + \left(\frac{F_z}{m} + GRAVZ \right) \times \Delta t \right] \times CON2 \quad \textbf{NEW}$$

$$\theta'_x = \left[\theta'_{xold} \times CON1 + \left(\frac{M_x}{I} \times \Delta t \right) \right] \times CON2$$

$$\theta'_y = \left[\theta'_{yold} \times CON1 + \left(\frac{M_y}{I} \times \Delta t \right) \right] \times CON2 \quad \textbf{NEW}$$

$$\theta'_z = \left[\theta'_{zyold} \times CON1 + \left(\frac{M_z}{I} \times \Delta t \right) \right] \times CON2 \quad \textbf{NEW}$$

$$x = x_{old} + v_x \times \Delta t$$

$$y = y_{old} + v_y \times \Delta t$$

$$z = z_{old} + v_z \times \Delta t \quad \textbf{NEW}$$

$$\theta_x = \theta_{xold} + \theta'_x \times \Delta t$$

$$\theta_y = \theta_{yold} + \theta'_y \times \Delta t \quad \textbf{NEW}$$

$$\theta_z = \theta_{zold} + \theta'_z \times \Delta t \quad \textbf{NEW}$$

Figure 7–12 *Equations for the 3-D Position Update*

The number of arithmetic operations involved in the position update unit (assuming all particles have the same radius) is shown in Table 7–12. If the particles have different radii, then the mass and the moment of inertia would not be constant and six of the KCMs would have to be replaced by dividers.

Table 7–12 *Number of arithmetic operations involved in the3-D position update units for particles of the same radius.*

ADDITIONS/ SUBTRACTIONS	MULTIPLICATIONS	
	MUL	KCM
15	0	30

The 3-D position update unit would take the same time as the 2-D unit, since the additional arithmetic operations required for 3-D can be computed in pipelines in parallel with the operations that are common between 2-D and 3-D. Instead of having 3 pipelines in parallel, there would be 6, with the same latency and throughput. This is shown graphically in Figure 7–13, where the 6 parallel pipelines are shown.

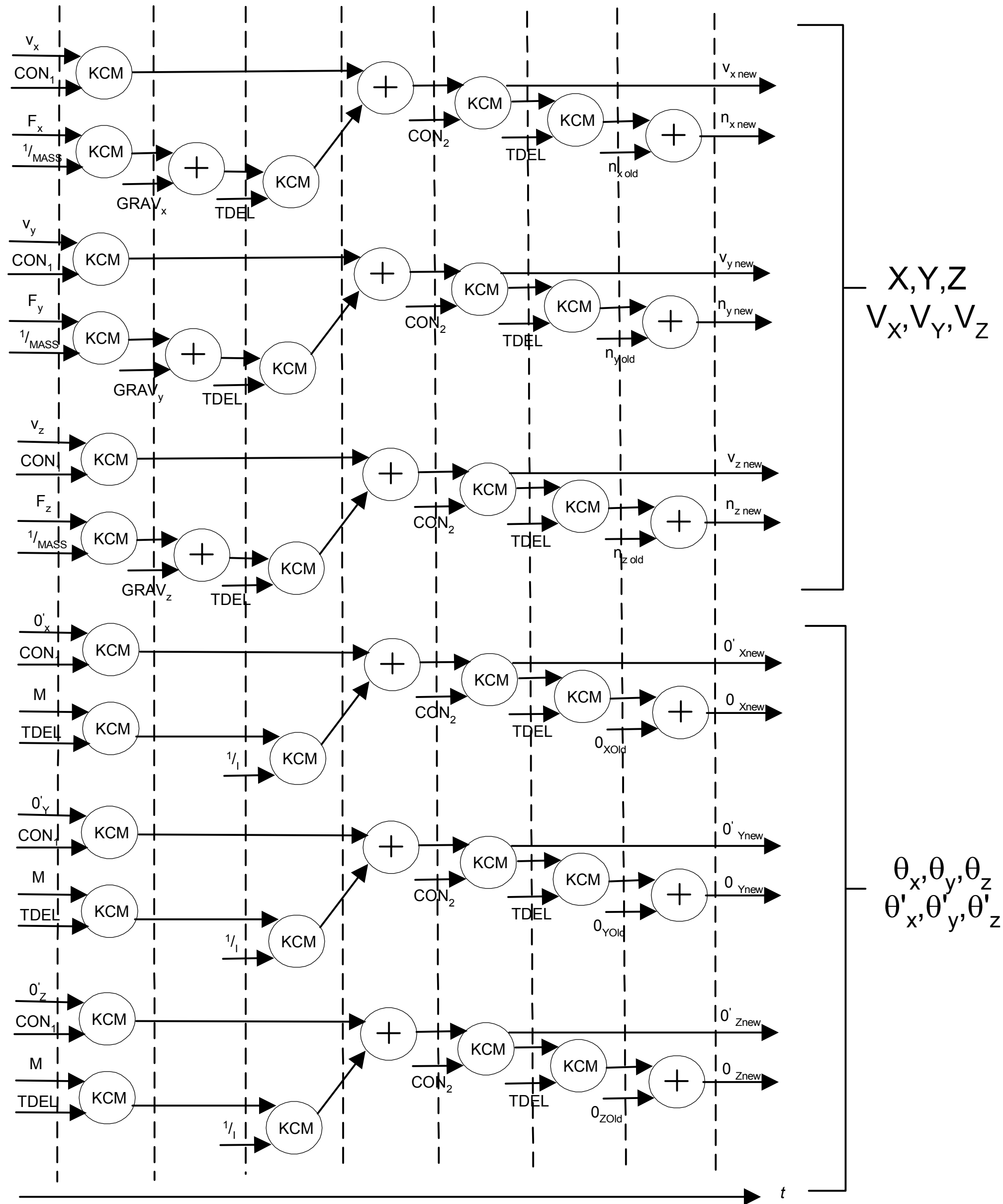


Figure 7–13 *Graphical Representations of the 3-D position update equations*

7.4.5 Arithmetic Operations Comparison between the 2-D v 3-D case

Table 7–13 summarises the different number of arithmetic operations needed by each of the tasks in the 2-D and in the 3-D case.

The contact check unit only needs 1 addition and 1 subtraction more, which is very cheap in terms of hardware resources. The position update unit needs twice as many resources in 3-D than in 2-D.

Table 7–13 *Comparison between the arithmetic operations for the 2-D and the 3-D case in each task*

	ADD/SUB	MUL		DIV	SQRT
3-D		MUL	KCM		
Cont check	**6**	**0**	**0**	**0**	**0**
Forces	**55**	**48**	**21**	**5**	**2**
Position	**16**	**0**	**30**	**0**	**0**
2-D					
Cont check	**4**	**0**	**0**	**0**	**0**
Forces	**18**	**10**	**8**	**1**	**0**
Position	**8**	**0**	**15**	**0**	**0**

This is logical as there are now 6 degrees of freedom (x, y , z, θ_x, ,θ_y,θ_z) instead of the 3 in the 2-D case (x, y,θ). The forces update unit is the one that is changed radically. The arithmetic operations needed here are over 3 times more than for the 2-D case. This means that a 3-D adaptation of the 2-D design would only be implementable on a very large FPGAs.

The internal FPGA memory also needs to be much larger. The extra parameters needed to describe the 3-D position and velocity, plus the contact forces (normal and shear) for each of the 12 possible contacts (rather than 6 for the 2-D case) gives a total of 864 bits to describe a single particle in 3-D for the case using 16-bit arithmetic. By comparison, the 2-D case needed only 432 bits to describe a particle and its contact forces, so the 3-D particle description requires 2 times more memory resources for each particle.

7.4.6 Hardware Resources needed to accommodate 3D balls

The previous sections have showed the total number of arithmetic operations needed in order to compute the contact checking, forces and position update for 3D balls instead of 2D. Table 7–14 shows the number of slices needed to accommodate the 3D arithmetic operations.

Table 7–14 *Slices needed to accommodate arithmetic operations for 3D*

	ADD	MUL	KCM	DIV	SQRT
Operations	77	48	51	5	2
Slices	616	6720	5610	2635	920

The total number of Xilinx™ Virtex slices needed to compute the contact checking, and the forces and position update units are 16 501. The implementation uses 15 360 slices. 35% of these slices (6,720 slices) were used to for the contact checking, position and motion update units and the other 45% (8,640 slices) were used for the rest of the circuit. The total number of slices therefore needed is given in Eq. 7–11 and Eq. 7–12.

$$slices_{total} = slices_{previos(no \quad cc+forces+position)} + slices_{3D(cc+forces+position)} \qquad \textbf{\textit{Eq. 7–11}}$$

$$total = 8{,}640 + 16{,}501 = 25{,}141 \quad slices \qquad \textbf{\textit{Eq. 7–12}}$$

The Xilinx FPGAs that might accommodate this size of design is the XCV3200E (3248 slices). If it is assumed that around 80% of the device resources could be used in order to allow it to route, then an extra 5,028 slices need to be added beyond those required to implement the design. The total number of slices would therefore need to be 30,169 (25,141+5,028) The XCV3200E should present no problems in this respect if a careful placement of the components is done.

Another option to free up some logic resources is to use a Virtex II FPGA, which contains embedded multipliers. This would free a considerable amount of logic resource, since multiplication is one of the most expensive arithmetic operations. In this case, the total number of slices needed, can be derived from Table 7–14 to give the result shown in Eq. 7–13

$$total = 8640 + 4171 = 12811 \quad slices + 99 \quad embedded \quad multipliers \qquad \textbf{\textit{Eq. 7–13}}$$

The XC2V3000 probably has insufficient resources for the 3D demands, as this FPGA has 14 336 slices and 96 multipliers, and the design would need 12.811 slices + 20% (2562 slices) to route. The next largest Virtex II FPGA is therefore needed, which is the XC2V4000, with 23,000 slices and 120 multipliers. This FPGA would have more than sufficient resources to fit the 3D equations.

In terms of memory resources needed, particles in 3D need more memory to be described, because:

- The Z axis component is added
- 3D balls of the same radius have a maximum of 12 balls in contact, while in the 2D case only 6 balls could be in contact.

$$Mem_{3Dball} = \left[30 \quad parameters \quad + 2 \times 12 (F_n and F_s) \right] \times 16 \quad bits = 864 \quad bits \qquad \textbf{Eq. 7–14}$$

$$Mem_{2Dball} = \left[15 \quad parameters \quad + 2 \times 6 (F_n and F_s) \right] \times 16 \quad bits = 432 \quad bits \qquad \textbf{Eq. 7–15}$$

As seen from Eq. 7–14 and Eq. 7–15, the total number of bits to describe a ball in 3D takes 2 times more memory than for the 2D case. From the Xilinx datasheets, it can be seen that the XC2V4000 has around 3 times more Block RAM than the XCV2000E, thus it should have sufficient memory to hold in 3D the same number of particles that a XCV2000E can handle in 2D.

