Scalability of Fixed-Radius Searching in Meshless Methods for Heterogeneous Architectures

by

LeRoi Vincent Pols

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Research) in the Faculty of Engineering at Stellenbosch University

Department of Civil Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Dr. D.Z. Turner

December 2014

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: .....

Abstract

Scalability of Fixed-Radius Searching in Meshless Methods for Heterogeneous Architectures

L.V. Pols

Department of Civil Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MEng (Civ)

December 2014

In this thesis we set out to design an algorithm for solving the all-pairs fixed-radius nearest neighbours search problem for a massively parallel heterogeneous system. The all-pairs search problem is stated as follows: Given a set of N points in d-dimensional space, find all pairs of points within a horizon distance δ of one another. This search is required by any nonlocal or meshless numerical modelling method to construct the neighbour list of each mesh point in the problem domain. Therefore, this work is applicable to a wide variety of fields, ranging from molecular dynamics to pattern recognition and geographical information systems. Here we focus on nonlocal solid mechanics methods.

The basic method of solving the all-pairs search is to calculate, for each mesh point, the distance to each other mesh point and compare with the horizon value to determine if the points are neighbours. This can be a very computationally intensive procedure, especially if the neighbourhood needs to be updated at every time step to account for changes in material configuration. The problem also becomes more complex if the analysis is done in parallel.

Furthermore, GPU computing has become very popular in the last decade. Most of the fastest supercomputers in the world today employ GPU processors as accelerators to CPU processors. It is also believed that the next-generation exascale supercomputers will be

heterogeneous. Therefore the focus is on how to develop a neighbour searching algorithm that will take advantage of next-generation hardware.

In this thesis we propose a CPU - multi GPU algorithm, which is an extension of the fixed-grid method, for the fixed-radius nearest neighbours search on massively parallel systems.

Uittreksel

Skaalbaarheid van Vaste-Radius Soektogte in Roosterlose Metodes vir Heterogene Argitektuur

("Scalability of Fixed-Radius Searching in Meshless Methods for Heterogeneous Architectures")

L.V. Pols

Departement Siviele Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MIng (Siv)

Desember 2014

In hierdie tesis het ons die ontwerp van 'n algoritme vir die oplossing van die alle-pare vaste-radius naaste bure soektog probleem vir groot skaal parallele heterogene stelsels aangepak. Die alle-pare soektog probleem is as volg gestel: Gegewe 'n stel van N punte in d-dimensionele ruimte, vind al die pare van punte wat binne 'n horison afstand δ van mekaar af is. Die soektog word deur enige nie-lokale of roosterlose numeriese metode benodig om die bure-lys van alle rooster-punte in die probleem te kry. Daarom is hierdie werk van toepassing op 'n wye verskeidenheid van velde, wat wissel van molekulêre dinamika tot patroon herkenning en geografiese inligtingstelsels. Hier is ons fokus op nie-lokale soliede meganika metodes.

Die basiese metode vir die oplossing van die alle-pare soektog is om vir elke rooster-punt, die afstand na elke ander rooster-punt te bereken en te vergelyk met die horison lente, om dus so te bepaal of die punte bure is. Dit kan 'n baie berekenings intensiewe proses wees, veral as die probleem by elke stap opgedateer moet word om die veranderinge in die materiaal konfigurasie daar te stel. Die probleem word ook baie meer kompleks as die analise in parallel gedoen word.

Verder het GVE's (Grafiese verwerkings eenhede) baie gewild geword in die afgelope dekade. Die meeste van die vinnigste superrekenaars in die wêreld vandag gebruik GVE's

as versnellers te same met SVE's (Sentrale verwerkings eenhede). Dit is ook van mening dat die volgende generasie exa-skaal superrekenaars GVE's sal implementeer. Daarom is die fokus op hoe om 'n bure-lys soektog algoritme te ontwikkel wat gebruik sal maak van die volgende generasie hardeware.

In hierdie tesis stel ons 'n SVE - veelvoudige GVE algoritme voor, wat 'n verlenging van die vaste-rooster metode is, vir die vaste-radius naaste bure soektog op groot skaal parallele stelsels.

Acknowledgements

I would like to thank my parents, Roean and Nici Pols, and my girlfriend Suzane for supporting me over these past two years. For always being there when I needed help and motivation. Then I would like to say a big thank you to my supervisor Dr. Dan Turner, from whom I have learned so much these past two years. He was always there to provide guidance and mentoring and I could not have asked for a more fun and helpful supervisor.

Dedications

I dedicate this thesis to my grandfather, Dr. Ivor Vincent Pols.

Contents

List of Figures

List of Tables

Nomenclature

Symbols

b_{id}	The number identifying a bin
$b_{(x,y,z)}$	A bin's coordinate along the specified dimension
$B_{(x,y,z)}$	The total amount of bins along the specified dimension
d	Amount of dimensions in a space
δ	The influence length of mesh points (horizon)
E	Scalability of algorithm
H_x	The family of x, all the mesh points within δ of x
L_s	Minkowski distance norm
L_1	Manhattan Minkowski distance
L_2	Euclidean Minkowski distance
L_∞	Chebyshev Minkowski distance
N	The set of mesh points in a problem
p	Amount of processors in a system
S	Speedup of algorithm
T_{all}	The total time combined of all p processors in a parallel algorithm
T_s	Serial run-time of an algorithm
T_p	Parallel run-time of an algorithm
T_o	Total overhead time of a parallel algorithm

X	A set of objects in metric space
x	A reference mesh point in the problem
x_i	A reference mesh point coordinate along the i dimension $(0 < i \leq d)$
y	A candidate neighbour mesh point in the problem
y_i	A candidate neighbour mesh point coordinate along the i dimension $(0 < i \leq d)$

Abbreviations

CPU **C**entral **P**rocessing **U**nit

CZE **C**ohesive **Z**one **E**lement

FEM **F**inite **E**lement **M**ethod

FLOPS **F**loating **P**oint **O**perations **P**er **S**econd

GIS **G**eographic **I**nformation **S**ystem

GPU **G**raphics **P**rocessing **U**nit

LEFM **L**inear **E**lastic **F**racture **M**echanics

MIMD **M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams

MISD **M**ultiple **I**nstruction streams, **S**ingle **D**ata stream

MPI **M**essage **P**assing **I**nterface

PC **P**ersonal **C**omputer

SISD **S**ingle **I**nstruction stream, **S**ingle **D**ata stream

SIMD **S**ingle **I**nstruction stream, **M**ultiple **D**ata streams

XFEM e**X**tended **F**inite **E**lement **M**ethod

Chapter 1

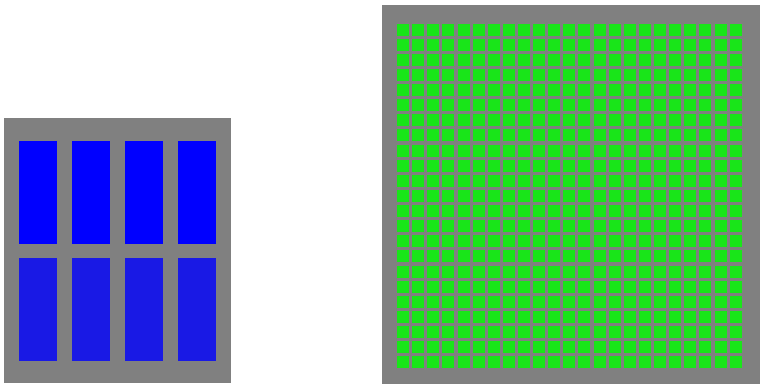
Background and Motivation

1.1 Introduction

To support modern research in the sciences and engineering extensive computational resources are required. Computational simulations, along with theory and experiments, have become a major building block towards scientific advancement. Numerical modelling is more flexible and less costly than experiments. The simulation of natural processes has increased our understanding of the natural world. Our ability to model more complex and larger natural processes keeps increasing. The sequential processing speed of a microchip however, is limited by thermodynamic considerations such as heat dissipation and power consumption. The frequency at which processors operate has plateaued since silicon chips have reached the material's limit in heat dissipation. The only means of reaching growing computational demands are through employing parallel computing. All the fastest computers in the world today are highly parallel systems. Even on a smaller scale, parallel clusters are the most economical way of achieving the required computational demand for small research groups and groups in industry. Therefore, it is prudent that existing algorithms must be adapted to take advantage of parallel computing. This thesis focuses on how to do neighbour list construction for solid mechanics methods on heterogeneous parallel architectures. This is a key step in any meshless or nonlocal mechanics method.

1.2 Heterogeneous Architectures

Parallel computing comes in many forms. A recent development (circa 2001) was that of using graphics processing units (GPUs) as accelerators coupled to central processing units (CPUs). The CPU and GPU work in tandem as the CPU controls the offloading of compute-intensive tasks to the GPU. The CPU also manages the copying and retrieval of data to and from the GPU memory. CPUs and GPUs together are referred to as a hybrid or heterogeneous computing architecture.



CPU: Multiple cores GPU: Thousands of cores

Figure 1.1: A GPU is massively parallel with thousands of cores

1.2.1 Development of GPUs

GPUs were originally designed as fixed-function processors that excel at three-dimensional graphics processing for gaming applications. Since then the GPU has evolved into a massively parallel programmable processor with vendors focusing on making their GPUs more accessible to programmers and researchers.

Over the years the capabilities of the GPU as a programmable processor has steadily increased. At first GPUs generated graphics with a simple non-programmable graphics pipeline. The input to the pipeline are vertex coordinates (usually representing triangles) in a three dimensional coordinate system. The GPU then transforms these vertices through a series of steps and finally outputs the colour value of every pixel to the screen. As every pixel's colour value can be calculated individually, the GPU processor evolved to be massively parallel. Conventional CPUs today have multiple cores, whereas GPUs consists of many thousands of smaller cores as shown in Figure 1.1. As computer graphics improved the operations to calculate the pixel values became more complex and were replaced by user-specified programs. This allowed programmers to attempt writing general-purpose code for the GPU. The data however, still had to be specified in terms of graphics primitives, which made programming very inaccessible. As the technology progressed, GPU vendors developed platforms that gave programmers and researchers easier access to utilise the computing power of GPUs. Today the GPU is a fully programmable stream processor.

1.2.2 Examples of GPU Computing

Following the adoption of the GPU as a general purpose streaming processor, researchers have adapted many algorithms to the GPU architecture. Krüger and Westermann (2003) developed a framework for the implementation of linear algebra operators on the GPU. They focus on solvers for sparse matrices. Fatahalian *et al.* (2004) evaluated the perfor-

mance of matrix-matrix multiplication. Govindaraju *et al.* (2005) developed a visibility ordering algorithm for non-overlapping geometric objects. Their algorithm rearranges objects in a front-to-back or back-to-front order given a certain viewpoint. Galoppo *et al.* (2005) developed an algorithm for solving dense linear algebra systems on the GPU. They demonstrate their algorithm by computing fluid flow simulations. Bustos *et al.* (2006) developed a nearest neighbour search algorithm on the GPU for database applications, stating their algorithm performs several times faster than an equivalent CPU implementation. Zhou *et al.* (2008) presents an algorithm for constructing k-d trees on a GPU. A k-d tree is a space-partitioning data structure (described in section 3.4.2) for organizing points in a d-dimensional space. The above are examples of sorting, linear algebra and database query algorithms that have been implemented to run on the GPU architecture. Many more exist and as research and development into heterogeneous systems continue many more will be developed.

1.2.3 Towards Exascale Computing

The considerable additional computing power that GPUs offer has led to much research in the last decade to develop algorithms that utilise the GPU. As general purpose computing on GPUs keeps improving, it will become easier for programmers to harness this computing power. Currently the worlds fastest supercomputers operate in the range of 10^{15} floating point operations per second (peta FLOPS). The next supercomputing threshold is to reach exascale speeds (10^{18} FLOPS). It is believed that exascale systems will only be obtainable around about the year 2020 and that such speeds will require heterogeneous systems. Most of the current leading supercomputers in the world are already heterogeneous. Even on a much smaller scale, GPUs provide a cost effective way of substantially increasing the computational power of desktop PCs. It is therefore important to develop new algorithms to take full advantage of GPUs.

1.3 Overview of Mechanics Methods that Require Neighbour List Construction

In the field of solid mechanics, various methods have been developed for the numerical modelling of material behaviour. An important distinction is that methods can either be local or nonlocal. This distinction specifies whether a method has an internal length parameter or not.