7.4.7 Timing Comparisons between the 2-D and the 3-D case

The previous sections formed estimates of the computing time needed by the three main units. It has been shown that all three units should take almost the same amount of time as in the 2-D case for the hardware implementation. The operation that would be the bottleneck in the 3-D case would be the writing and reading data to and from the external memory. In order establish a baseline to assess the speed-up between hardware and

software, a 3-D software simulator was implemented. Figure 7–14 shows a screen-shot of 3-D simulation.

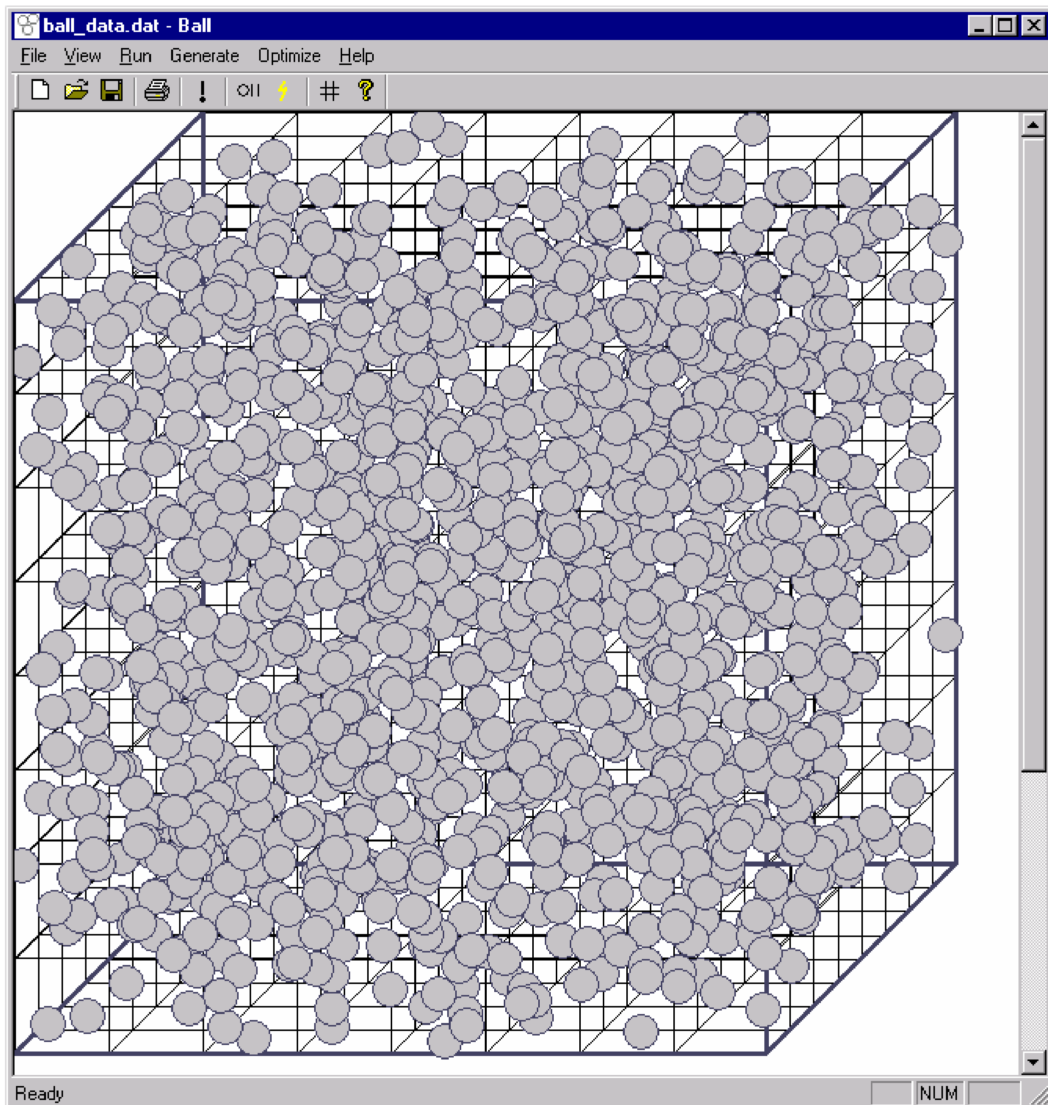


Figure 7–14 *Screen-shot of the 3-D software simulator*

This simulator was also optimised, in the same way as the 2-D simulator, in terms of data structure and grid size (section 3.3.1). A simulation was run for the 2-D and the 3-D simulator, with the same number of particles and same system parameters, for 1000 cycles. Table 7–15 and Figure 7–15 (in log scale) show the comparative runtime results of the 2-D and the 3-D simulators. It can be seen from the table that as the number of particles in the simulated assembly doubles/trebles, the runtime of the 2-D and the 3-D simulation approximately.

Table 7–15 *Runtime simulation results for 2-D and 3-D assemblies with the same properties*

NUMBER OF BALLS	1,250	2,500	5,000	10,000	20,000
T[s] 2-D	5	13	40	113	304
T[s] 3-D	20	40	105	309	1115

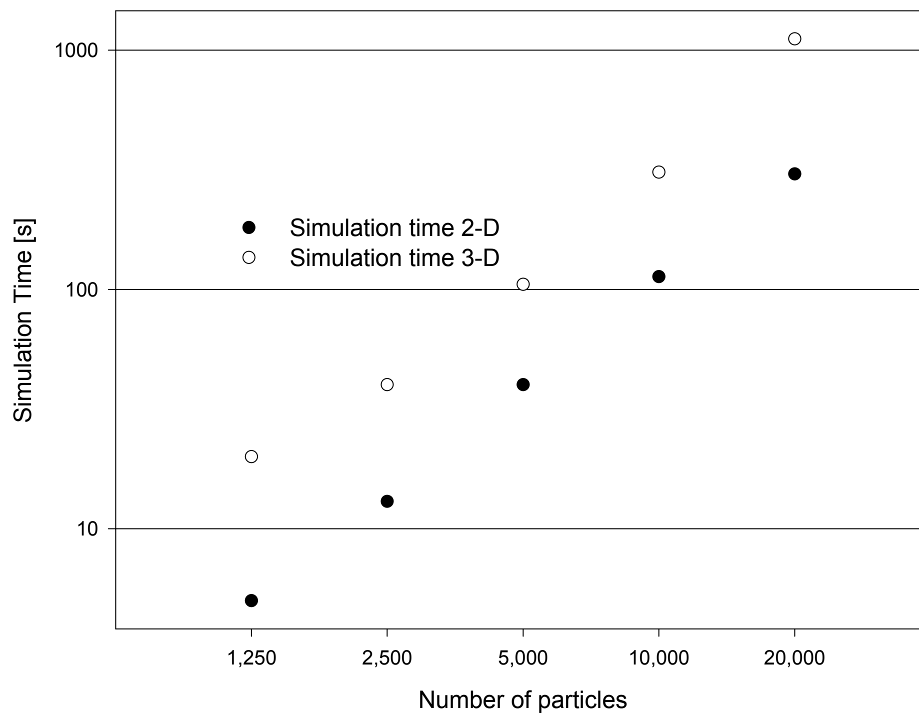


Figure 7–15 *Graphical representation of the Simulation time of 2-D and 3-D system*

Also, the 3-D software simulation takes on average 3 times longer than the 2-D simulation of the equivalent system, because:

- The SW simulator takes longer to check for contacts as a new degree of freedom is now introduced
- The force update unit's equation are computationally more intensive, taking longer to compute the force between two particles

- The position update units' equations also require more operations, which requires more CPU time.
- Re-boxing particles transitioning from one box to another takes also longer as the number of boxes is also larger.

In the 3-D case, it is assumed that the longest operation to perform will be the position update together with the read/write to the external memory, as in the 2-D case. The time taken for this is shown in Eq. 7–16.

$$t(slowest) = \left(N + \frac{N \times 33}{2} \right) \times nr \quad cycles \times \frac{1}{f[Hz]} \times nr \quad of \quad cols \qquad \textbf{Eq. 7–16}$$

Where f=30Mhz, N=100, *nr cycles* =1000 and *nr of cols* = N_{total}/100. The projected speeds-ups that could be achieved, based on the measured timing of the 3-D software simulation, and predicted timing of the hardware simulation are shown below.

Table 7–16 *Theoretical speed-up of the 3-D SW simulator and the theoretical HW design*

NUMBER OF BALLS	1,250	2,500	5,000	10,000	20,000
T[s] 3-D	20	40	105	309	1115
T[s] HW	0.76	41.46	2.92	5.83	11.66
Speed-up	26	27	36	53	95

These results show that with the architecture an even higher speed-up could be achieved for the 3-D case, at a penalty of extremely high hardware costs, as the arithmetic operations involved in the forces update unit are very numerous.

7.4.8 Discussion of the 3-D Implementation

This section has shown that it is possible to implement a 3-D DEM version based on the 2-D high and low level parallelism HW design. The fact that every arithmetic unit is implemented as a large pipeline allows the 3-D HW implementations arithmetic units to

finish at the same time as the 2-D. (However, there will be slight time penalty, as the bottleneck of the design is not the computational units, but the reading and writing data to the external memory; in 3-D every ball has 33 parameters to be written as opposed to 17 for the 2-D case.) Given that the 3-D software takes about 3 times longer than its 2-D equivalent, the speed-up achieved by the hardware should be considerably greater in 3-D than in 2-D.

This would of course be at the expenses of some HW costs. Approximately 3 times more HW resources are needed in comparison with the 2-D version.

The memory requirements also change for the 3-D case, as every particle needs 864 bits for its representation using 16-bit arithmetic, which means that that every particle in 3-D needs twice as many memory resources.

7.5 Summary and Conclusions

It has been shown in this chapter that the previously described HW implementations can be adapted for more complex DEM implementation, and that the only limitation is the available size of FPGAs.

Inclusion of walls into the design adds very little to the length of time taken by simulations. If only horizontal and vertical walls are used, very little additional hardware resource is needed. If the walls are allowed to be inclined, then there will be a significant cost in extra hardware. However, much of the computation for ball-wall contacts is the same as for ball-ball contacts, so much of the hardware can be re-used. This keeps the hardware penalty moderate.

The inclusion of particles of different radii would increase the hardware resources required due to the use of full multipliers or dividers rather than constant coefficient multipliers. However, the number of multipliers involved only 6, so the hardware penalty is not large. A more significant issue is that the data format must be changed, as the fixed number of contact slots allocated for particles of the same size is no longer valid. One possible data

format is present in section 7.3.2, which shows that it is possible to have a dynamic memory allocation scheme, as reading and writing data to and from the external memory in each time step allows the allocation and de-allocation of memory dynamically with no penalty.

Lastly, a 3-D implementation has been analysed. The contact check and the position update unit do not need too much extra hardware resource, but the force update unit would need a very large device in order to be able to have a fully pipelined structure generating one result per clock cycle. As the forces update unit is not the slowest unit, a different approach could be used which compromises on speed to save hardware. The time taken for a 3-D simulation in hardware would be the same as for the 2-D case, as one result would be obtained every clock cycle. By contrast, software becomes substantially slower on migrating from a 2-D to a 3-D approach. The speed-up achieved by hardware for 3-D simulations is therefore expected to be considerably higher than for the 2-D case.