In local methods, such as classical continuum mechanics, the material is discretised into finite elements with nodes. The field values are solved at the nodes. The field values in an element between nodes are then solved using interpolation functions that are derived for the specific element being used. Elements are only influenced by adjacent elements,

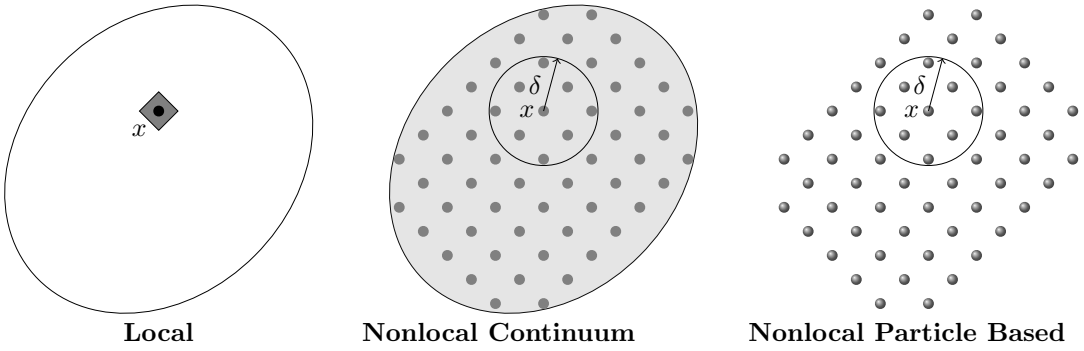


Figure 1.2: An illustration of a local, continuum nonlocal and particle based nonlocal method

which are found through an elemental connectivity matrix. There is no internal length parameter that accounts for elements further away.

In nonlocal methods however, the domain is discretised into mesh points or particles. Weight functions determine the influence that surrounding points have on each other. Mesh points are considered to influence each other if they are within a certain distance from each other. This distance is defined by some characteristic length δ that creates a sphere of influence around each mesh point called the *horizon*. All the points that are within the horizon of point x are called the family (H_x) of x. The families of each point in the domain must be obtained before the relevant mechanics calculations can be done. Therefore, a *neighbour list* must be constructed for each mesh point in the problem.

Nonlocal methods can further be categorized into methods that represent the material body as a continuum or as a particle cloud. For nonlocal continuum methods the material body is discretised into mesh points. The term mesh points is used because there is still a background mesh present from where the mesh points are mapped. Nonlocal particle methods refer to when there is no background mesh present. The material is discretised purely into a particle cloud. Both continuum and particle based nonlocal methods require a neighbour list construction. In this thesis therefore, the terms mesh points and particles are interchangeable. The term meshless refers both to a continuum nonlocal (regardless of the background mesh) and particle based nonlocal method. Therefore the terms meshless and nonlocal are also interchangeable. Figure 1.2 illustrates the difference between the methods explained above.

1.3.1 Examples of Meshless Methods

One of the first meshless methods developed was smooth particle hydrodynamics (SPH) by Lucy (1977) and Gingold and Monaghan (1977). It was developed for problems in astrophysics, and later on in fluid dynamics. SPH was first used in solid mechanics by Allahdadi *et al.* (1993) to model impact problems. Other meshless methods applicable to solid mechanics are the Element-free Galerkin (EFG) method, developed by Belytschko

et al. (1994) for elasticity and heat conduction problems. One year later the Reproducing Kernel Particle Method (RKPM) was developed by Liu *et al.* (1995). Another method applicable to solid mechanics is the hp-cloud method, developed in 1996 by Duarte and Oden (1996) and Liszka *et al.* (1996). The partition of unity finite element method (PUFEM), which is very similar to the hp-cloud method, was developed by Melenk and Babuška (1996). The Meshless Local Petrov-Galerkin (MLPG) method developed by Atluri and Zhu (1998) and (Shen, 2002) is another method used in solid and fluid mechanics. A method mainly used in fluid flow problems, is the Finite Point method developed by Onate *et al.* (1996), Oñate and Idelsohn (1998), and Löhner *et al.* (2002).

Another method, showing great promise in progressive material failure modelling, is peridynamics. It is a relatively new method that arose from the reformulation of the elasticity theory by Silling (2000). In peridynamics the continuum theory is reformulated in terms of integral equations, rather than differential equations. Integral equations stay valid as discontinuities form in the body. Progressive material failure can therefore be modelled without requiring additional crack growth criteria as is the case in conventional methods. Peridynamics is a continuum model where the force on a point is calculated from all the pairwise forces comprising the family of that point. All the above methods are nonlocal and require a neighbour list construction.

1.3.2 Reasons for Developing Meshless Methods

The finite element method (FEM) has long been used with great success in modelling stresses, strains, and displacements in the field of solid mechanics. FEM is however, not without limitations. The underlying theory of classical continuum mechanics assumes the body to remain a continuum as it deforms. In classical continuum mechanics, spatial partial derivatives are used to represent relative displacements between points. Partial derivatives with respect to spatial coordinates become undefined along discontinuities such as cracks. Therefore, the mathematical formulation used when modelling a material with FEM breaks down when discontinuities start to appear.

Incorporating material failure within classical continuum mechanics started with the pioneering study by Griffith (1921). Griffith showed that stresses at the tip of a crack become infinite. His study led to the concept of Linear Elastic Fracture Mechanics (LEFM). In LEFM, a pre-existing crack in the material is necessary and the crack growth is governed by propagation criteria. Furthermore, Eringen *et al.* (1977) showed that the crack size plays an important role in determining fracture resistance, however solutions from the classical continuum theory or LEFM are independent of crack size. This is because the classical continuum theory has no internal length parameter. When using FEM to model LEFM special elements are required to handle the infinite stresses at the crack tips. Furthermore, to handle the break down of spatial partial derivatives at cracks, the

body needs to be redefined with the cracks as boundaries. All of these difficulties, the required pre-existing crack tip, external propagation criteria, lack of an internal length parameter, infinite stresses at crack tips, and re-meshing or redefining of the body as the crack grows makes it clear that FEM based on conventional continuum mechanics cannot model complicated failure problems.

Various techniques have been proposed to improve the capabilities of FEM to model material failure. However all the techniques require some external information or have some limiting factors. Dugdale (1960) and Barenblatt (1962) introduced the cohesive zone concept. This lead to the introduction of Cohesive Zone Elements (CZE) by Hillerborg *et al.* (1976) for the Mode-I fracture mode and by Xu and Needleman (1994) for a mixed-mode fracture. Cohesive zone elements are surface elements that are placed along normal element boundaries. Crack growth can then only occur between normal elements where the CZE are. It has been found however, that with decreasing mesh size the amount of cohesive elements increase, leading to softening of the material properties. Also, the crack paths are highly sensitive to the mesh alignment (Klein *et al.*, 2001). Another concept introduced to address these difficulties is the eXtended Finite Element Method (XFEM). XFEM allows cracks to propagate on any surface within an element rather than just on the boundaries of the elements. XFEM has had success in modelling some fracture problems, however it still requires some external crack growth criteria.

It can be seen that failure prediction has some difficulties when using the traditional FEM or various enhanced finite element techniques. One of the objectives of meshless methods are to improve on the limitations of mesh based methods such as FEM. Especially progressive material failure, as meshless methods are better suited to handle discontinuities. Some advantages and disadvantages that meshless methods have when compared to FEM (Nguyen *et al.*, 2008) are:

- Advantages
  - Meshless methods are inherently better suited to handle discontinuities such as crack propagation, shear bonds and phase transformation.
  - Large deformations can be handled more robustly.
  - Meshless methods are not concerned with mesh alignment issues.
- Disadvantages
  - Essential boundary conditions are much more complicated to handle in meshless methods than in mesh-based methods.
  - The computational costs of meshless methods are generally much higher than mesh-based methods. The neighbour list construction, if not done properly, is a major contributor to computational cost.

1.4 Summary

Today's supercomputers are striving to reach exascale speeds. It is believed that the only way of doing this is through employing heterogeneous computing architectures. This and the cost effectiveness of GPUs will ensure that GPU computing will grow in the future.

Meshless methods are not on par with FEM in calculating stress, strain and displacement. They are however, inherently better at modelling progressive material failure. Therefore, meshless methods are still a very important research field. A key step in any meshless method is that of calculating the neighbour list. If not done properly, the neighbour list construction is a major contributor to the computational cost of any meshless method.

This thesis will focus on developing a scalable neighbour search algorithm that takes advantage of heterogeneous architectures. The algorithm must run on CPU and GPU processors and be scalable to massively parallel systems to enable extreme scale simulations.

1.5 Thesis Structure

This thesis is structured as follows: Chapter two gives an overview of the different types of parallel architectures and parallel algorithm performance measurements. Chapter three gives a review of the different types of proximity problems that exist, and the numerous data indexing structures that have evolved to solve them. The focus is specifically on the all-pairs fixed-radius nearest neighbours search or self-spatial join search. This search is needed to compute the neighbour list. In chapter four the first algorithm, designed for a CPU distributed memory system, is explained and tested. In chapter five we explain and test the same algorithm altered to run on a heterogeneous system. The thesis concludes with chapter six were we discuss and compare the results obtained from chapters four and five.

Chapter 2

Parallel Computing

2.1 Introduction

In parallel computing there are many different configurations available, depending on the memory layout and processor types. There are three different types of parallel systems, namely shared memory, distributed memory and heterogeneous systems. Real world systems usually consist of a combination of these. In this chapter we will explain the processor layout and memory organization of these three types of systems.

We start the chapter by stating some definitions and terminology used in parallel computing. We then give an overview of the three types of parallel architectures available. Furthermore, we also give performance measurements used to measure parallel algorithm performance. The definitions given here are used in subsequent chapters to explain the algorithm design and performance measurements.

2.2 Terminology and Definitions

In the traditional *von Neumann* model of computation, a single instruction stream is executed on a single data stream. When performing computations in parallel however, many more possibilities arise. A taxonomy of computer architectures is given by Flynn (Flynn and Rudd, 1996) as follows:

1. SISD: single instruction stream, single data stream
2. SIMD: single instruction stream, multiple data streams
3. MISD: multiple instruction streams, single data stream
4. MIMD: multiple instruction streams, multiple data streams

Conventional computation described by the *von Neumann* model is described in Flynn's taxonomy by SISD. Most parallel computers have a MIMD architecture however, a more practical way of programming for MIMD architectures are called SPMD, which refers to *single program stream, multiple data streams*. A single program is executed on all the processors with different sets of data. The program on each processor may then branch and follow different paths. MISD is not very useful and is rarely if ever used. SIMD refers to data-parallel programming where the same single-line (no branching of code) instructions are executed on different data streams. In high performance parallel computing the following terms are usually used:

Node: A compute node refers to a server rack that can contain multiple processors and memory locations. Compute nodes are connected to each other through a network.

Processor: A processor refers to a single physical microchip on the compute node. A processor can consist of one or more cores.

Core: A core refers to the most basic computational unit that can only run a single instruction stream.

2.3 Parallel Architectures

2.3.1 Shared Memory

Shared memory implies that all processors have access to and share the same memory. In this case no communication between processors are required because they can all read and write from the same location. Shared memory systems are not very scalable because it becomes impractical for to many processors to have access to the same memory. Another difficulty in shared memory systems is the occurrence of race conditions. A race condition occurs when the output of a program varies from run to run although the same input data is supplied. This happens when the order in which two processors access the same memory location is variable from run to run. Shared memory systems are usually small scale and are mainly used to parallelize loops. The memory layout is shown in Figure 2.1.

2.3.2 Distributed Memory

Distributed memory implies that each processor has its own local memory. Data from other processor's memory can be obtained through network communication, usually through the Message Passing Interface (MPI) protocol. Distributed memory systems usually follow a SPMD programming model. Distributed memory systems are very scalable, with all the largest parallel computers in the world being distributed systems. The memory layout is shown in Figure 2.2.

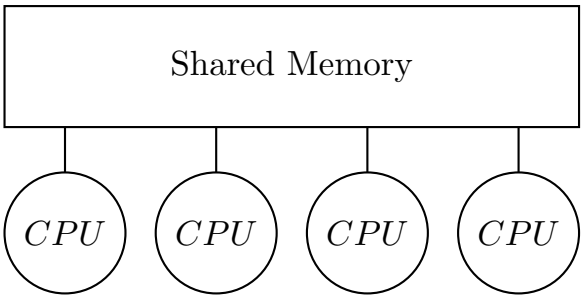


Figure 2.1: Memory layout of a shared memory system

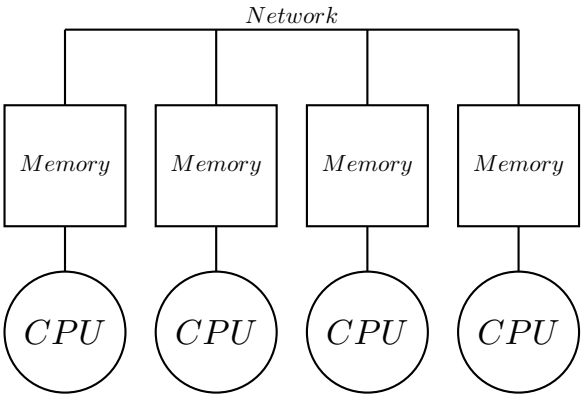


Figure 2.2: Memory layout of a distributed memory system

In distributed memory architectures the processors communicate with each other across the network through MPI. Various MPI message patterns exist to facilitate communication. The message patterns are as follows:

1. broadcast: one-to-all
2. reduction: all-to-one
3. multi-node broadcast: all-to-all
4. scatter / gather: one-to-all / all-to-one
5. total exchange: personalized all-to-all
6. scan or prefix
7. circular shift
8. barrier

Broadcast implies that the source processor sends the same message to the other $p-1$ processors. *Reduction* implies that all the data from the p processors are combined on the specified root processor through applying an associative operation to the data to obtain

a single result. A *multi-node broadcast* is a broadcast from all p processors. A *scatter* is analogous to broadcast, however the source processor sends dissimilar messages to the other $p-1$ processors. *Gather* is similarly analogous to reduction, however the data received by the root processor is concatenated instead of combined through an associative operation. *Total exchange* is analogous to a multi-node broadcast, however each processor exchanges a distinct message with each of the other processors. In the *scan* or *prefix* operation each processor produces a data value $(x_0, x_2, ..., x_{p-1})$ and an associative operation we will designate with $\oplus$. Each processor then combines all the data values from processor zero up to its own processor number k, with the given associative operation, as

$$x_k = x_0 \oplus x_1 \oplus ... \oplus x_k, \tag{2.3.1}$$

with $0 \leq k \leq p-1$.

A *circular shift* is when the data on each processor is shifted along a specified amount. The increment that the data is shifted is specified by a constant k. Each processor i $(0 \leq i \leq p-1)$ then sends it's data to processor $(i+k) \bmod p$. A *barrier* operation is used to synchronize all the processors. No processor is allowed to continue with the program unless all of them have reached the barrier.

2.3.3 Heterogeneous Architecture

A heterogeneous architecture refers to a system with more than one type of processor. Today the most popular co-processor used as an accelerator attached to the CPU is a GPU. Usually a single CPU handles the offloading of calculations to a single GPU. A GPU has it's own memory and is not aware of the shared or distributed memory set-up of its controlling CPU.

2.3.3.1 GPU Programming Model

GPUs are massively parallel processors. They evolved into massively parallel processors because of the demands of real-time three-dimensional graphics computation. A GPU can be seen as a streaming processor. The computation done on a GPU consists of many streams, with a stream representing an ordered set of data. The program that is executed on the GPU is called a kernel. The kernel is the function that is applied to each element in the data stream. A GPU is therefore data-parallel, following a SPMD programming model. Branching for the data elements in a stream are allowed, however severely reduce the performance of the GPU. Therefore kernels must be written with as few branching as possible. The ideal kernel function is a straight-line program with no branches, thus following a SIMD programming model. In GPU computing the host refers to the CPU and the device refers to the GPU. The flow of GPU computing is as follows:

1. The programmer defines a computation domain (the input data). The data is copied from the host memory to device memory. This is the input data that will be operated on by the kernel.
2. The CPU invokes the kernel, the function that the GPU will apply to the data streams.
3. The GPU divides the data into many parallel streams or threads of data and performs the kernel function on these threads.
4. The results of the calculation are copied back from device memory to host memory.

Each thread or stream can read and write to the GPU global shared memory. Threads cannot however, communicate directly with each other. The output of threads computed by the kernel can be stored in GPU global shared memory and be used as input for subsequent kernels, instead of being copied back to CPU memory.

The data-parallel operations present in distributed parallel computing can also be mapped to the GPU threads. These operations are:

1. **Scatter/Gather:** The threads, as mentioned above, can read from and write to GPU global memory. Therefore scatter and gather operations can be done between threads.
2. **Map:** A map operation is when one applies the kernel function to every element in a collection. This is a parallel for loop where each element is operated on concurrently. A map represents the outer for loop in a sequential program, as each thread can further loop in the kernel function.
3. **Reduce:** Repeatedly apply a binary associative operation to reduce all the threads to a single thread.
4. **Scan:** A scan or prefix operation can be done over all the threads as explained in section 2.3.2.

2.3.3.2 GPU Memory Layout

The memory layout of the GPU programming model is shown in Figure 2.3. This Figure illustrates a very simplified model of the memory, because in reality GPU memory is much more complicated. This layout is however sufficient to illustrate the important fact that any computations on the GPU requires the data to be transferred from CPU to GPU memory. If this communication of data is very time consuming, it might negate the additional computing performance provided by the GPU processor.

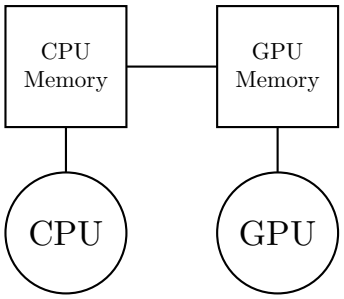


Figure 2.3: Memory layout of a heterogeneous system

2.4 Performance Metrics of Parallel Algorithms

In serial computations the performance of an algorithm is typically measured by the time T_s it takes to complete its computations. When the same problem is run in parallel with p processors the time it takes will be $T_p > \frac{T_s}{p}$. The parallel time T_p will never be equal to $\frac{T_s}{p}$ because there will always be overhead costs incurred from making an algorithm parallel. Let T_{all} be the total time spent by all the processors, then $T_{all} = pT_p$. The total overhead time T_o is then defined by $T_o = pT_p - T_s$ and represents the total time collectively used for things other than the algorithm calculations.

2.4.1 Sources of Parallel Overhead

Parallel overhead is wasted computational time and energy. Large scale computer systems are expensive and require a large power input. It is important that algorithms are made to have as little overhead as possible, thereby maximizing the gains obtained from running the computational problem concurrently.

The amount of parallelism that can be achieved by an algorithm is inherent to the problem being solved. A problem can be classified as *fine-grained*, *coarse-grained* or *embarrassingly parallel*. These three classifications represent the amount of communication that is required among a problem's subtasks. A problem is fine-grained when its subtasks need to communicate very frequently. A problem is coarse-grained when its subtasks need to communicate only occasionally and embarrassingly parallel when its subtasks rarely or never have to communicate.

There are three sources of parallel overhead, resulting from operations that would not have normally occurred during serial execution.

Communication: Transferring data between processors is usually the dominant source and as described above, the amount required is usually inherent to the problem.

Idle Processors: This results from communication where a process may wait to receive a message or waits at a barrier. Idle processors also result from unequal load bal-

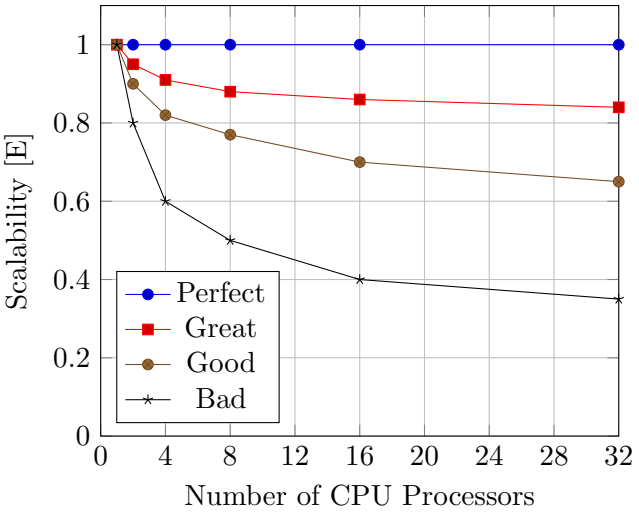


Figure 2.4: An example of different scalability performances

ancing, where the workload of the problem is unequally divided among the available processors.

Excess Computation: Additional processor instruction and memory access cycles are required to determine what data to communicate and when. Sometimes to limit communication processors may all execute the same computation.

We will investigate the parallel overhead of the designed algorithms in chapters four and five to determine how well they will perform on massively parallel systems.

2.4.2 Scalability and Speedup

Two, very similar, metrics used to measure the parallel performance of an algorithm are *scalability* and *speedup*. Scalability measures how efficient each processor is by calculating the percentage of computational resources that is being used effectively and is defined as

$$E = \frac{T_s}{pT_p}. \tag{2.4.1}$$

This leads to upper and lower bounds as $0 < E < 1$. A scalability of one indicates perfect efficiency with no parallel overhead, which is not possible. As E becomes less it implies worse efficiency and utilization of the available computing power. Scalability is measured as p increases for a given problem size. Figure 2.4 shows examples of scalability performance. The objective is to keep the algorithm scalability as close to one as possible.

Speedup is very similar to scalability and is calculated as

$$S = \frac{T_s}{T_p}. \tag{2.4.2}$$

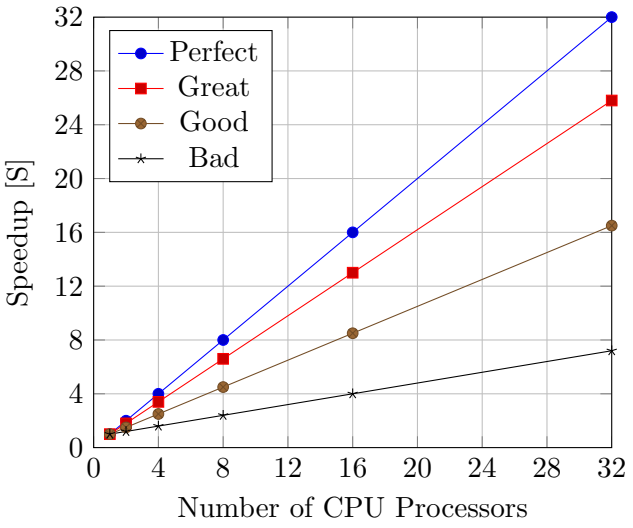


Figure 2.5: An example of different speedup performances

Speedup is measured as p increases for a given problem size and is bounded by $0 < S < p$. Speedup represents the performance gain that was obtained by going parallel. The upper bound is p-times the serial time, as one cannot gain more than p-times the performance with p-times the amount of computational power. Figure 2.5 shows examples of different speedup performances. The ideal speedup would be as close to the $S = p$ line as possible.

2.4.3 Iso-Efficiency Curves

A useful plot to analyse the performance of an algorithm in parallel is the iso-efficiency curves, which are contour lines of scalability plotted against the number of processors and problem size. The number of processors and problem size are on the x-axis and y-axis respectively, while scalability is on the z-axis. Iso-efficiency curves show what level of efficiency the algorithm will achieve for a given problem size and number of processors.

2.4.4 Throughput

Throughput is another performance measure, though not necessarily for parallel performance. Throughput is the amount of work that can be done per time unit and is defined as the problem size divided by the time it took for the problem to complete:

$$\text{Throughput} = \frac{N}{T_p}, \tag{2.4.3}$$

where N is the amount of mesh points and T_p is the average runtime of the p processors in parallel.

2.5 Summary

In this chapter an overview was given of parallel computing concepts that we will use in discussions to follow. We started with an overview of the definitions and terminology used in parallel computing. We also gave an overview of the different types of parallel architectures available. These are shared memory systems, distributed memory systems, and heterogeneous systems. For all three we discussed the memory layout and programming model. In chapter four we will describe a distributed CPU algorithm that we designed and in chapter five a heterogeneous algorithm. Furthermore, we reviewed parallel performance metrics namely: scalability, speedup, iso-efficiency curves, and throughput. These metrics will be used to measure and compare the designed algorithms.

Chapter 3

Fixed-Radius Nearest Neighbours Searching

3.1 Introduction

This chapter starts with a concise overview of the types of spaces and distance functions that are relevant to this study. A space contains a set of objects and has a distance function defined that returns a distance between two objects. A distance can be real-valued or simply a similarity rating between objects. There exist various types of spaces and distance functions, some of which are described below. There are many more that are not covered.

Secondly a formal definition is given of the three types of proximity problems that have arisen over the years. These problems are applicable to a vast amount of fields ranging from computational geometry to geographic information systems (GIS), to name but a few. Here we focus on proximity problems applicable to meshless methods in solid mechanics, specifically the all-pairs nearest neighbours search.

Then a description and classification of different types of indexing structures are given. The indexing structure is a key focus of this study, as it is used to solve proximity problems in an optimised way. This is done through organizing the objects in space in some way. We will look at different strategies that exist to create indexing structures for spaces and distance functions.

3.2 Basic Types of Spaces and Distance Functions

To understand proximity problems a formal definition of the types of search spaces and distance metrics that are applicable to this study are given. A good overview of the

different types of spaces and distance functions are given in Chávez and Navarro (2001) and Castelli (2004).