The architecture that was actually implemented used a simplified DEM algorithm in order that a working prototype could be built in the available hardware. This chapter shows that the architecture can be modified in order to perform more complex DEM simulations.

7.6 References

[1] Cundall, P.A Strack, O.D.L. "*A discrete numerical model for granular assemblies*". Geotechnique 29, pp. 1-8, 1979.

[2] Cundall, P.A. Strack, O.D.L., *"The Distinct Element Method as a tool for research in Granular Media",* Report to the National Science Foundation Concerning NSF Grant ENG76-20711, pp 40-41, Minnesota, November 1978.

[3] Antonioletti M., *"A program to model granular media using the distinct element method"*, Available at http://www.epcc.ed.ac.uk/epcc-tec/JTAP/DEM.html

Chapter 8

SCALABLE AND ALTERNATIVE IMPLEMENTATIONS OF THE DEM

8.1 Introduction

A hardware implementation of the 2-D DEM on one single FPGA was shown to have a speed-up of a factor of 30 in chapter 6. This might be enough for some simulations, but in the case of entire engineering structures, millions of particles are involved. In order to model these structures an even higher speed-up is needed. The only way to achieve speed-ups that are orders of magnitude bigger than what has been achieved so far is to have multiple FPGAs working in parallel.

A very important aspect of the high and low level parallelism hardware implementation presented in this thesis is how well it will scale onto a multiple FPGA system. Will the speed-up be linear with the number of FPGAs, or will communication and synchronization overheads and load balancing problems degrade the overall FPGA speed-up considerably as in the multiprocessor systems?

The term "scalability" tries to express the benefit of solving large problems on a system with multiple processing elements. The algorithm is regarded as scalable if the efficiency is more or less constant as problem size and number of processors varies [1], where the efficiency is defined as the speed-up divided by the number of processing units.

The two most popular approaches to parallel computing using multiple processors are:

1. Distributed memory systems, where each of the processing units has its own memory.
2. Shared memory system, where each processing unit can access each of the memory banks

Multi-FPGA systems can be developed that are analogous to these configurations.

This chapter will describe a multiple FPGA implementation based on two RC1000 boards connected in parallel. Each of these boards has its own 8 Mbytes of memory, as shown in section 5.4. As the RC1000 boards each have one FPGA each with its own memory connected via the PCI bus, this system can be considered as a distributed memory system.

An alternative multi-FPGA system, a shared memory system, is also considered and some predictions of its performance and behaviour are made.

This chapter will also discuss how well the high and low level parallelism hardware implementation could benefit from the features of advanced FPGAs:

- Run-time reconfiguration of the FPGA
- FPGAs with embedded microprocessors

8.2 Multi-FPGA Distributed Memory System

This section will describe a multi-FPGA system based on two RC1000 boards connected in parallel via the PCI bus. The domain decomposition method used facilitates the spreading of the simulation across multiple FPGA boards with minimal communications overhead,

which means that near linear speed-up can be achieved. It will be shown that the implementation allows the full overlap of computation and communication between boards.

8.2.1 System Description

Figure 8–1 shows the diagram of a distributed memory system containing N RC1000 boards. Each board contains an FPGA, whose block RAM is organised as 6 dual-port RAMs, each of which is to be used to contain the data for a sub-domain. This data can be swapped in and out of four banks of static RAM present on each RC1000 board. The boards communicate with one another across the PCI bus. As long as the amount of data being transferred across the PCI bus between the boards remains small, linear speed-up can be expected as more FPGA boards are added, provided that the load balancing is good.

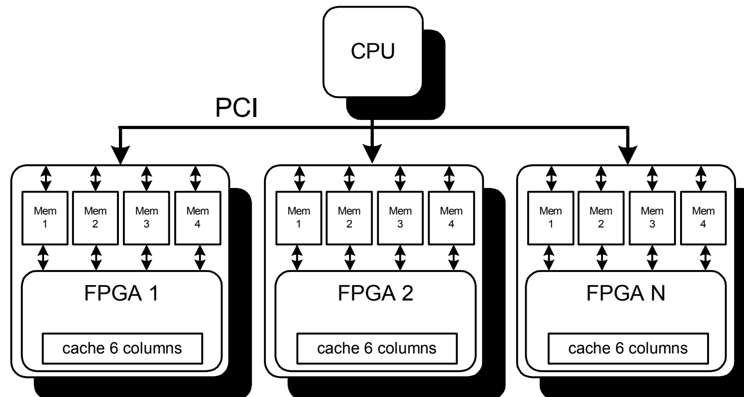


Figure 8–1 *Distributed Memory Multi-FPGA system*

The software frontend generates the initial configuration of the particles. The boards that are to be used must then be selected from a menu, as shown in Figure 8–3. The prototype

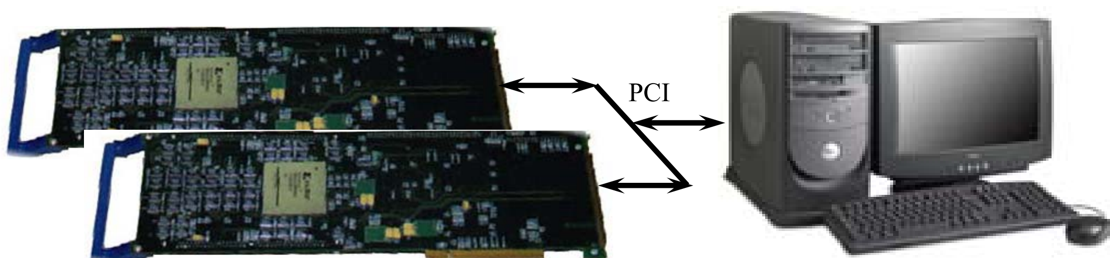


Figure 8–2 *Two RC1000-PP system*

implementation was set up to accept only two boards, since only two boards were currently available in the laboratory.

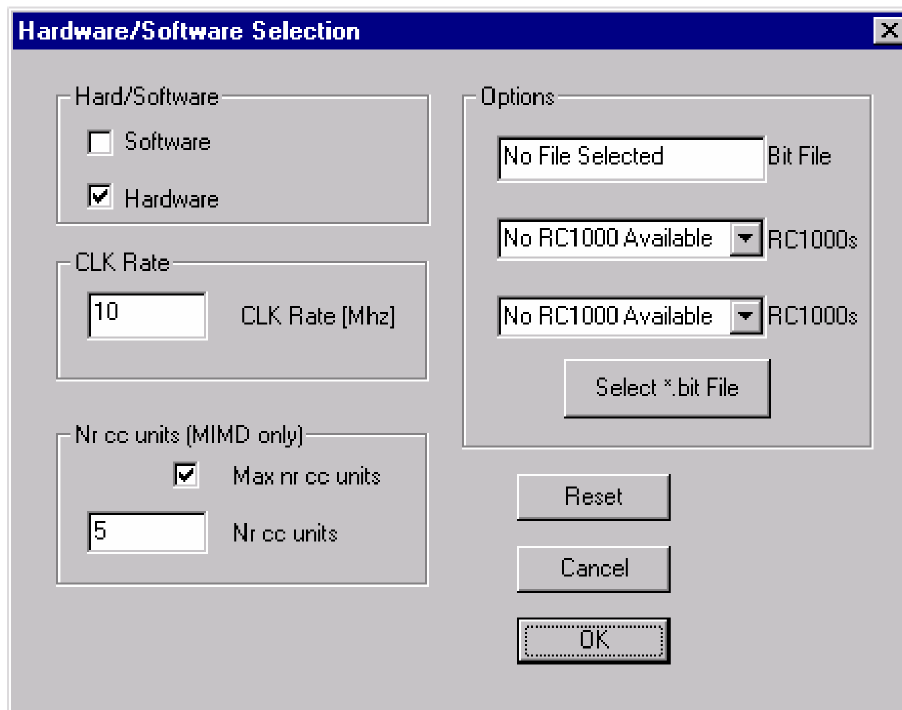


Figure 8–3 *Board selection for the Multiple FPGA design*

Initially the domain is split across the boards so as to equalise the workloads between the two boards (see Figure 8–4). After one time step is completed, each board needs to exchange its right most column including the data structures that catch particles transitioning across sub-domain boundaries, (see Figure 8–5) with its right hand neighbour. Similarly each board must exchange its leftmost column with its left hand neighbour. If this transfer can be completely overlapped with computation, then none of the computational pipelines on the FPGAs ever need to stall, and if the load balancing is good speed-up should be linear i.e. use of N boards should provide N times speed-up in comparison to a single board.

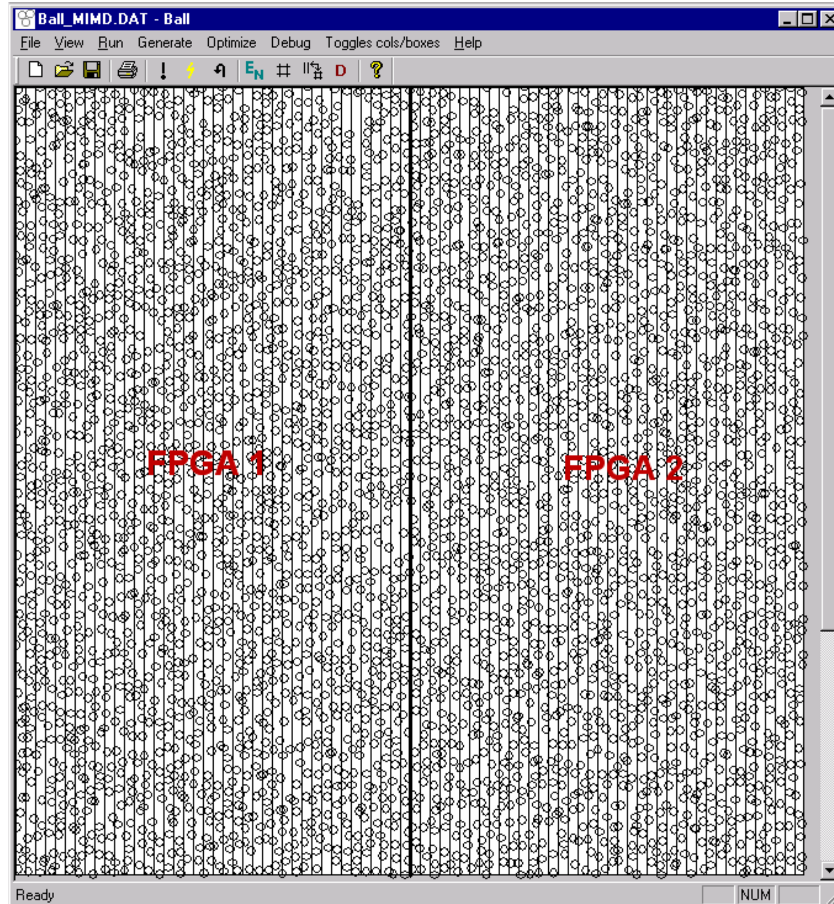


Figure 8–4 *Screenshot of the software program for the multiple FPGA design. The domain is split in two equally loaded parts*