3.2.1 Metric Space

Metric space is any space which contains a set of objects X and has a distance function $d(x, y)$ that returns a non-negative *distance* between two objects x and y of X. The *distance* here is not necessarily the actual distance between two objects, but can be how similar the objects are to each other. Distance functions in metric spaces must have the following properties:

1. Positiveness: $\forall x, y \in X, d(x, y) \geq 0$
2. Symmetrical: $\forall x, y \in X, d(x, y) = d(y, x)$
3. Reflexive: $\forall x \in X, d(x, x) = 0$
4. Strict Positiveness: $\forall x, y \in X, x \neq y, \Rightarrow d(x, y) > 0$
5. Triangle Inequality: $\forall x, y, z \in X, d(x, y) \leq d(x, z) + d(z, y)$

3.2.2 Vector Space and the Minkowski Distance Metric

If the set of objects X has real-valued coordinates, the metric space becomes a vector space. A d-dimensional vector space has d real-valued coordinates $(x_1, x_2, ..., x_d)$. In vector spaces there are a number of different distance functions but the most widely used is the Minkowski (L_s) distance norm, which is defined as

$$L_s\{(x_1, x_2, ..., x_d), (y_1, y_2, ..., y_d)\} = (\sum_{i=1}^{d} |x_i - y_i|^s)^{\frac{1}{s}}. \tag{3.2.1}$$

The L_1 metric is called the *Manhattan* or *block* distance which measures the sum of the differences between the coordinates. It is called the *Manhattan* or *block* distance because in two dimensions it corresponds to the distance one has to walk between two points in a city of rectangular blocks. It is defined as

$$L_1\{(x_1, x_2, ..., x_d), (y_1, y_2, ..., y_d)\} = \sum_{i=1}^{d} |x_i - y_i|. \tag{3.2.2}$$

The L_2 metric is called Euclidean distance and refers to Euclidean space. Euclidean space is the most common of vector spaces and Euclidean distance can be seen as the real world distance where the distance is defined as

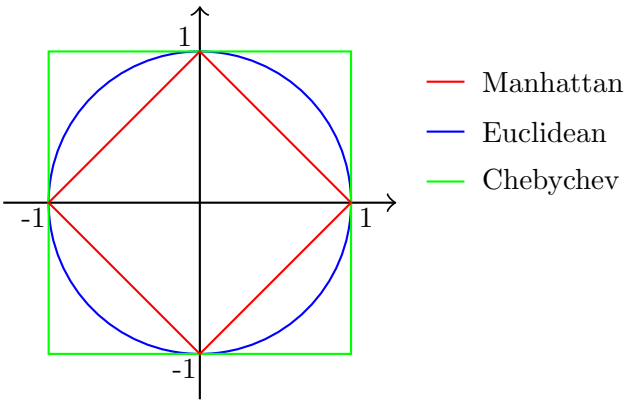


Figure 3.1: Unit spheres defined for Manhattan, Euclidean and Chebychev distances respectively

$$L_2\{(x_1, x_2, ..., x_d), (y_1, y_2, ..., y_d)\} = (\sum_{i=1}^{d} |x_i - y_i|^2)^{\frac{1}{2}}. \tag{3.2.3}$$

Another Minkowski distance is the Chebychev (L_∞) metric, taking the limit of L_s as s goes to infinity. In this case the distance between two points become the maximum difference along a coordinate as

$$L_\infty\{(x_1, x_2, ..., x_d), (y_1, y_2, ..., y_d)\} = \max_{i=1}^{d} |x_i - y_i|. \tag{3.2.4}$$

Figure 3.1 shows unit spheres (or balls) for the Manhattan, Euclidean, and Chebychev distances respectively. The different balls can be interpreted as follows: A ball centred at a point x having a radius r is the set of points having distance r from x. A ball in Euclidean space is the familiar sphere. A ball, according to the Chebychev distance, is a hyper-square aligned with the coordinate axis that inscribes the Euclidean sphere. A ball in the Manhattan metric is a square with vertices on the axis coordinates that is inscribed by the Euclidean sphere.

An interesting remark about Minkowski distances can be derived from the spheres: With the L_1 metric, the contribution of the individual differences along each dimension is equally summed towards the total distance. As s increases towards ∞ however, the total distance is increasingly determined by the larger of the differences along each dimension. This continues until for the L_∞ metric the total distance is defined as the maximum distance along a dimension.

All of the Minkowski distances described above are very similar to each other from the viewpoint of proximity queries. Some of the algorithms described below were developed for a specific Minkowski metric in mind, but that does not exclude all other Minkowski metrics to be used with the same algorithm. Cost-wise the Euclidean distance is the most

expensive to perform and takes $O(d)$ operations. Chebychev distance is less computationally expensive, however still requires $O(d)$ operations while the Manhattan distance is as computationally expensive as calculating a squared Euclidean distance. This study focuses on real world Euclidean vector space, as we are interested in the numerical modelling of real world objects in solid mechanics. The discussion in the rest of this chapter pertain to vector spaces only.

3.3 Types of Proximity Queries

In the extensive technical literature available on proximity queries many different wordings of definitions for the types of searches have arisen. Definitions given here might not correspond word for word to definitions given elsewhere, but the concepts are the same. Proximity queries can be grouped into three main classes. These problems are not only limited to points in space, but are applicable to any arbitrary geometric shape:

1. **Range Search:** Given a set of N points in d-dimensional space, find all points within the specified range. Ranges can be specified on any or all of the dimensions, and can be in any shape: square, rectangular, circular or arbitrary. Range queries initially came from database information retrieval where a database consists of N records with d attributes. Each record can then be thought of as a point in d-dimensional space.

2. **k-Nearest Neighbour Search:** Given a set of N points in d-dimensional space, find the k nearest points to a specified query point.

3. **Fixed-Radius Search:** Given a set of N points in d-dimensional space, find all the points that are within a distance δ from a query point. A slight extension of this problem is called the **All-pairs Fixed-Radius Nearest Neighbours Search** by (Aref and Barbara, 1995) or **Self-Spatial Join Search** by (Noske, 2004) and is defined as: Given a set of N points in d-dimensional space, find all pairs of points within a distance δ from each other.

Solving any of the above queries will result in the same general strategy being followed. Which is creating an indexing structure of the space. An indexing structure serves to limit the amount of actual distance calculations that are computed between points or objects. Without an indexing structure the *brute-force* or *sequential-scan* method must be followed, where each point is compared to all the other points each time a query is answered. Instead, an indexing structure is built during the preprocessing phase, leading to reduced search times during the query phase.

For this study the primary field of focus is nonlocal solid mechanics numerical modelling methods. In such methods the materials being modelled are discretized into mesh points. Each mesh point is then assumed to only be influenced by the neighbouring points that are within the horizon (δ) of that point. Therefore an *all-pairs fixed-radius nearest neighbours search* must be done to determine the neighbouring mesh points of all the N mesh points in the model before the relevant mechanics calculations can be done. Thus, we will look at all indexing structures from the perspective of the solving the *all-pairs* search. Note however, that indexing structures made for a specific type of proximity problem mentioned above can generally be used for the other types of proximity problems as well. All the indexing structures mentioned below, unless stated otherwise, can be used for all three types of proximity searches.

The fixed-radius nearest neighbours search problem is an important step in many areas of interest other than numerical mechanics modelling, ranging from geographical information systems (GIS), air traffic control, pattern recognition, molecular dynamics and N-body problems to similarity searching in object databases. Therefore the algorithms designed here are applicable to many other fields of study.

Another thing to note regarding the *all-pairs* problems is the *sparsity condition*. The sparsity condition arises when a ball of radius δ in the space contains no more than some c points. Thus, the sparsity condition implies that all the points in the space are spread out and not clumped together. For most of the engineering problems that we are interested in the points will be spread out, and the sparsity condition will be present. It is important to note that all *all-pairs fixed-radius nearest neighbours search* algorithms are designed assuming the sparsity condition will be present. If the condition is not present all points may be positioned in a clump of size δ. If this is the case there will be $O(N^2)$ pairs of points present. Any neighbour searching algorithm will thus have a minimum running time of $O(N^2)$, which is equivalent to the sequential search brute-force approach of the problem. Thus, if the sparsity condition is not present, it precludes the design of an algorithm.

3.4 Types of Indexing Structures in Vector Space

Vast amounts of indexing structures have been developed over the years, each with their own advantages and disadvantages, preprocessing times and query times. Various ways of categorizing indexing structures exists. Here we divide them into non-hierarchical methods, recursive partitioning methods, and a third category for miscellaneous methods that do not fit into the previous two.

3.4.1 Non-Hierarchical Methods

Methods described in this category are the *inverted list* or *projection*, *space-filling curves*, and *fixed-grid* methods. The *brute-force* or *sequential-scan* method described above also belong to this category.

We start with the *inverted list* technique described by Knuth (1973). This technique is referred to as the *projection* method by Bentley and Friedman (1979). The method involves keeping, for each dimension, a sequence of the points in space sorted according to that dimension. Thus, the indexing structure consists of d sorted arrays of length N. The search is done by looking at one of the d arrays and eliminating all candidate points outside the search range for that dimension. All the remaining points are then searched exhaustively. This method works well when the points are uniformly spaced in a certain d-dimension. Looking at that d-array will then eliminate most of the points required to further search through. The d sorted arrays can be constructed in $O(dN\log N)$ time according to Bentley and Friedman (1979).

Space-filling curve methods work by mapping all d-dimensional points onto a real line and from there using one-dimensional indexing structures for answering approximate queries. The mapping is done by interleaving a space-filling curve through all the points in the space. The ordering of the points on the real line is the positions of the points on the space-filling curve. Various space-filling curves exists such as the *Hilbert* or *Peano-Hilbert* curve (Sagan, 1994), and *Morton ordering* or *z-ordering* (Morton, 1966).

One of the first mentions of the *fixed-grid* method was by Bentley *et al.* (1977). They divide the search space into equally sized non-overlapping hyper cubes by overlaying a fixed-grid over all the points. Hyper cubes are cubes in higher dimensions where $d \geq 4$. We call the subspaces the hyper cubes create *bins*. The points that lie in the same hypercube are grouped together in the same bin. The bin that a point occupies can be determined in constant time. The fixed-grid method works by making the bin side-lengths (i.e. grid spacing) equal to δ. A point x is inside a specific bin that has $3^d - 1$ adjacent bins. To construct the neighbour list of point x an exhaustive search only has to done for points that occupy the same bin as point x or are in one of the $3^d - 1$ adjacent bins. Bentley *et al.* (1977) prove that the number of distance calculations required is linear ($O(N)$) given that the sparsity condition is present, and propose three data structures to handle the fixed-grid index namely, a hash table, a balanced tree, and a multidimensional array. The hash table and multidimensional array's build time is $O(dN)$ while the balanced tree's build time is $O(dN\log N)$. The distance measure they use is the L_∞ metric, where a hyper sphere with radius δ becomes a hypercube with side length δ. There algorithm can also be extended to other L_s metrics.

The fixed-grid method works well for very low dimensions ($d \leq 3$), however in higher

dimensions ($d \geq 4$) it starts suffering from poor space utilization as most bins become empty (Castelli, 2004). Therefore, *grid-files* were developed as extensions of the *fixed-grid* to accommodate higher dimensions better. More information on *grid-files* can be found in Tamminen (1982), Hinrichs and Nievergelt (1983), and Nievergelt *et al.* (1984). For this study the maximum amount of dimensions will always be three.

3.4.2 Recursive Partitioning Methods

The following indexing structures recursively divide the objects in space according to some criteria, and are usually stored using a tree data structure. Notable trees we will be describing here are the *K-d tree*, *Quadtree*, and *R-tree* but many more exist.

The k-d tree or multidimensional binary tree was first developed by Bentley (1975). A k-d tree recursively divides the objects or points in space by a $d-1$ dimensional hyper plane perpendicular to an axis. The hyper plane splits the data creating two subsets of equal size. The axis chosen to divide along can be alternate, or it can be calculated according to the layout of the data. Each internal node in the tree thus has two children nodes. With the initial k-d tree points were stored in internal nodes and leaf nodes. Subsequent improvements made that all points are stored in leaf nodes only. Many variations and extensions of the k-d tree exist. In Bentley (1990) the k-d tree is improved to handle semi-dynamic point sets better. In Al-Furaih *et al.* (2000) the authors represent several algorithms for the parallel construction of a k-d tree for distributed memory systems.

Quadtrees (Samet, 1984) recursively divide the search space into 2^d subspaces by splitting equally each axis in half during each division step. Each internal node therefore has 2^d children nodes. In Andreica and Tapus (2012) the authors construct a multidimensional quad-tree index to answer fixed-radius nearest neighbour queries consisting of complex arbitrary geometries. They build a quadtree index of complex geometrical shapes in any d-dimensional space. Thereafter, given a query point or polyhedron q and a search radius r, they quickly enumerate all objects within distance r from q. There quadtree data structure can also be distributed over multiple machines. Given a query, the nodes of the quadtree containing the reduced search space is computed and sent to all the machines, which then send back all objects contained in those nodes as candidates.

R-trees (Guttman, 1984) work by recursively dividing the space into hyper rectangles. The hyper rectangle of an internal node encloses the hyper rectangles of its children nodes. Hyper rectangles on the same level of the tree may possibly overlap creating complexities not present in the above-mentioned trees. The points in space are all stored in the leaf nodes of the tree. Over the years there have been many extensions to the R-tree. The R^+-tree (Sellis *et al.*, 1987) minimizes the overlap of the hyper rectangles on the same level by using splitting rules that do not allow the hyper rectangles to overlap.

The R^*-tree (Beckmann *et al.*, 1990) improves upon most of the performance issues of the original R-tree. In Sharifzadeh and Shahabi (2010) the R-tree is combined with the Voronoi diagram, creating a unique structure for solving nearest neighbour queries.

A recursive method that does not result in a tree data structure, called divide and conquer, is described by Bentley and Shamos (1976) to solve the *all-pairs* search specifically. Their method works by recursively dividing the point set N in half through a $d-1$ dimensional plane. All the points on either side of the plane within a distance δ are then projected onto the plane and neighbouring pairs are calculated.