The data for each particle in 2 dimensions consists of 34 bytes: 2 bytes each for x, y, v_x, v_y, $\dot{\theta}$, plus the normal and shear force for up to six contacts. The number of particles in a column is limited to 128 in order to avoid overflow of the block RAM. The load balancing method introduced in section 5.9.2 will adaptively reduce the size of any domain whose particle population approaches this limit. So for N boards, the maximum amount of data to be transferred across the PCI bus for each time step of the DEM method is 34 × 128 × 4 × N bytes, =17,41 N Kbytes. The 4 in the previous calculation comes in because two columns (left and right most column) have to be read from each FPGA and two have to be written in the FPGA Eq. 8–1 gives the general expression for the amount of data that needs to be transferred between boards for each time step.

$$Data_{transfered/board} = n^{o}{}_{of\ \ parameters} \times 2bytes \times \max \ \ balls \ \ col \times 4 \qquad \textbf{Eq. 8–1}$$

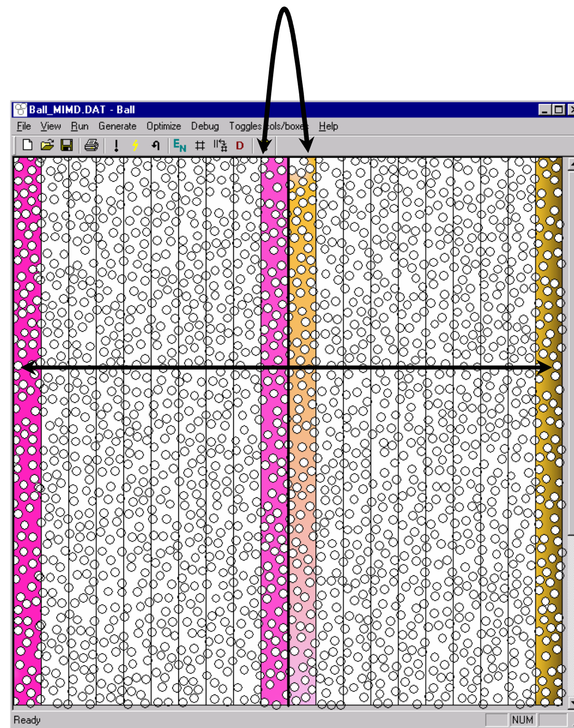


Figure 8–5 *Columns that need to be transferred from one board to another after every cycle*

Within each board, the FPGA uses one RAM bank at a time. Transfer of edge data to an adjacent board can be initiated when a bank of memory is released by the FPGA. The transfer must be completed before the FPGA attempts to re-acquire that bank, which occurs after it has finished processing the contents of the other three RAM banks on the board (see Figure 8–6). This amounts to a period of time as shown in Eq. 8–4. The slowest

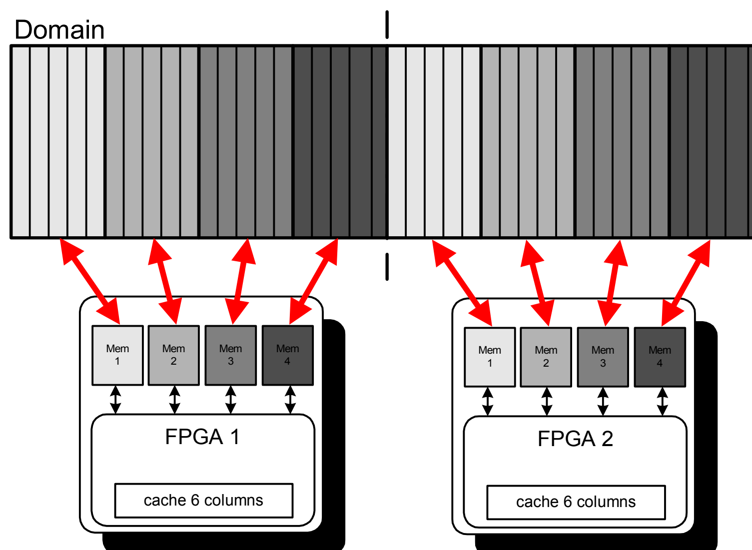


Figure 8–6 *Domain mapping to the 4 memory units of RC1000-PP board.*

operation in the processing of a sub-domain containing M particles is the position update and data transfer. The time $t_{pos_dat_transfer}$, taken for this operation for 1 cycle is shown in Eq. 8–3, where $t_{1column}$ is the time needed to compute 1 sub-domain, in this case equal to $t_{pos_dat_transfer}$ as this is the slowest task, and f is the clock frequency at which the FPGA works.

$$n^{o}{}_{cols/mem\ \ unit} = \frac{Mem\ \ unit\ \ size[Mbytes]}{Max\ \ nr\ \ balls_{1col} \times Parameters\ \ to\ \ describe\ \ Ball \times 2Bytes} \qquad \textbf{Eq. 8–2}$$

$$t_{pos_data_transfer(1cycle)} = \left(M + \frac{17 \times M}{2}\right) \times \frac{1}{f[Hz]} \qquad \textbf{Eq. 8–3}$$

$$t_{data\ \ transfer} = Memory\ \ units - 1 \times t_{pos_data_transfer(1cycle)} \times n^{o}{}_{cols/mem\ \ unit} \qquad \textbf{Eq. 8–4}$$

$$N = \frac{t_{data\ \ transfer}}{Data_{transferred/board} \big/ t_{PCI}} \qquad \textbf{Eq. 8–5}$$

The number of sub-domains contained in each RAM is shown in Eq. 8–2 for the 2MBytes memory available on the current FPGAs.

The boards are capable of sustaining DMA transfers across the PCI bus at about 12 Mbytes/s, which means that saturation of the bus will not occur for a number of boards N below about 153. This means that if ideal load balancing is achieved then speed-up can be expected to be linear for number of boards to be less than 153.

Table 8–1 shows an approximate number of boards that could be connected in parallel with this configuration, if it is considered that all 4 memory units are full with data.

Table 8–1 *Example of the number of boards that can work in parallel without having to stall any operation in any of the FPGAs.*

VARIABLE	VALUE	DESCRIPTION
M	128	Maximum number of balls in a column. (Limited by the XCV2000E internal memory)
$N^{o}_{parameters}$	17	Number of parameters that describes each ball
$Data_{transferred/board}$	17.41Kbytes	Number of bytes that each board needs to send and receive after each cycle (Eq. 8–1)
$t_{data\ transfer}$	0.223[s]	Time needed for the CPU to transfer the data to the boards while the FPGAs access the 4th memory unit (Eq. 8–4).
$t_{pos_data_transfer}$	0.162[ms]	Time needed for the slowest unit (position update + data transfer) to finish processing one sub-domain (Eq. 8–3)
$N^{o}_{subdomains/mem\ units}$	459	Number of sub-domains that each Memory unit of the RC1000 board can host (Eq. 8–2)
t_{PCI}	12 Mbytes/s	Transfer rate across the PCI bus (experimental , though theoretical should be 133 Mbytes/s)
N	153	Theoretical maximum for the number of boards that could work in parallel without having to stop the FPGA (Eq. 8–5)

A very important consideration for this configuration is the granularity of the board memory. It is more convenient and efficient to have the memory units split into many small banks, which can be accessed either by the FPGA or the CPU, than to have only a few large FPGA memory banks. This would allow an even higher number of FPGAs to be connected in parallel, as the CPU would have more time to read and write data to the FPGAs, before the FPGA requests the information in that particular memory bank. Figure 8–7 shows an example of this. If, for example, the 8 Mbytes of the board memory were distributed across 8 banks of 1 Mbyte (instead of the 4 banks of 2 Mbyte) approximately

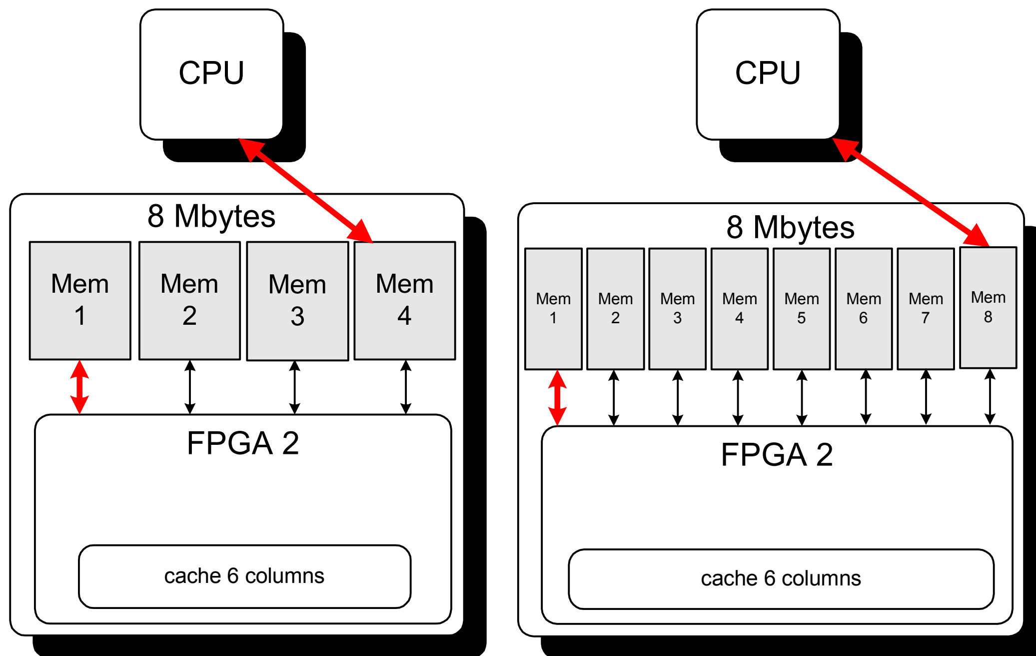


Figure 8–7 *Influence of a finer and coarse grained board memory*

17 % more boards (179 rather than 153) can be fitted in parallel in the system (assuming that all memory banks are filled with data). Table 8–2 shows the new timing considerations for this case.

Table 8–2 *Example of the number of boards that can work in parallel without having to stall the operation of any FPGA for 8 memory units instead of 4*

VARIABLE	VALUE	DESCRIPTION
M	128	Maximum number of balls in a column
$N^o_{parameters}$	17	Number of parameters that describes each ball
$Data_{transferred/board}$	17.41 Kbytes	Number of bytes that each board needs to send/receive after each cycle (Eq. 8–1)
$t_{data\ transfer}$	0.26 s	Time that the CPU has to transfer the data to the Boards while the FPGAs access the 4th memory unit (Eq. 8–4).
$t_{pos_data_transfer}$	0.162 ms	Time Needed for the longest unit (position update + data transfer) to finish for 1 sub-domain (Eq. 8–3)
$N^o_{subdomains/mem\ units}$	229	Number of sub-domains that each memory unit of

		the RC1000 board can host (Eq. 8–2)
t_{PCI}	12 Mbyte/s	Transfer rate across the PCI bus
N	179	Theoretical maximum number of boards that could work in parallel without having to stop the operation of any FPGA.

The next sub-section will show the results of the 2 RC1000 board system just described.

8.2.2 Simulations Results

In section 6.2.2, a domain was simulated for varying numbers of particles. The average speed-up was around 30 in comparison to the optimised software version running on a Pentium© III processor with 1.3 Gbytes of RAM. It is therefore reasonable to expect a speed-up of around 60 with a two-board system, as communication overheads should not influence the computing time, as computation and communication are completely overlapped.

Table 8–3 *Comparisons of speed-up obtained by hardware DEM for a single FPGA and two FPGAs compared to an optimised software version.*

NO. OF PARTICLES	50,000	75,000	100,000	125,000	150,000
Speed-up $_{measured}$ (1 board)	35.3	31.0	29.8	30.2	29.5
Speed-up $_{measured}$ (2 boards)	54.0	55.2	54.7	53.7	54.9

The hardware simulation for a system with two boards gave a result slightly worst than the expected linear speed-up of 60. This is due to the synchronization of the FPGAs needed after every cycle. The fastest FPGA needs to wait for the slower ones to complete their computation; the system is not completely balanced.

8.3 Shared Memory System

This section will analyse if the FPGA hardware architecture would be suitable for a shared memory system. A theoretical scalable implementation based on a shared memory approach will be described in detail, as well as some speed-up predictions based on the single and on the distributed memory system results.

8.3.1 System Description

Figure 8–8 shows an example of a shared memory structure. Here any of the FPGAs can access any of the memory modules through the interconnection network. Computation results are stored in the memory by the FPGA that executed the task Two major problems can arise in this type of architecture:

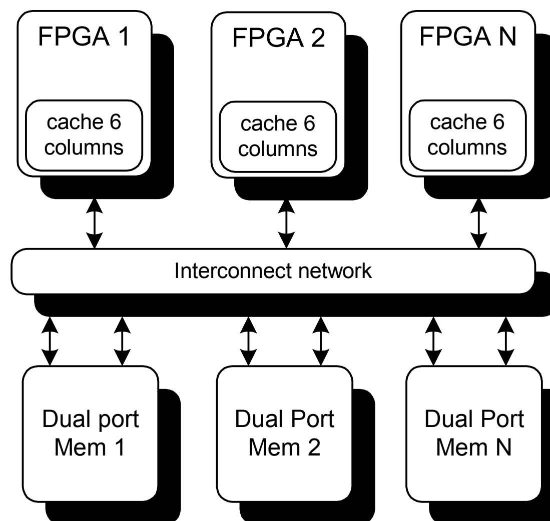


Figure 8–8 *Example of a shared memory system*

1. While the data is in one FPGA cache waiting to be updated another FPGA can access that data and generate a different result for it.
2. Two FPGAs try to write to the same memory location at the same time, i.e. memory contention.

A major benefit of the domain decomposition described in the section 5.9 is that it involves a geometrical division of the domain, so that all the data corresponding to one sub-domain will be stored in one memory unit. Given that each sub-domain is uniquely associated with

one FPGA, this implies that no two FPGAs will access the same memory location at the same time.

It can sometimes occur that two FPGAs attempt to access the same memory unit, but not the same memory location. This occurs when the FPGA is dealing with the leftmost and rightmost columns of one sub-domain. In this case it also needs to access the columns to the right and left of these columns. Using a dual port RAM can enable two simultaneous accesses to a memory unit. With the configuration shown in Figure 8–9 an FPGA will never try to access a memory location currently accessed by another. In this case FPGA 3 is lagging slightly compared to FPGA 2. In this case, both might access the same memory unit at the same time, but as seen in this figure the memory location will always be different as the geometrical distribution of the problem is also reflected in the data storage. Thus using dual port RAM would solve the possible memory contention problem.

The only problem that may occur is that the FPGAs need to be synchronized after every time step. This means that, as with the distributed memory system, the time needed to compute one cycle would be dictated by the slowest FPGA, as the system will never be totally balanced.

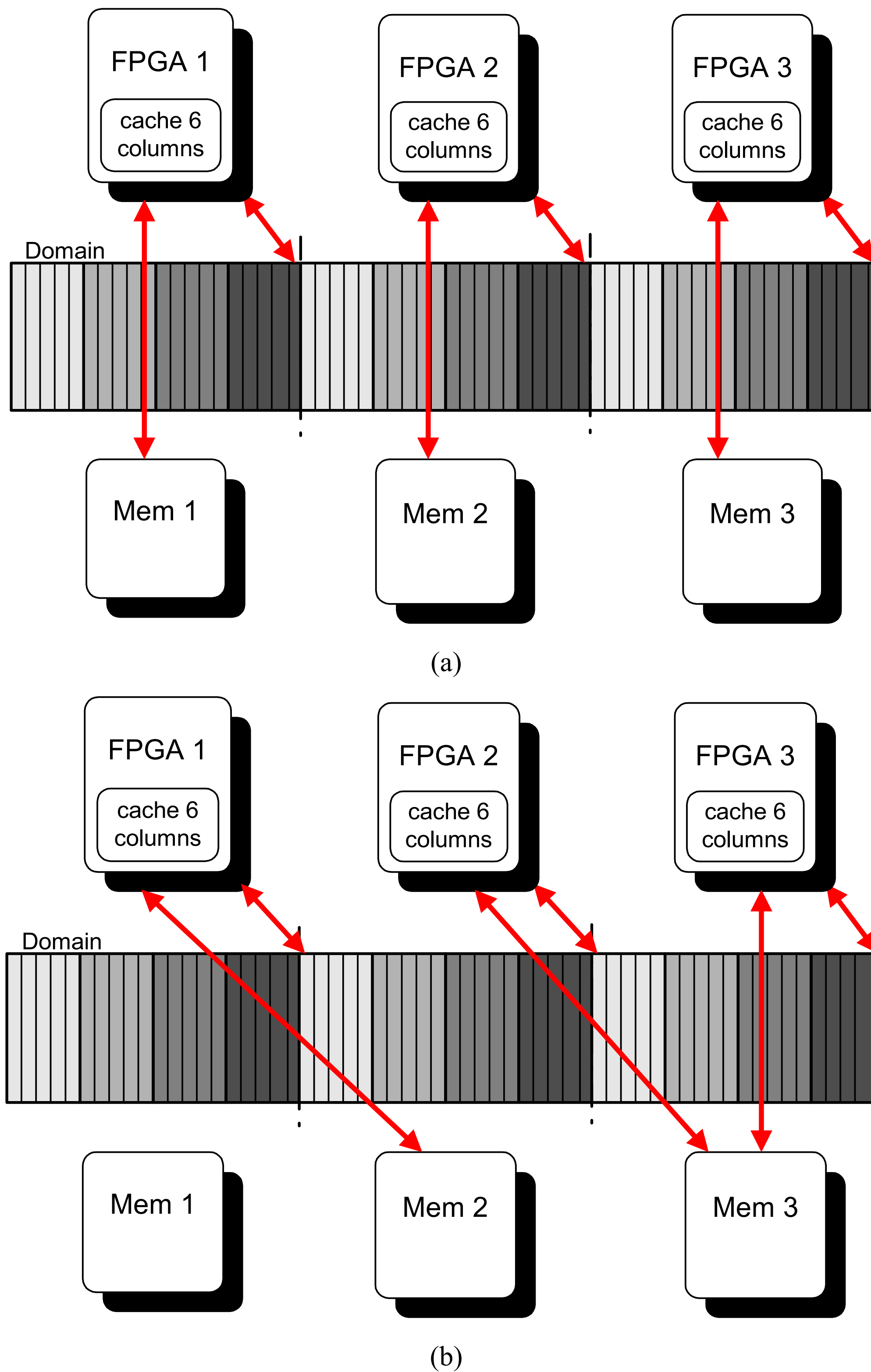


Figure 8–9 *FPGAs' memory accesses*

8.3.2 Speed-up Predictions

The configuration given in the last sub-section should give the same ideal speed-up results as the distributed memory system, i.e. a factor of 30 for each FPGA. As before, in practice the speed up is limited by the *quality* of the load balancing, but not by the communication overheads between FPGAs. The only moment when one of the FPGAs have to stall is when it reaches the end of the sub-domain and there is at least one other FPGA that has not finished processing its sub-domain.

The advantage of this type of configuration over the distributed memory system is that:

- The system can be built with an unlimited number of FPGAs without any penalty, assuming an ideal bus, which doesn't saturate.
- The design is independent of the amount of data stored in the external memory as it can be guaranteed that no communication overhead will occur as in this case an FPGA will NEVER access the same memory location as the neighbouring ones.

That having been said, in practice, interconnection networks do not scale well, and this solution may not be practical for a large number of FPGAs.

8.4 Alternative Single FPGA implementation

This section will provide a review of alternative single FPGA architectures. There are many different types of FPGAs, and many different design configuration possibilities enabled by newer FPGAs. This section will analyse how well the hardware design can be adapted to take advantage of newer FPGA features, such as run-time reconfiguration and embedded microprocessors.

8.4.1 Runtime Reconfigurable Architecture

Run-time reconfiguration (RTR) has been around since the Xilinx™ 6200 series [2] Much research has been done on runtime reconfiguration since the Xilinx™ 6200 series was released [3][4]. The device allows partial reconfiguration at run-time in a fine-grained manner. However, the XC6200 is now obsolete. Instead, the Xilinx™ Virtex series has

become available. The Virtex architecture allows partial dynamic reconfiguration in a more coarse grained fashion, although this feature is currently not supported by the Xilinx design tools. However, it has been very difficult to find numerical intensive applications, which could demonstrate an improvement over conventional systems by making use of this feature.

Normally the time needed to re-program the FPGA makes RTR prohibitive for most applications. As FPGAs have become more sophisticated, the rate at which they can be reconfigured has become higher, and the re-programming some parts of the FPGA whilst other parts are still performing useful computation has become possible. A theoretical design of a HW design using run-time reconfiguration will be presented in this section.