3.4.3 Miscellaneous Methods

In Chazelle (1983), the author combines several range searching procedures to solve the *all-pairs* search. The algorithm only works for points in a plane and not for higher dimensions. It works by using the planar point location algorithm of Preparata (1981), along with range searching procedures developed by Bentley and Ottmann (1979), Edelsbrunner (1980), and McCreight (1980). All the circles created by the N points on the plane create a graph, which is stored using a tree. Preparata's method of planar point location is then used to return in which region of the graph or in what circle the query point is located in. The preprocessing time for creating the indexing structure of Chazelle's algorithm is $O(N^2 \log N)$, while the query time for a single point is $O(I + \log N)$ with I being the number of pairs reported.

Dickerson and Drysdale (1990) developed an algorithm that uses Delaunay triangulation to solve the *all-pairs* search. The Delaunay triangulation is the dual of the Voronoi diagram first introduced by Shamos and Hoey (1975). Their algorithm works by first constructing, as a preprocessing phase, the Delaunay triangulation of the point set in a plane in $O(N \log N)$ time. Thereafter the neighbour search can be done in $O(N + I)$ time, with I being the number of pairs reported. Their algorithm has an advantage over the linear time *fixed-grid* algorithm by Bentley *et al.* (1977) in that after the preprocessing phase the Delaunay triangulation can accommodate various horizon lengths. Whereas the *fixed-grid* indexing structure set up by Bentley *et al.* (1977) is done for a fixed horizon.

The algorithm by Dickerson and Drysdale (1990) is further improved upon by Turau (1991). Turau added an extra information gathering step during the preprocessing phase to improve the search time of $O(N + I)$ to $O(I)$, where I is again the number of pairs reported. The preprocessing time remains $O(N \log N)$.

Kanda and Sugihara (2005) describes an algorithm they develop for doing range queries on a plane using the Voronoi diagram. Usually the shapes of range queries are square or circular. Their method works for any general query shape. Their algorithm compared, on average, favourably to the fixed-grid and k-d tree techniques for general query shapes.

3.4.4 Choosing an Indexing Structure

With the vast amount of indexing structures available in the literature, it is important to choose the appropriate one that will give the best performance. In various articles the authors discuss comparative results of non-hierarchical methods against recursive-based methods.

In Bentley and Friedman (1979), the authors give an overview of indexing structures that can be used for range searching. The methods they reviewed (among others) are: the brute-force approach, projection or inverted lists, fixed-grid method, and the k-d tree. The authors state that the fixed-grid method give the best results when the points are uniformly spaced, while the k-d tree is the best data structure to use when the points are more variably spaced.

In Artemova *et al.* (2011) the authors illustrate again that hierarchical indexing structures perform better than fixed-grid indexing structures where the point sets are more variably spaced. They compare neighbour list constructions for molecular based simulations where the point sets are typically much more varied than solid mechanics material modelling and find that the hierarchical based algorithms perform slightly better than the grid-based algorithm. Further, in Castelli (2004) the authors state that recursive based methods that result in tree data structures are much more amenable to handling higher dimensions than non-hierarchical methods.

Putting this all together the fixed-grid method is a natural choice for constructing neighbour lists in numerical material modelling. The fixed-grid is simple to construct and can be built in linear time ($O(N)$). The dimensions will be low, being either two or three. Furthermore, the point layout will be mostly uniform, meaning there will be very few or mostly no empty bins. Numerical material modelling also does not require multiple neighbour list construction such as say molecular dynamics. The indexing structure therefore, does not have to be dynamic. The neighbour list only needs to be constructed once.

3.5 Summary

There are three main classes of proximity queries: range search, k-nearest neighbour search, and fixed-radius nearest neighbour searching. The nearest neighbour search is applicable to this study. Specifically the all-pairs fixed-radius nearest neighbours search variation. Our focus is on meshless mechanics methods were all mesh points have pairwise interaction forces if they are within a certain distance δ of each other.

We have seen that to solve proximity searches an indexing structure is used to index the objects in space. This serves to minimize the amount of distance calculations required, thereby greatly reducing the search time.

A large number of indexing structures, with variations and extensions, are available in the literature. In this chapter we covered a few:

- Non-Hierarchical Methods
  - Brute-force or Sequential scan
  - Inverted lists or Projection
  - Space-filling curves
  - Fixed-grid
- Recursive-based Methods
  - K-d Tree
  - Quadtree
  - R-Tree
  - Divide and Conquer
- Miscellaneous Methods
  - Planar point location
  - Delaunay Triangulation

The fixed-grid method is the most promising of all the techniques. It is one of the more simpler data structures to create and builds in linear time ($O(N)$). It is sufficient in handling lower dimensional ($d \leq 3$) search spaces, and point sets in numerical material modelling will be mostly uniformly spaced. To the best of our knowledge nobody has previously attempted to design an algorithm, implementing the fixed-grid method, for massively parallel systems before.

Chapter 4

Fixed-Grid Algorithm for Distributed CPUs

4.1 Introduction

This chapter describes the fixed-grid method of solving the all-pairs fixed-radius nearest neighbours search. It further describes the design of the CPU algorithm, implementing the fixed-grid method, for distributed memory architecture and discusses the scalability results that were obtained.

4.2 The Fixed-Grid Method

The brute force method of solving the fixed-radius neighbour search is to do distance calculations between all of the mesh points in the problem. For N points this results in N^2 distance calculations. The brute force method does not scale well and quickly becomes very time consuming.

The fixed grid method is a well-known method of solving the fixed-radius neighbour search in a much quicker time. The method works by limiting the search space required for each mesh point to find all of its neighbours. This is done by overlaying a grid over the problem domain. The grid divides the domain up into separate regions that are called bins. The grid spacing is equal to the maximum horizon δ_{max} of the problem domain. Thus, a mesh point's neighbourhood can only occupy the point's own bin and adjacent bins. With the brute-force approach the whole problem domain needs to be searched for prospective neighbours. Now the search space of each mesh point is limited to occupied and adjacent bins. Figure 4.1 shows a problem domain that is divided up among 25 bins, with four mesh points inside each bin. The indicated mesh point's neighbourhood does not stretch beyond the adjacent bins of the occupied bin, therefore only the shaded area needs to be searched. This reduces the search space substantially where the fixed-grid method

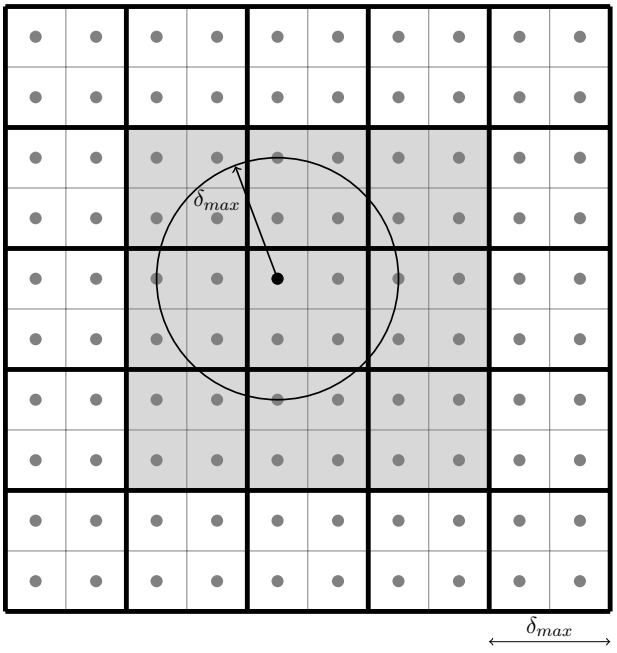


Figure 4.1: The neighbourhood area of mesh points never reach beyond adjacent bins

requires $O(N)$ operations while the brute force method mentioned previously requires $O(N^2)$ operations.

For the fixed-grid method a certain amount of preprocessing is required before the neighbour search can be done. First the fixed-grid parameters must be calculated. Then the unique number of each bin must be calculated. The adjacent bins of every bin must also be calculated and stored. The mesh points must then be stored according to the bins that they occupy.

4.2.1 The Unique Bin Number

The unique number (or ID) for each bin is calculated with

$$b_{id} = (b_z - 1)B_xB_y + (b_y - 1)B_x + b_x \tag{4.2.1}$$

for three dimensions and with

$$b_{id} = (b_y - 1)B_x + b_x \tag{4.2.2}$$

for two dimension. The b_x, b_y and b_z symbols are the bin coordinates along each dimension (counting in bin increments), while B_x and B_y are the total number of bins in the direction of that dimension. The Figure 4.2 shows the square two-dimensional problem domain from Figure 4.1 divided up among 25 bins, five along each dimension. Only the bin numbers are shown. Bin number 14 will be calculated using equation 4.2.2 as $(3 - 1)5 + 4 = 14$. In this way all the bins have a unique number.

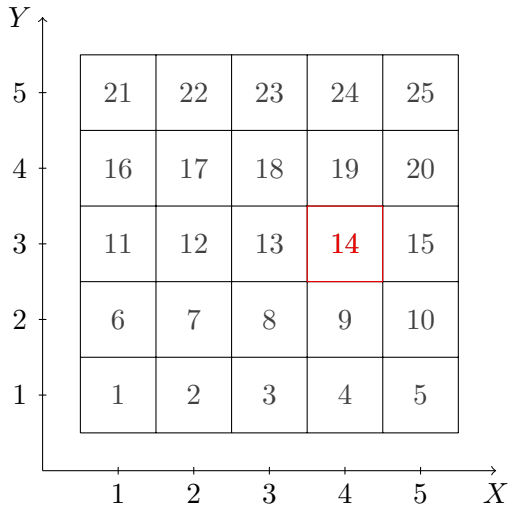


Figure 4.2: A simple two-dimensional problem domain showing only the bins

4.2.2 Adjacent Bin Calculation and Storage

The adjacent bin numbers are calculated using equation 4.2.1 and 4.2.2 again. However now the b_x, b_y, and b_z bin coordinates are incremented from -1 to $+1$ to calculate all the bin numbers adjacent to a specific bin. The bounds of the incremented coordinates are checked to make sure it falls within the domain. For example, the adjacent bin numbers of bin 14 are 8, 9, 10, 13, 15, 18, 19 and 20.

The adjacent bin list is stored using one array. The array holds the adjacent bin numbers for each bin sequentially. In two dimensions each bin has a maximum of eight adjacent neighbours, while in three dimensions each bin has a maximum of 26. Thus, the starting index of a bin's adjacent list stored in the array can easily be obtained by multiplying eight or 26 by one minus the bin number. For example from Figure 4.2 the bin array will start with $\{2, 6, 7, 0, 0, 0, 0, 0, 1, ...\}$, showing the adjacent bin numbers of bin number one. The one indicates the first neighbouring bin number of bin number two at index number eight.

4.2.3 Storage of Mesh Points According to Bins

All the mesh points must be stored according to which bin they occupy, for quick access when the neighbour search is being done. This is done using three arrays. The first array holds the amount of mesh points per bin. Thus, it is known how many mesh points are in each bin. The second array holds the sum off all the previous elements in the first array. The third array is a long list of all the mesh point id's ordered according to the bin number they occupy. Thus, to obtain all the material point id's of a specific bin, one first gets the starting index from the second array and then the amount of points from the first array. The starting index and the amount of points are then used to loop through

all the mesh points of a specific bin in the third array.

With the adjacent bins of every bin known, and the mesh points stored according to the bins, only looping through the relevant portions of all the arrays can quickly do the neighbour search. All the candidate mesh points in the shaded area of Figure 4.1 can quickly be accessed for all the reference mesh points in this way.

4.2.4 Alternative Grid Spacing

An alternative, of the fixed-grid method, is to make the grid spacing smaller than the maximum horizon δ_{max}. A grid spacing smaller than δ_{max} may limit the search space required for each mesh point, depending on the size chosen. It does however require that more than one level of adjacent bins need to be searched along each dimension. Different grid spacing's were not investigated in this study. Here we only focus on parallel scalability. Therefore we chose to make the grid spacing equal to δ_{max}.

4.2.5 Mesh Point Horizon

In peridynamics the horizon of a mesh point is dependant on the size of the material cell represented by the mesh point. The problem domain can thus be discretised into mesh points with different horizons. In this case the δ chosen for the grid spacing will be equal to the maximum horizon of all the mesh points. This will result in the search space (the shaded area in Figure 4.1) being larger relevant to the horizon of the mesh point. Thus, the more mesh point horizons differ in the problem domain, the worse the performance of the fixed grid method will be, given that one picks the grid spacing equal to the maximum horizon size. The influence of a variable horizon size was not tested in this study. Here we focussed on the parallel scalability of the algorithm and kept the horizon sizes constant for all problems. For a study on the ways of optimizing the fixed-grid method, for serial computing, see Noske (2004).

4.3 Memory Organization

The CPU fixed-grid algorithm was designed for parallel-distributed memory computation as shown in Figure 2.2. This implies that each processor has its own local memory connected to the other processor's local memory via a network. The mesh points are divided among the processors. Thus, each processor only has access to its local part of the domain. Any other data, such as mesh point coordinates and horizon sizes, required from other parts of the domain, must be communicated across the network.