A block diagram of a run-time reconfigurable architecture is shown in Figure 8–10. This implementation takes the area used in the static design for the contact check, the force update and the position update units, and replaces them with a large pool of logic resources that will be reconfigured on demand to perform contact checking, force update and position update. The pool of resources left for run-time reconfiguration accounts for about 35% of the logic resources of the XCV2000E FPGA. The advantage of this architecture is each phase of the DEM algorithm can be allocated as much resource as it is capable of benefiting from. Table 8–4 shows how many units can be instantiated into this area for each of the computational phases of the DEM

Table 8–4 *Number of units that can be implemented in the reconfigurable area*

	CONTACT CHECK UNITS	FORCES UPDATE	POSITION UPDATE
% slices XCV2000E for 1 unit	1 %	15 %	11 %
Nr of units	35	2	3

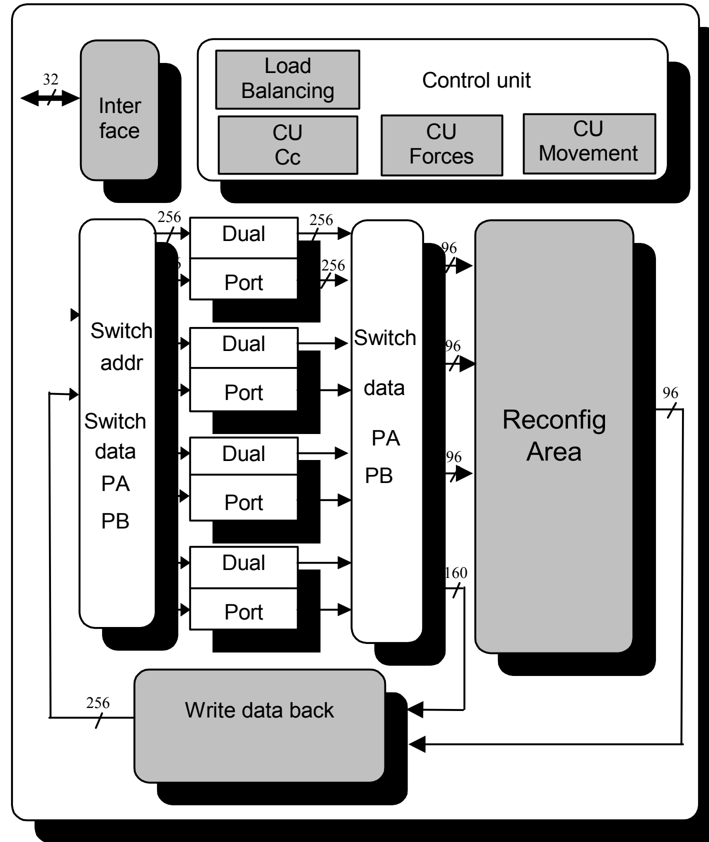


Figure 8–10 *Runtime reconfigurable architecture*

The disadvantage of this approach is that the reconfiguration will take a lot of precious time, which might be prohibitive. Eq. 8–6 shows the time needed to compute one column of the domain and Figure 8–11 shows a graphical representation of the sequence. The reconfiguration time will be the same in each case as the total amount of reconfigurable area is used instantiating multiple units in parallel.

$$t(1\quad col) = t(cc) + t(forces) + t(position) + 3 \times t(reconfiguration\quad time) \qquad \textbf{Eq. 8–6}$$

With this approach, the time taken by each of the major phases of the DEM, t(cc), t(forces) and t(position), will be decreased significantly. However, a reconfiguration penalty will be incurred three times for each sub-domain for each time step.

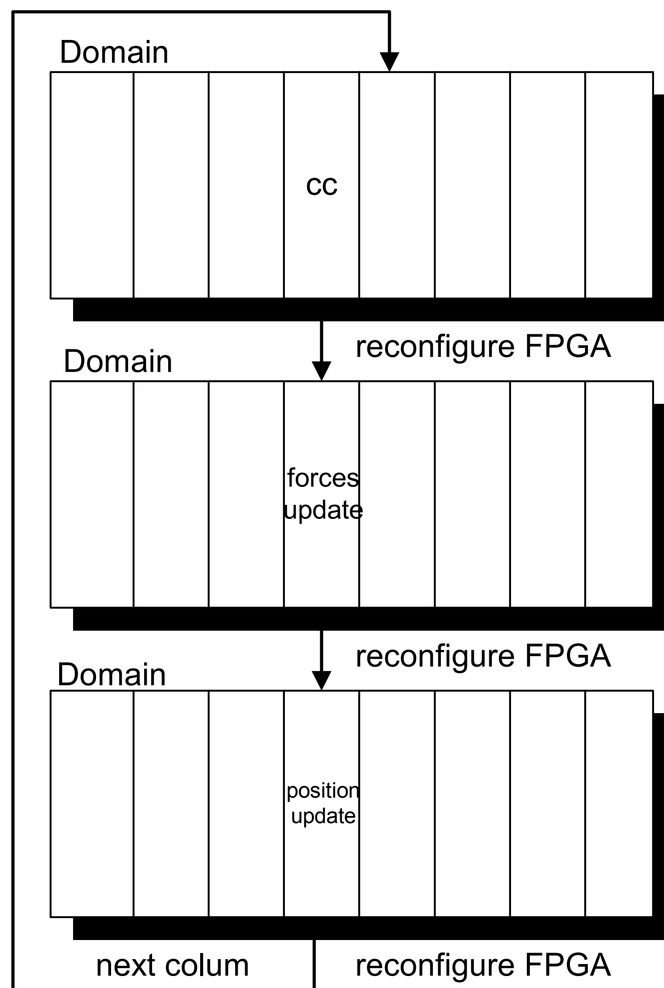


Figure 8–11 *Reconfigurable sequence*

It should be noted that in this case the internal FPGA memory is split into just 4 equally sized blocks instead of 6, as all three tasks are performed on the same column. So the FPGA memory stores the column presently undergoing processing, its two neighbouring columns (to handle edge effects), plus one more so that when processing of the current column is complete, write back of the column data to the external memory can be overlapped with processing of the next column. This is shown more clearly in Figure 8–12, where it can be seen which columns are cached in the FPGA memory at a particular time of the simulation.

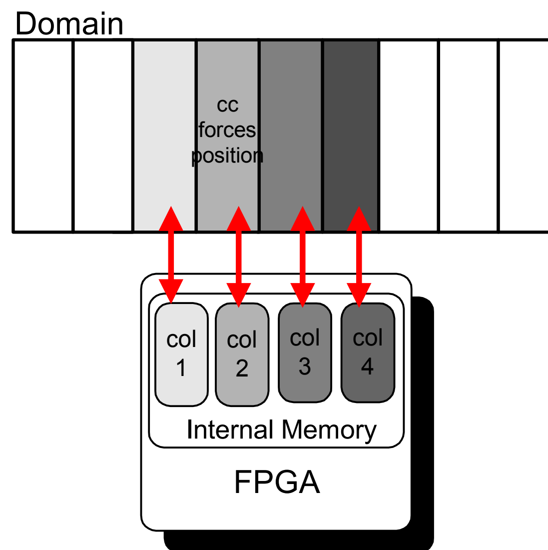


Figure 8–12 *Number of columns to be cached into the FPGA*

8.4.2 Case Study of a Runtime Reconfigurable Design

In this case study a run-time reconfigurable system will be analysed based on the Xilinx Virtex ™ XCV2000E device. The configuration bit stream required to configure the entire FPGA is 1.2699 Mbytes in size. As only around a third of the FPGA needs to be reconfigured after each of the major tasks has been finished, only $1.2699/3 = 0.4233 Mbytes$ will need to be reconfigured. The FPGA has 4 modes available for the reading of configuration data [2]:

1. Master-Serial mode
2. Slave Serial mode
3. Boundary Scan mode
4. SelectMAP mode

In both serial modes, and also in boundary scan mode, the FPGA receives the configuration data in bit-serial form. By contrast, the SelectMAP mode received the data as a 1-byte wide data stream, and is therefore the fastest configuration option. The maximum frequency at which the SelectMAP mode can operate for this device is 66 MHz.

In order to analyze whether this design could be faster and more efficient than the static architecture presented in chapter 5, an analytical study has been performed. Table 8–5 shows the time taken for each step of the DEM algorithm, and the time required to reconfigure the FPGA using the SelectMAP mode.

Table 8–5 *Values of the time needed to compute the cc, forces and position update versus the time needed to reconfigure the 50% of the Xilinx XCV2000E.*

Value	Description
0.4233 [Mbytes]	Amount of data needed to reconfigure the 35% of the XCV2000E device
66 [MHz]	Frequency at which the reconfiguration can take place
1 byte	Data width to write to the FPGA
6.4 [ms]	Time needed to reconfigure the 35% of the FPGA
9.5 [μs]	Time needed to compute the contact check for 100 balls/col and 35 contact check units @30MHz
40[μs]	Time needed to compute the forces for 100 balls/col and 2 forces update units @30Mhz/4
4.4[μs]	Time needed to compute the positions for 100 balls/col and 3 positions update unit @30Mhz/4
6.5 [ms]	Time needed to compute 1 column
98.5 %	Percentage of computation time spent on reconfiguration
0.156 [ms]	Time needed to compute 1 column with the *normal* architecture (5 cc units, 1 forces and 1 position update unit)

As can be seen from this study, 99.1 % of the time taken to compute the data in one column is spent reconfiguring the FPGA, while the major computation tasks take only 0.9 % of the time. This means that the run-time reconfigurable design would have a very low efficiency, as the FPGA spends most of the time in a reconfiguration mode. Comparing this result with the time needed to compute one column in the design, where no reconfiguration is required, but with fewer units instantiated in parallel, this architecture would be $6.5/0.156 = 41$ times slower than the static design.

The conclusions that can be drawn from this configuration is that the time needed to reconfigure the FPGA is too big, making the efficiency of this design very low.

In order to have a system that would at least perform as well as the design, the FPGA would need to be able to reconfigured at 12.5 GHz one byte at time, or with 188 bytes at a time at 66 MHz. This can be deduced from Eq. 8–7, where the FPGA needs to be

$$3 \times \frac{0.4233[Mbytes]}{n^o \ \ of \ \ bytes \ \ in \ \ parallel} \times \frac{1}{f_{reconfiguration}} = 0.156[ms] - t_{arithmetic \ \ units \ \ reconfig \ \ case} \quad \textbf{Eq. 8–7}$$

reconfigured three times in every cycle. Future FPGAs are unlikely to be able to offer such high reconfiguration rates.

8.4.3 Implementation on an FPGA with embedded Microprocessors

New FPGAs, such as the Xilinx™ Virtex©-Pro FPGA, incorporate up to 4 embedded Power PC RISC microprocessors. This section will analyse whether a DEM hardware implementation on FPGAs with embedded microprocessors could bring even better results. Figure 8–13 shows the internal logic structure of an FPGA with embedded microprocessors. These microprocessors are embedded between the logic resources so that they can work closely with the custom logic. These new FPGAs also have embedded 18 bit multipliers (the XC2VP125 has up to 556 multipliers), which could be used to free up more logic resources.