Figure 4.3 illustrates that communication between processors will always be required given that we are modelling materials in which the mesh points will always be evenly

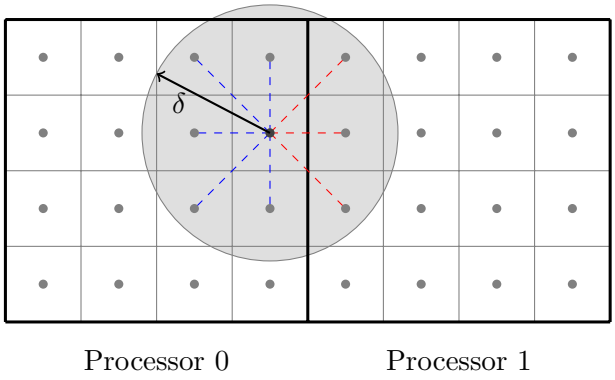


Figure 4.3: Mesh points distributed across processors

distributed. There will always be mesh point bonds overlapping processor boundaries. This necessitates a communication step in the CPU algorithm before the neighbour search is done.

The goal in designing any parallel algorithm is to minimize the required communication as much as possible. Thereby gaining the most performance from a parallel system. Therefore it would be foolish to communicate all the mesh point data to all the processors. This communication would take longer than running the problem on a single processor. Therefore the communication step must have as little communication as possible.

The communication step in the CPU fixed-grid algorithm works as follows: Each processor has its own mesh points and these points occupy bins. The search space required for a mesh point is the occupied bin and adjacent bins as illustrated in Figure 4.1. Thus, all the points in the adjacent bins of the occupied bins of the processor at the boundaries of the processor must be sent across as shown in Figure 4.4. In the Figure all the blue mesh points must be sent from processor 0 to 1, and all the red mesh points must be sent from processor 1 to 0. It is worth noting that the processor boundaries will not necessarily be in line with the grid spacing as is the case in Figure 4.4.

4.4 Hardware and Software

4.4.1 Software

The code written for this thesis made use of Trilinos (www.trilinos.org) software libraries. Trilinos is an object-oriented software framework developed for engineering and scientific applications, at Sandia National Laboratories. It consists of various software packages, each with a specific use in mind. For the CPU fixed-grid algorithm the Teuchos and Tpetra packages were used. Teuchos has an array wrapper class, which was used for all arrays in the algorithm. Tpetra has a set of classes that facilitate distributed memory

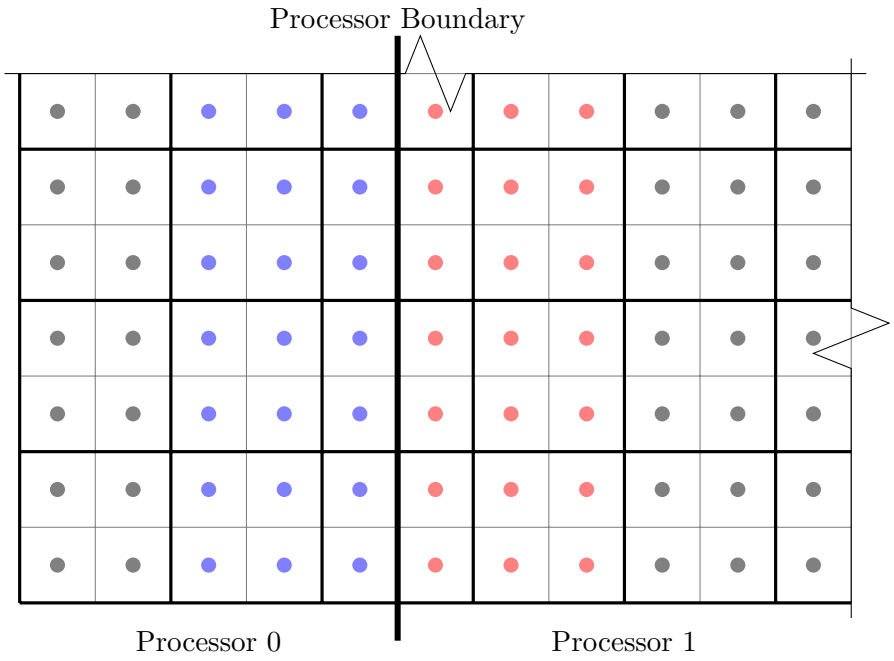


Figure 4.4: The coloured mesh points all need to be communicated from one processor's memory to another and vice versa

communication by utilizing MPI protocols. Among these, the Map, Multivector, and Exporter classes were used. Furthermore the algorithm was written in C++ in an existing peridynamics application.

4.4.2 Hardware

The code was written and tested on an Apple MacBook Air mid 2012 model. It has a 1.8 GHz Intel Core i5 processor and 4 GB of 1600 MHz DDR3 memory. The timing results of the algorithm were then obtained from a system running two Intel Xeon E5-2680 CPUs with eight physical cores each. Each CPU can have a total of 16 logical cores if hyper-threading is enabled. The system has 32 GB of memory. The parallel performance of the algorithm was tested across the cores of the two CPU processors. This represents a best-case scenario of a system with real distributed CPU processors. Therefore, in the results we talk about the amount of cores on which the problems were run. However, when we talk about the algorithm in general we use processors to allude to a large distributed system.

4.5 Algorithm Design

The algorithm has three phases: Preprocessing, communication, and neighbour searching. In the preprocessing step, the input data is obtained and preprocessed into the relevant data arrays that are required for communication and neighbour searching. These data arrays are the *adjacent bin* array (section 4.2.2), the *mesh point* arrays (section 4.2.3),

and the *occupied bin* array. Each processor calculates the adjacent bin list of all the bins covering the domain. The occupied bin array holds all the bin numbers that the local processor has mesh points inside off. The communication step works by looping through the number of processor in the system ($i \leftarrow 0$ to $p-1$). For each loop all the mesh points that need to be communicated to processor p are determined and sent. Lastly the neighbour search is conducted where the candidate neighbour points are only in the occupied and adjacent bins of the reference mesh point. Therefore the search time for each mesh point is significantly reduced. The algorithm is shown in Algorithm 1.

Algorithm 1 The CPU algorithm for distributed memory

INPUT PARAMETERS: n coordinates, n horizon sizes ▷ n is local subset of N
PREPROCESSING PHASE
Calculate local min and max coordinates along each dimension
Process 0 **gathers** local min and max coordinates from all processors
Process 0 calculates global min and max coordinates along each dimension
Process 0 **scatters** global min and max coordinates to all processors
Calculate bin size and fixed-grid parameters
Calculate adjacent bin list
Insert local mesh points into bins
Calculate occupied bin list

COMMUNICATION PHASE
for $i \leftarrow 0$ to $p-1$ **do**
    Processor i **scatters** occupied bin list to all other processors
    if my process $\neq i$ **then**
        Calculate all local mesh points k that are within occupied or adjacent bins to process $i's$ occupied bin list
    end if
    Process i **gathers** all mesh points k
end for
Insert newly received mesh points k into bins

NEIGHBOUR SEARCH
for $i \leftarrow 0$ to n **do** ▷ n is local subset of N
    for $b \leftarrow$ adjacent and occupied bins of i **do**
        for $j \leftarrow$ mesh points in b **do**
            $L_2 \leftarrow$ distance between i and j
            if $L_2 <$ horizon **then**
                Store j as neighbour of i
            end if
        end for
    end for
end for

Table 4.1: Problem Sizes Used in Testing the CPU Algorithm

Problem Name	Mesh points (N)
50K	50,653
100K	103,823
150K	148,877
200K	205,379
250K	250,047
500K	512,000

Table 4.2: CPU Fixed-Grid Algorithm Results

Number of Cores	Average Time (sec)					
	50K	100K	150K	200K	250K	500K
1	0.055	0.116	0.168	0.233	0.280	0.578
2	0.029	0.058	0.088	0.122	0.149	0.309
4	0.018	0.036	0.055	0.072	0.090	0.172
8	0.017	0.031	0.043	0.055	0.067	0.125
16	0.025	0.039	0.050	0.070	0.075	0.128
32	0.068	0.098	0.117	0.159	0.162	0.263

4.6 Performance Testing Details

A set of problems was run to test the algorithm. All the problems were simple cubes with side lengths of 10 with no applied load or boundary conditions. The horizon size δ was set to equal three times the mesh point radius. All the mesh points were the same size. Timing data was only measured over the execution of the algorithm (to compute the neighbour list) and not for other peridynamic calculations. The neighbour list was constructed only once per run, as subsequent searches during the simulation will give similar results. The different problem sizes are shown in Table 4.1. All the problem sizes were run with 1, 2, 4, 8, 16, and 32 cores respectively. Numerous runs were done to get the average for each combination of problem size and system size.

4.7 Results

4.7.1 Timing

The timing results of the CPU fixed-grid algorithm are shown in Table 4.2.

Figure 4.5 shows the run-times from Table 4.2. Each plot is for a different problem size. It can clearly be seen that the run-times are reduced as the number of cores increases up to a point between eight and 16 cores. After that the run-time for the algorithm starts increasing. This is interesting because it shows that the algorithm actually becomes slower as one increases the cores beyond a certain amount. Furthermore, the run-times increase

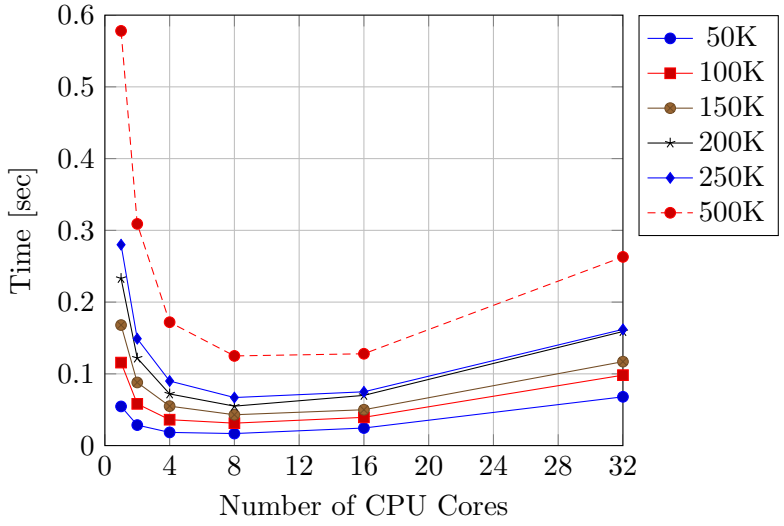


Figure 4.5: Run-times of CPU algorithm for increasing number of cores

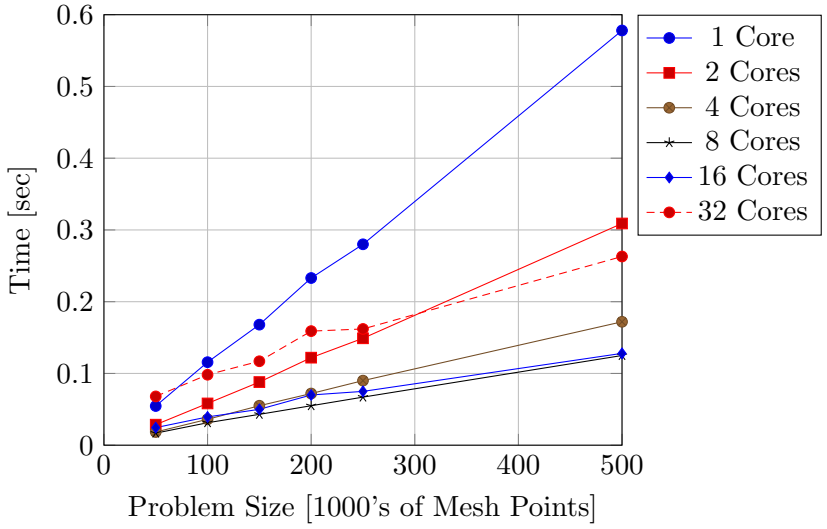


Figure 4.6: Run-times of CPU algorithm for increasing problem size

as the problem size increases, as expected. This Figure shows that there is a sweet spot, between eight and 16 cores, where the run-time is a minimum. Increasing the amount of cores beyond this point results in slower algorithm run-times.

Another representation of Table 4.2 is given in Figure 4.6. Figure 4.6 shows the run-times as the problem size increases for different number of cores. This Figure shows again that the highest number of cores does not necessarily result in the fastest run-time. The fastest run-times are obtained from using eight and 16 cores. The Figure also indicates that the algorithm indeed scales linearly ($O(N)$) as expected.