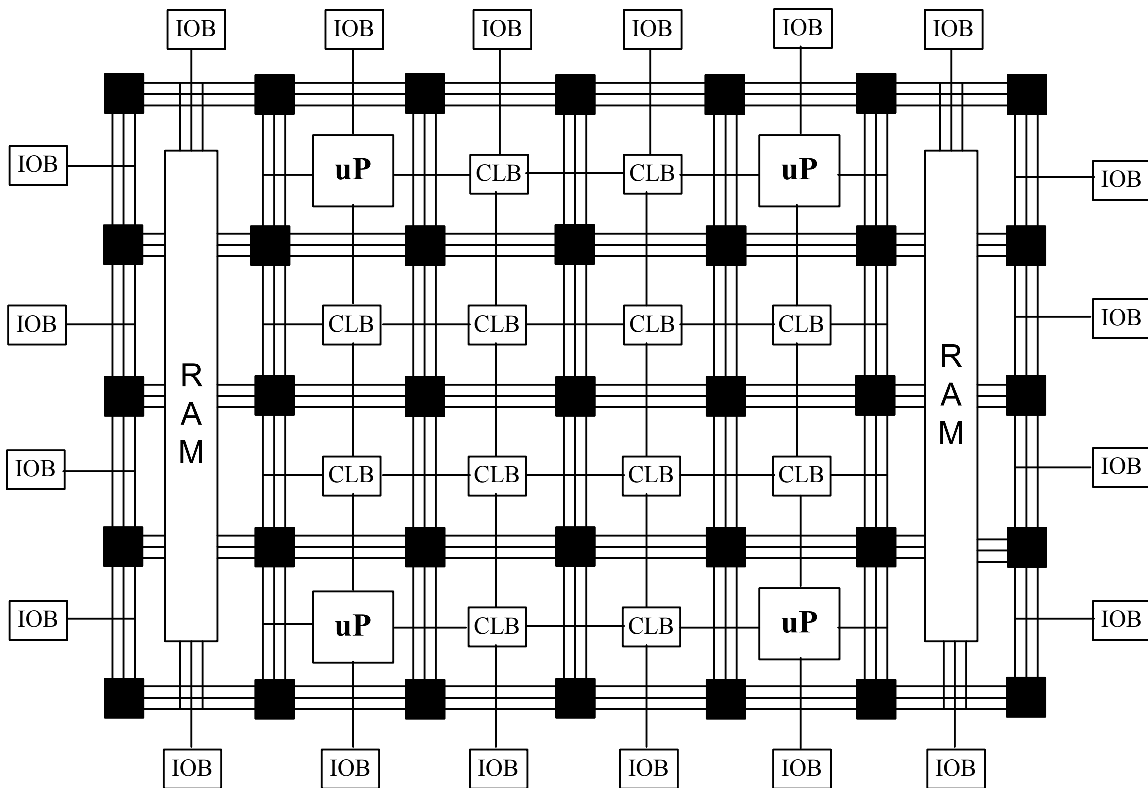


Figure 8–13 *Internal FPGA structure with microprocessors*

The next section will describe a theoretical HW implementation of the design on an FPGA with embedded microprocessors and will estimate its possible performance.

8.4.4 Case Study of an FPGA with Embedded Microprocessors

A design which could be implemented on a Xilinx™ Virtex2™ XC2VP40 with 2 embedded Power PC microprocessors is analysed in this case study. The processors are RISCs (Reduced Instruction Set Computers) type processors with a core running at 300 MHz.

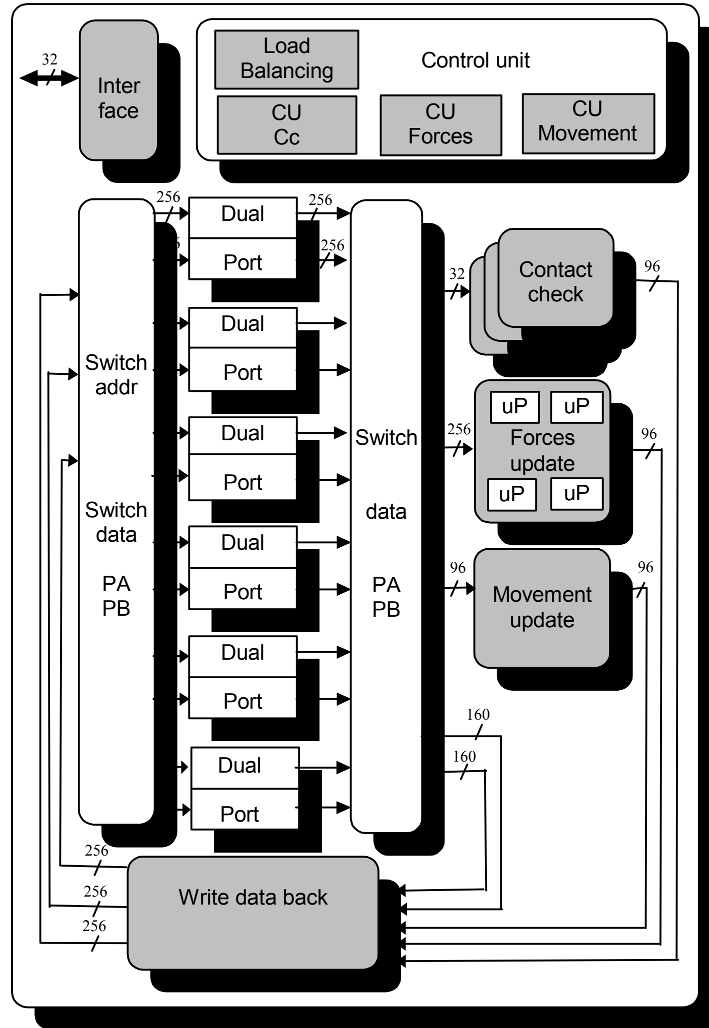


__Figure 8–14__ Hardware implementation using an FPGA with embedded microprocessors

For the design presented in chapter 5, the force update unit is the consumes the most hardware. This unit could therefore be allocated to the microprocessors, as shown in Figure 8–14. This should give the best efficiency for the whole system.

The number of arithmetic operations needed by the forces update unit is shown in Table 8–6.

Table 8–6 *Number of arithmetic operations involved in the forces update unit*

	Additions/Sub	Multiplications	Divisions
Number or arithmetic operations	16	18	1

Xilinx's data sheets give the cycle time required to compute each of the arithmetic operations as shown in Table 8–7.

Table 8–7 *Number of operations needed for each arithmetic operation.*

	Additions/Sub	Multiplications	Divisions
Cycles needed by the Power PC (32 bits ops)	1	4	35

This means that the time to compute the forces between two particles in contact would now be as shown in Eq. 8–8. The unit requires 16 additions that can be done in a single cycle, 18 multiplications that will take 4 cycles each, and 1 division that will take 35 cycles:

$$t_{forces} = [16 + (18 \times 4) + 35] \times \frac{1}{300MHz} = 0.41\mu s \Rightarrow throughput = 2.4MHz \qquad \textbf{Eq. 8–8}$$

New data has to be fed to the microprocessor after every 0.41 μs, which is equivalent to a frequency of 2.4 MHz. This would mean that data should be read from the FPGA's internal memory at 5 MHz in order to keep all the 2 microprocessors fully working.

The resources freed by the microprocessor can now be allocated to the position update unit as well as to the contact check units. Considering that the forces update unit took 15 % of the XCV2000E device, the position update unit 1%, and the contact check units 1%, the space saved by migrating the force update into the microprocessors could be used to instantiate additional position update unit and additional contact check units. (The XCV2000E and the XC2VP40 have almost the same amount of logic resources available).

Table 8–8 *List with the number of units implemented in parallel.*

	Contact check	Forces	Position
Nr of units in parallel	9	2 microprocessors	2

The position update task, combined with the reading and writing of data to the external memory, was the bottleneck of the original design presented in chapter 5. For the new design using embedded microprocessors, this would still be the case. Although the position update has been accelerated by a factor of 6, the reading and writing of data to the external memory remains the same, as shown in Figure 8–15.

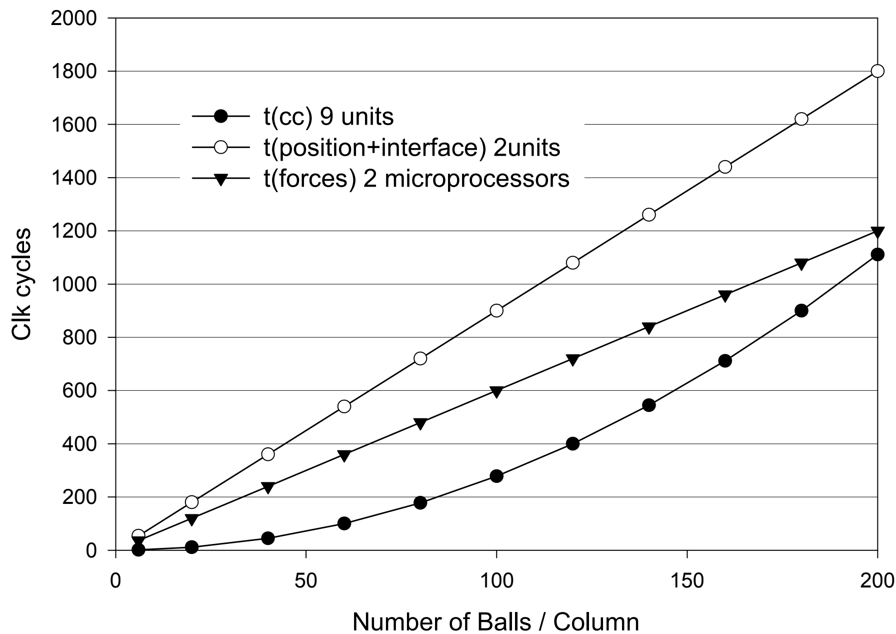


Figure 8–15 *Computation time of the three main task, replacing the forces update unit with 2 microprocessors*

As reading and writing data to the external memory is the bottleneck of the design, the performance would be improved by using a board with memory of higher speed.

8.4.5 Discussion for the Proposed Single FPGA Architectures based on the High and Low Level Parallelism HW design

Two alternative architectures were proposed in section 8.4 that make use of two of the key features of modern FPGAs:

- Run-time reconfiguration
- Embedded microprocessors

The implementation of a hardware implementations using run-time reconfiguration based on the high and low level parallelism architecture was presented in section 8.4.1. The advantages of this approach is that the total amount of logic resources used by the major tasks (contact checking, forces and position update) can be combined to perform each at a time, instantiating more of these units in parallel and therefore speeding their processing up. The disadvantages are that these operations will no longer be performed in parallel, and that the computing will have to stall until the FPGA is re-configured.

A case study using the Xilinx™ XCV2000E was presented, which showed that 97.4 % of the time in each time step was spent reconfiguring the FPGA, thus achieving a very low design efficiency. In order to achieve the same computing speed as the original architecture presented in section 5.9, the reconfiguration should take place at 8 Gbyte/s instead of the 66 Mbyte/s supported by present day Xilinx FPGAs.

The second proposed architecture involved FPGAs with embedded microprocessors. By using an FPGA with embedded microprocessors, logic resources of the device can be freed and some hardware-consuming task can be allocated to the processors.

A case study using a Xilinx™ Virtex2™ XC2VP40 with two embedded RISC microprocessors showed that allocating them to the forces update task would free some resources which could be used to speed the position update and contact checking units. The RISC processors will need to be fed at a frequency of 2.5 MHz,

As the bottleneck of the design is not the computational units, but the data transfer rate between the internal FPGA memory and the external memory, no improvements in term of speed-up can be projected in this case. Having an improved reconfigurable computing platform with higher memory bandwidth could alleviate this bottleneck.

8.5 Summary and Conclusions

The design of a distributed memory multi-board system has been analysed and implemented. As more boards are added, almost linear speed-up is achieved. Only the synchronization of the FPGAs after every time step keeps the speed-up slightly worse than linear. The overall performance depends on how well the system load is balanced: the most heavily loaded FPGA limits the speed-up of the entire system. Up to 153 boards could be connected in parallel before an FPGA had to stall because communication and computation could not be overlapped. This depends of course on the amount of data in each board, as this estimate assumes that four memory banks of the RC1000 board are filled with data and that the CPU will have the time that the FPGA needs to compute the data of 3 memory units available to perform the data transfer.