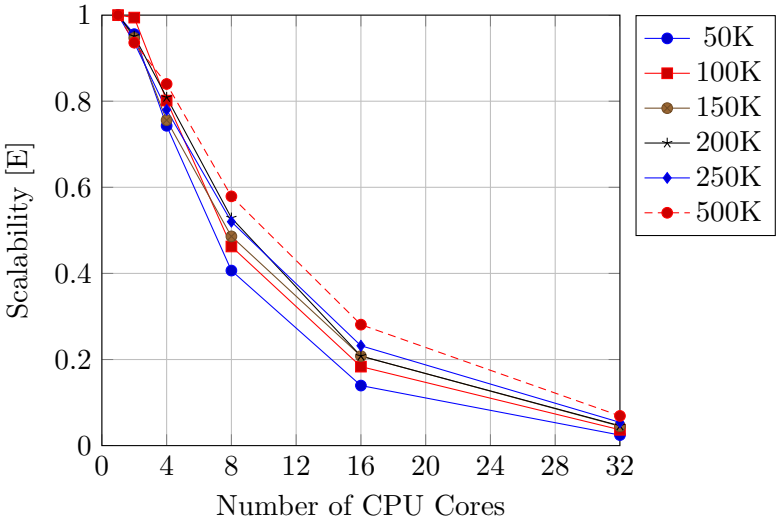


Figure 4.7: Scalability of CPU algorithm

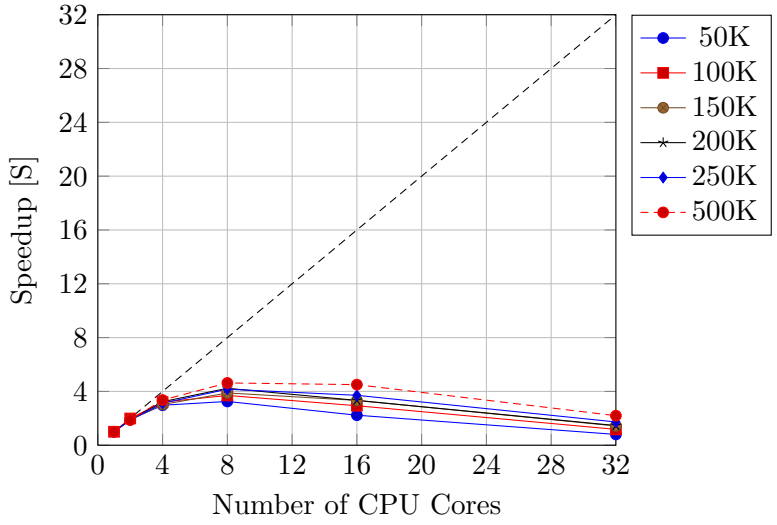


Figure 4.8: Speedup of CPU algorithm

4.7.2 Scalability and Speedup

Figures 4.7 and 4.8 show the scalability and speedup of the algorithm for different problem sizes. These two figures enforce the run-time results of the previous two figures. The scalability reduces drastically as the number of cores increases, showing that the algorithm is not efficient for large parallel systems. The speedup shows the same thing. The solid straight line in Figure 4.8 represents ideal speedup. The maximum amount of performance gain is only four-fold between eight and 16 cores. Thereafter the performance gain reduces as the number of cores increase. Both the scalability and speedup results show that the CPU algorithm's scaling is poor and that it is not at all amenable to a massively parallel-distributed system.

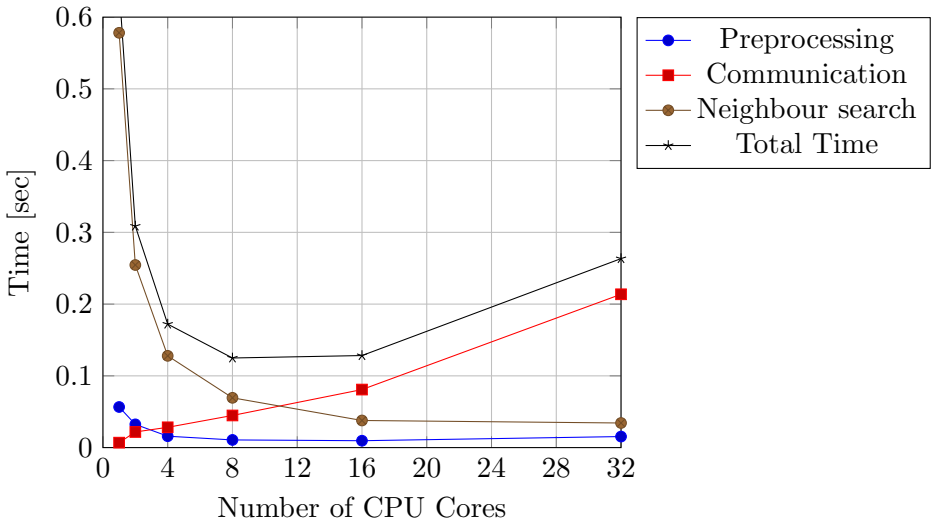


Figure 4.9: Run-times of the three different steps of the CPU algorithm for a 500K problem

4.7.3 Timing of Algorithm Parts

Figures 4.9 and 4.10 reveal the reason why the algorithm's scaling is poor. Figure 4.9 shows the run-times of the three main steps in the algorithm as explained in section 4.5. Figure 4.10 shows the same data but only the percentage of total time for each step. As one starts with one core the nearest neighbour search step dominates. The preprocessing requires very little time and there is no communication. As the number of cores increases the preprocessing and neighbour searching step times both reduce, while the communication step time increases. With 32 cores the communication step takes up 80% of the time. It turns out that the fixed-grid method is so effective and the neighbour search calculations so inexpensive that any communication will trump calculations very quickly. The communication cost relative to the other steps keeps rising as the number of cores increase. In this example the communication cost increases because the number of boundaries across which mesh point data must be communicated becomes more while the amount of mesh points per process becomes less. This result highlights a problem in scaling the fixed-grid algorithm to many distributed processors.

4.7.4 Iso-efficiency Curves

Figure 4.11 shows the iso-efficiency curves of the algorithm. The contour lines show the scalability that was achieved for a certain problem size and number of cores. Vertical lines as shown in the Figure indicate very poor scalability while horizontal lines indicate great scalability. The vertical lines indicate that the problem size must increase infinitely to achieve the same scalability for more cores. This proves that the fixed-grid algorithm is not scalable to large amounts of distributed processors.

The algorithm is however, very inexpensive if communication costs are excluded. If the

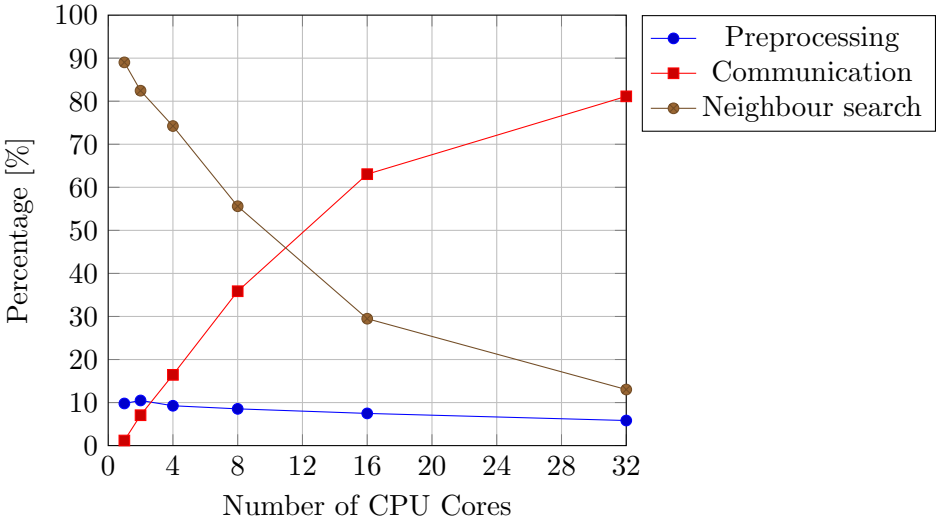


Figure 4.10: Percentage of total run-time of the three different steps of the CPU algorithm for a 500K problem

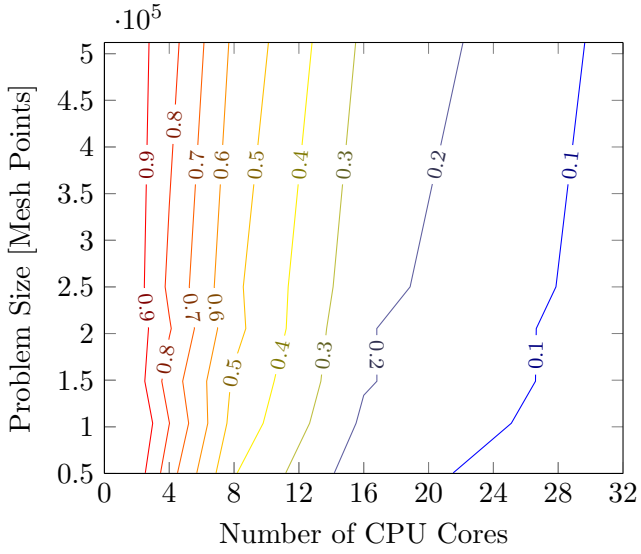


Figure 4.11: Iso-efficiency curves of the CPU algorithm

problem is divided up among distributed processors such that no communication is required the algorithm will be much faster and more scalable.

4.8 Summary

A description of how the fixed-grid method works for solving the all-pairs fixed-radius nearest neighbours search was provided in this chapter. The design of an algorithm was also described, implementing the fixed-grid method, for CPU distributed parallel systems. It was found that the algorithm is not scalable whatsoever. A minimum run-time between eight and 16 cores were reached and increased as more cores were used. It was found that the communication step of the algorithm totally dominates as the number of cores increase.

This is because of the increasing amount of boundaries across which mesh points must be communicated. The algorithm, which reaches a maximum speedup between eight and 16 cores, will be very proficient for desktop type set-ups where the amount of parallelism is low. The algorithm however, will not fully harness next-generation supercomputers that are massively parallel and distributed. Partitioning the problem without any overlapping regions results in too much communication across processors. A method of reducing this communication must be found.

Chapter 5

Fixed-Grid Algorithm for Heterogeneous Systems

5.1 Introduction

In this chapter a neighbour searching algorithm is designed for heterogeneous systems. Focus is placed on partitioning the problem so that there is no communication of the mesh point data required. The algorithm is designed to offload the bulk of the calculations onto a GPU accelerator. The algorithm is tested on a system with one and two GPUs respectively. Due to hardware constraints we could only test on a maximum of two GPUs.

The chapter starts with the algorithm design, and explains the hardware and software used. Thereafter the results of the single GPU are compared with the chapter four results for distributed CPUs. The results of using two GPUs are analysed to determine the scalability of a larger system, given the new problem-partitioning scheme.

5.2 Algorithm Design

The algorithm described for the heterogeneous system is similar to the CPU algorithm of section 4.5. Only now there is no communication required between processors. From chapter 4 we have discovered that the communication step of the fixed-grid algorithm is dominant. Therefore in this algorithm the problem is divided up among the CPU processors in an overlapping manner such that no communication is required between CPUs. The CPUs only initiate the bulk of the calculations on their respective GPUs. Therefore there is only data transferred between the GPU and CPU memory.

The GPU algorithm also has one small improvement compared to the CPU algorithm. Instead of calculating and storing the adjacent bin list, we calculate a stencil array that provides the offset of the bin number of each adjacent bin. The stencil array is at most

27 elements long for three dimensions i.e. the maximum amount of adjacent bins in three dimensions is 26. Thus, giving 27 when the reference bin is included. Instead of looping through the stored adjacent bin numbers, the program loops through the stencil array and increments the reference bin number by the offset stored in the stencil. Using the stencil technique for the CPU algorithm won't improve the CPU times significantly because calculating the adjacent bin list was part of the preprocessing step (section 4.5). From Figure 4.9 it can be seen that the preprocessing step's time was almost negligible. To use the stencil technique one requires that none of the occupied bins are on the edge of the domain. Therefore, one would extend the amount of bins by two in each dimension to ensure that empty bins surround the domain.

The algorithm consists of two kernels (also called functors), namely a binning functor and a building functor. The binning functor inserts the mesh points into the bins, while the building functor builds the actual neighbour list. For the GPU algorithm one must distinguish between what happens on the CPU and what kernels are offloaded to the GPU. Furthermore, one must also keep track of where data is stored i.e. on the device memory or host memory.

There are six arrays used in the algorithm. *Coordinates* is a two-dimensional array storing the mesh point coordinates. The *stencil* array holds the offsets to calculate the adjacent bin numbers of a reference bin. The *bins* array is a two-dimensional array that stores the mesh point id's according to the bin the points occupy. The *bincount* array keeps count of how many mesh points are inserted into each bin. Lastly there is the *neighbours* and *numneigh* arrays. *Neighbours*, a two-dimensional array, holds the mesh point id of every neighbour for each point (the neighbour list), and *numneigh* keeps count of the amount of neighbours for each mesh point. The algorithm is shown in Algorithm 2.

5.3 Hardware and Software

5.3.1 Software

The code written for the GPU algorithm again made use of Trilinos (www.trilinos.org) software libraries. For the GPU algorithm the Kokkos package was used. Kokkos implements a shared memory parallel programming model and facilitates the memory management and kernel invocation on the GPU. It provides an abstract envelope (layer) for the Nvidia CUDA (Compute Unified Device Architecture) programming model, facilitating the portable use of CUDA. Furthermore the algorithm was written in C++.