An alternative multiple FPGA system has also been proposed, based on a shared memory approach. Section 8.3 provides estimates of its theoretical performance. Using dual-port external memory units and a suitable interconnect network, a linear speed-up can be achieved independently of the amount of particles in the system, thus having a theoretically perfect scalable system. However, this system suffers from the same problem as the distributed memory. The fastest FPGA will need to wait for the slowest one to complete its processing before a new time step can begin. Thus the time to compute one cycle will depend on the quality of the load balancing, which must try to ensure that all the FPGAs have the same workload.

The last part of this chapter described alternative single FPGA systems, based on the hardware design. A runtime reconfigurable design was presented; estimates based on the capabilities of present day FPGAs suggest that this approach is not promising, as the reconfiguration overhead is unacceptably large.

The use of embedded microprocessors within the FPGA was also considered. According to theoretical calculations, this showed some improvements over the pure hardware architecture, as the position update task could be computed faster; however the I/O bottleneck with the external memory stills exists.

Overall it has been shown that the HW architecture scales almost linearly with multiple FPGAs working in parallel and that it can serve as the basis for other single FPGA architecture that exploit the features of new FPGAs to provide very promising results.

8.6 References

[1] Smith, B., Bjorstad, P., Gropp, W. *"Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations"*, Cambridge University Press, 1996.

[2] www.xilinx.com

[3] Lopez-Buedo S, Riviere, P, Pernas, P, Boemo, E, *"Run-Time Reconfiguration to Check Temperatures in Custom Computers: An Application of JBits Technology",* Field Programmable Logic and Applications (FPL'2002) Montpellier, 2002.

[4] Smit, G.J.M, Havinga, P.J.M, Smit, L.T., Heysters, P.M, Rosien, M.A.J. *"Dynamic Reconfiguration in Mobile Systems"*, Field Programmable Logic and Applications (FPL'2002) Montpellier, 2002

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

The study of granular materials is important to many engineering disciplines. The Discrete Element Method is one of the best methods to model the behaviour of particle assemblies. Applying the DEM to assemblies of particles bonded together to form a solid body is also promising for the modelling of effects, such as crack propagation, that are not well modelled by continuum methods such as the finite element method. However, the usefulness of the DEM is limited by its extreme computational demands. These derive from the fact that it treats every particle individually, and the time step must be very small in order to maintain numerical stability.

A number of previous studies have used multiprocessor computer systems to accelerate the DEM. However, high speed-ups could only be achieved for "nice" problems, whose geometry was chosen so as to be very benign for parallel processing. For more awkward problems, speed-ups were far less than linear, due to communication and synchronization overheads and load balancing problems.

This thesis has presented a completely new approach to the acceleration of the DEM computation based on reconfigurable computing (this use of FPGAs to provide custom hardware accelerators that can be used as co-processors in standard computers). This novel approach exploits the intrinsic low and high-level parallelism of the DEM by scheduling the arithmetic operations in parallel, and by decomposing the domain so that the main stages of the DEM (contact checking, force update and position update) can be performed concurrently. A single FPGA implementation exhibited a speed-up of at least a factor of 30 over an optimised software version.

One of the key features of the design that enabled the exploitation of high level parallelism is the domain decomposition used, and the manner in which this was mapped onto the FPGA. The domain was decomposed into columns and the FPGA internal memory was divided into 6 equal blocks. This allowed 6 columns of the domain to be cached onto the FPGA at one time, and is the key factor that enables overlap of the computational units.

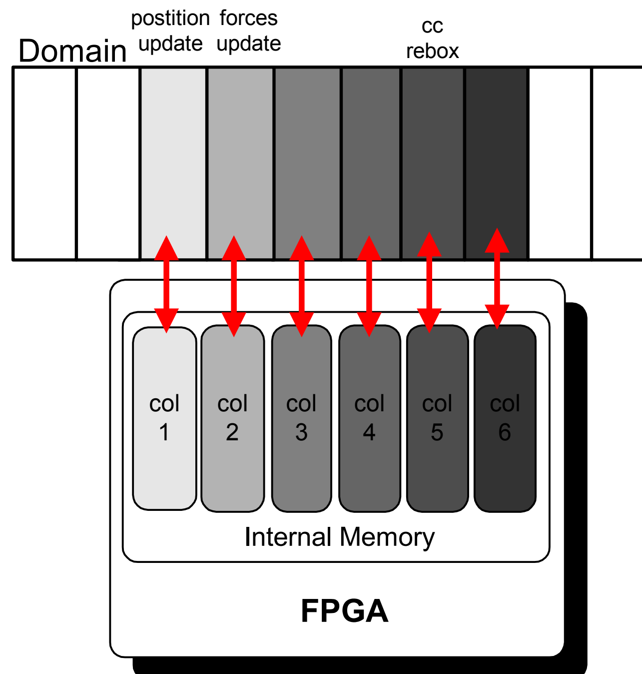


Figure 9–1 *Mapping of the domain decomposition on the FPGA's internal memory*

It is important to note that the re-boxing of particles transitioning from one column to another is completely free, as is the dynamic adaptation of the domain boundaries in order

to maintain good load balance. This is the key to why the FPGA version can achieve a more linear speed-up than can a conventional parallel processor computer.

The design included an interesting example of the ability of reconfigurable computing to trade off time against hardware. The contact check unit requires $O(N^2)$ operations (as opposed to the other computational units, which are $O(N)$). In a purely sequential implementation, it would therefore dominate the simulation time. However, the contact check unit requires extremely simple hardware compared to the other computational units. Therefore a large number of contact check units can be instantiated in parallel without consuming excessive hardware resource, thus giving an excellent speed-up. The design is organised so that the number of contact check units is a generic parameter within the VHDL description. So as the design is retargeted to a larger FPGA, recompilation of the design with just one parameter changed enables the additional FPGA hardware to be used to instantiate a much larger number of these units, providing still further speed-up.

A multiple FPGA solution was designed that can completely overlap computation and communication even for large numbers of FPGAs. A 2 FPGA design was implemented, and was demonstrated to have a speed-up of almost a factor of almost 60 over the software version. A truly linear speed-up cannot be achieved, since the load balance will never be perfect, and there is an overhead associated with re-synchronising the FPGAs at the end of each time step. However, the FPGA solution can, without time penalty, overlap communication between domains, and perform dynamic domain boundary adjustment in order to optimise load balance. It is therefore reasonable to assume that its speed-up should be closer to linear than can be achieved by parallel computers.

In order to fit the design onto an FPGA which is, by modern standards, rather small, a number of simplifications had to be made to the DEM algorithm. The simulator could only treat 2-D simulations of a domain containing no walls with all particles having the same radius. An evaluation was carried out of how the design could be adapted to remove these simplifications. The resulting hardware complexity was found to be acceptable, and the speed-up compared to software would be excellent. However, a design that uses a variable radius would be somewhat unsatisfactory due to the complexity of the data structures that

would have to be used, which greatly increases the complexity of the control path, and imposes a severe strain on the memory bandwidth.

A software simulator for the 2-D and 3-D DEM was also developed. This was used for three purposes.

Firstly, it was used to investigate the interaction of domain decomposition and problem geometry with the capabilities of a multi-processor computer platform. It was demonstrated that for realistic problems, communication overheads are substantial, and will impose an overall limit on the speed-up that can be achieved. By comparison, the FPGA implementation can totally overlap communication and computation, and escapes this problem.

Secondly, the software provided a user-friendly front-end for the hardware simulator, that could be used to set up simulations, and could provide visual feedback to the user on the progress of the hardware simulation.

Thirdly, the software version was used to provide a reference implementation, whose results could be regarded as "correct" for the purpose of evaluating the correctness and accuracy of the hardware simulations. (This software version was itself validated by comparison of its results with those produced by a standard DEM code and by analytical expressions). The software also implemented a debug mode, in which the software and hardware versions could be run in parallel, and their results compared.

In order to conserve hardware resources, the FPGA uses short wordlength arithmetic. A careful error analysis was carried out, in order to identify areas in the computational pipelines that could give rise to problems. At appropriate points, extended wordlength or exception-handling hardware was used in order to avoid catastrophic loss of precision. Error analysis, and experimental comparison with the software version, demonstrated that the results produced by the FPGA are acceptable in 16 bit arithmetic (although 24 or 32 bit arithmetic would undoubtedly be better, and should be preferred as very large FPGAs become cheaper).

9.2 Future Work

The design implementation in this work provides a basic architecture on which more complex DEM models can be developed. These might include:

- Having particles with different radii
- Including walls in the domain
- A 3-D DEM model
- Having a higher numerical precision

The feasibility study presented in this thesis certainly indicates that these extensions are feasible and desirable (though it must be admitted that variable particle radius is somewhat problematic). A full detailed implementation of these features would be a useful extension of the work.

The scalability of the multi-FPGA system could only be demonstrated on a 2-FPGA system. It would be useful to carry out an experimental evaluation of a system with a large number of FPGAs in order to find out how well the load can be balanced, and the synchronisation overheads amortized.

More modern FPGAs contain embedded multipliers and embedded processors. This gives a much greater freedom to implement the various computational stages in the most suitable form, but all combined on the same FPGA. This holds out some very interesting possibilities in respect of dynamic load balancing, as tasks might migrate between hardware and software according to simulation conditions. A full investigation of the possibilities would be an interesting and useful extension of the present work.

APPENDIX

Appendix A.1: Publication List

Journal papers

1. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, " *Acceleration of the Discrete Element Method on a reconfigurable computing platform", Computers & Structures (Under review)*

Refereed conference papers

1. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, " *Numeric Modelling of the Mechanical interaction between non-biological particles using reconfigurable computing* ", 9th Annual conference of the Association for Computational Mechanics in Engineering (ACME), 2001, Birmingham, pages 53-56.

2. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, " *Evaluation of an FPGA implementation of the Discrete Element Method (DEM)* ", 11th International Conference on Field Programmable Logic and Applications (FPL), Belfast, 2001, pages 306-314 (c) Springer-Verlag.

3. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, " *Description of a dedicated Hardware Architecture for the Discrete Element Method (DEM) implemented on a Field Programmable Gate Array (FPGA)* ",10th Annual conference of the

Association for Computational Mechanics in Engineering (ACME), 2002, Swansea, pages 51-54.

4. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, " *Analysis and Implementation of the Discrete Element Method using a dedicated highly parallel Architecture in Reconfigurable Computing* ", 2002 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa Valley, California. (c) IEEE Computer Society.

5. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, " *Implementation of the Discrete Element Method using reconfigurable computing (FPGAs)* ", 15th Engineering Mechanics Conference (EM2002), Columbia University, New York.

6. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, "*Scalable Implementation of the Discrete Element Method on a Reconfigurable Computing Platform*", 12th International Conference on Field Programmable Logic and Applications (FPL), Montpellier, 2002, (c) Springer-Verlag.

7. Carrión Schäfer, B., Quigley, S.F, Chan, A.H.C, "*Scalablity Analysis of an Implementation of the Discrete Element Method on a Field Programmable Gate Array (FPGA"* 11th Annual ACME Conference, University of Strathclyde, Glasgow, UK, 24th-25th April 2003, WHEEL M. A. (ed.), University of Strathclyde, Glasgow, 177-180