Algorithm 2 The GPU algorithm

```
1: procedure HOST ALGORITHM(coordinates, horizon)
2:     Calculate bin size and fixed-grid parameters
3:     Calculate stencil array
4:     Copy stencil and coordinates array to device memory
5:     Allocate bincount, bins, neighbours and numneigh arrays on device memory
6:     Binning Functor          ▷ A map operation of each mesh point x in coordinates
7:     Building Functor         ▷ A map operation of each mesh point x in coordinates
8: end procedure

9: procedure BINNING FUNCTOR(x)
10:    b_id ← bin number of x
11:    Add x to bins array at index of b_id
12:    Increment bincount at index of b_id
13: end procedure

14: procedure BUILDING FUNCTOR(x)
15:    b_id ← bin number of x
16:    for b ← adjacent bins of b_id do          ▷ Adjacent bins calculated using stencil
17:        for y ← mesh point id's in b do
18:            Get coordinates of y from coordinates
19:            L_2 ← distance between x and y
20:            if L_2 < horizon then
21:                Store y in neighbours at index of x
22:                Increment numneigh at index of x
23:            end if
24:        end for
25:    end for
26: end procedure
```

5.3.2 Hardware

The timing results of the algorithm was then obtained from a system with two Nvidia Tesla K40 GPUs. Each K40 processor has 2880 CUDA cores and can deliver a peak double precision floating point performance of 1.43 Tflops.

5.4 Measuring CPU vs. GPU Performance

Nvidia's CUDA parallel computing platform poses a problem as to measuring the scalability or speedup of the GPU algorithm as was done previously in chapter four. Because CUDA decides the amount of threads or streams that the data is divided into, there is no way of specifying the amount of cores that is being utilised on the GPU processor. Previously, for the CPU, we used different amounts of cores to measure the scalability of the algorithm. We only have access to two GPU processors, which severely limits the scalability that can be measured. Therefore relative throughput was used to compare the

CPU and GPU results. Throughput is defined in section 2.4.4. Relative throughput is defined as

$$\text{Relative Throughput} = \frac{\text{GPU Throughput}}{\text{CPU Throughput}} \tag{5.4.1}$$

Relative throughput measures the ratio of GPU throughput over CPU throughput. As throughput is a good measure of how much work is done per a given time, relative throughput serves as an indication of how much performance is gained.

5.5 Performance Testing Details

The algorithm performance was tested with two different configurations. First, a single CPU-GPU pair was used. These results are discussed in section 5.6.1. Second, two CPU-GPU pairs were used. These results are discussed in section 5.6.2. In the first case, the whole problem resides on the single CPU memory. The coordinates array transferred to GPU memory for calculations contain the coordinates of all the mesh points in the problem. In the second case, the problem is split in half between the two CPUs. In chapter four however, we have shown that a distributed mesh leads to an unscalable algorithm. If the CPUs require communication before the calculations are offloaded to their GPUs it would still result in an unscalable algorithm. Therefore, the problem was split in half *with overlapping regions* so that no communication is required between the distributed CPU memories beforehand. The coordinates array in the second case has half the mesh points plus some overlapping region. Therefore, what was tested represents a heterogeneous architecture with shared memory between the CPUs.

A set of problems was run to test the proposed algorithm. All the problems were simple cubes with side lengths of 10 with no applied load or boundary conditions. The horizon size was set to equal three times the mesh point radius. Furthermore, all the mesh points were the same size. The problem sizes tested are shown in Table 5.1. Numerous test runs were done to get the average time of the algorithm for each problem size.

5.6 Results

Table 5.2 shows the run-time and throughput results obtained from using one and two GPUs respectively.

5.6.1 One GPU Compared with One CPU

In the first case, we compare one GPU with the results from chapter four. Figure 5.1 shows the relative throughput of one GPU from Table 5.2 compared to the CPU throughput

Table 5.1: Problem Sizes Used in Testing the GPU Algorithm

Problem Name	Mesh Points (N)
125K	125,000
1,000K	1,000,000
3,375K	3,375,000
8,000K	8,000,000
15,625K	15,625,000
27,000K	27,000,000
42,875K	42,875,000
64,000K	64,000,000

Table 5.2: GPU Fixed-Grid Algorithm Results

Problem	One GPU		Two GPUs	
Name	Time (sec)	Throughput ($10^6 \frac{Points}{second}$)	Time (sec)	Throughput ($10^6 \frac{Points}{second}$)
125K	0.004	32.5	0.003	40.8
1,000K	0.024	42.4	0.013	75.0
3,375K	0.103	32.8	0.052	65.4
8,000K	0.248	32.3	0.121	66.1
15,625K	0.501	31.2	0.256	61.1
27,000K	0.859	31.4	0.443	60.9
42,875K	1.403	30.6	0.726	59.0
64,000K	2.105	30.4	1.063	60.2

of chapter four's results. One can see that the GPU outperforms the CPUs significantly. Important to note is that a single GPU delivers between 30 and 36 times the throughput when compared to a single CPU. This shows that the GPU is faster regardless of whether communication is required or not. The GPU, overall, provides from six to 40 times the performance, depending on how many CPU cores were used. The results show that a GPU does give better throughput than a CPU. This is however, to be expected as GPUs are high throughput devices. From Table 5.2 and Figure 5.4 one can see the throughput delivered by the single GPU is constant at 30 million points per second.

In Figure 5.2 the GPU times from Table 5.2 are plotted against the CPU times from Figure 4.6. One can see that the GPU times are much faster. Important to note is the single GPU time compared to the single CPU time. This shows again that a single GPU is faster than a single CPU regardless of whether communication is required or not. These results show that the algorithm is well suited to take advantage of the parallelism offered by a GPU processor and that significant performance gains can be obtained in such a way.

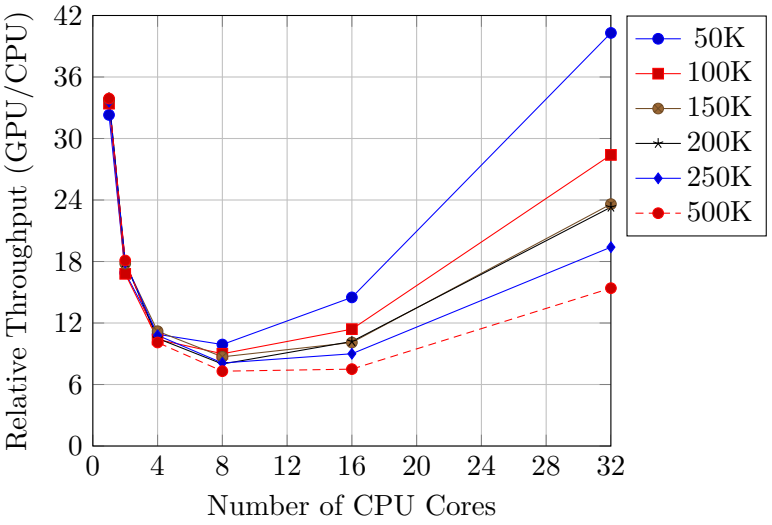


Figure 5.1: Relative throughput of the CPU algorithm from chapter four vs. a single CPU-GPU pair. The GPUs throughput was taken as a constant 30 million

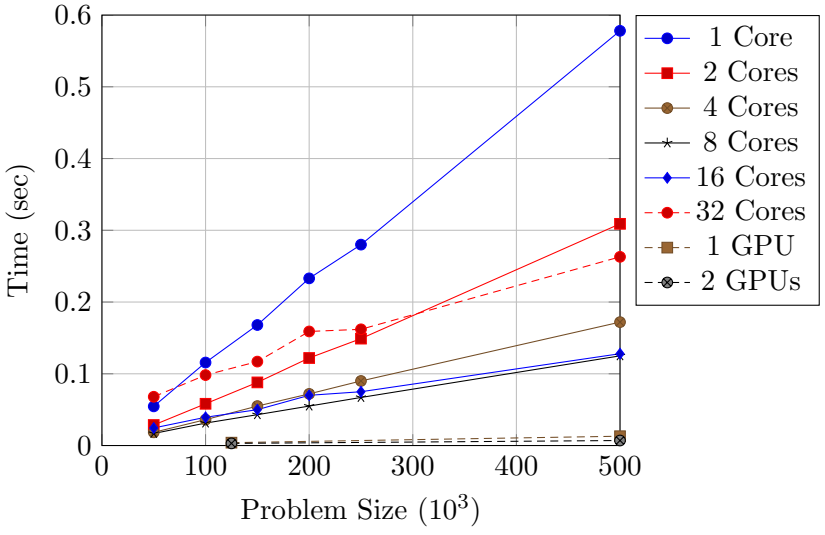


Figure 5.2: The run-times of the CPU algorithm from chapter four with the new GPU timing results from Table 5.2. The times shown for the GPUs at 500K was obtained by interpolating

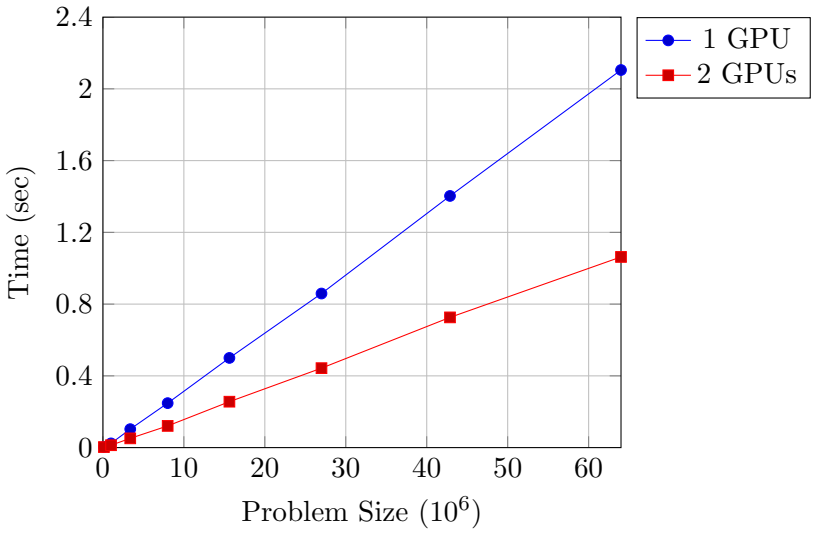


Figure 5.3: The single and double CPU-GPU pair's run-time from Table 5.2

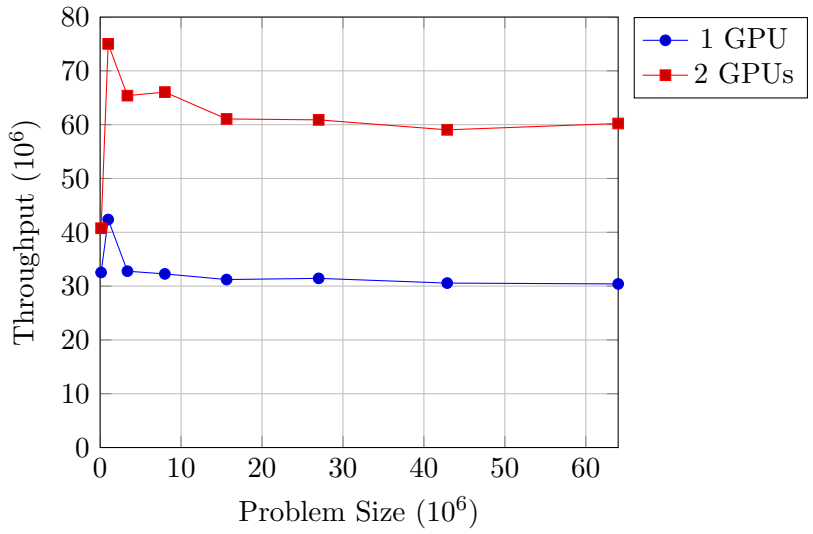


Figure 5.4: The single and double CPU-GPU pair's throughput from Table 5.2

5.6.2 Scalability of Two GPUs

In the second case we compare the performance of two GPUs with one. Figure 5.3 and 5.4 show the run-times and throughput of the GPU algorithm for one and two GPUs respectively. Here it can be seen that going from one to two GPUs perfectly doubles the obtained throughput and halves the run-time. This is a result of major importance which indicate that a very large problem can be mapped to multiple GPUs without losing much efficiency of the system. This is provided that the problem is partitioned in an overlapping manner, such that the CPUs controlling the GPUs do not have to communicate mesh point data beforehand.

5.7 Summary

In this chapter the design of a GPU neighbour searching algorithm was explained. The algorithm performance was measured on one and two GPUs. In the case of one GPU it was found that a single GPU significantly outperforms a single CPU. It was found therefore, that the algorithm could take full advantage of the parallelism of a GPU processor. In the case of two GPUs, the problem was partitioned with an overlap so that no communication prior to neighbour searching is required. It was found that two GPUs delivered twice as much throughput compared to one, showing that the proposed algorithm may be scalable to massively parallel systems with multiple GPUs.

Chapter 6

Conclusion

6.1 Summary of Results

This thesis set out to design an algorithm for solving the all-pairs fixed-radius nearest neighbours search that is scalable to massively parallel systems. The all-pairs search problem is stated as follows: Given a set of N points in d-dimensional space, find all pairs of points within a distance δ of one another. This search is required by any nonlocal solid mechanics method to construct the neighbour list of each mesh point in the problem domain prior to the relevant mechanics calculations. Furthermore, GPU computing has become very popular in the last decade. Most of the fastest supercomputers in the world today employ GPU processors as accelerators to CPU processors. It is also believed that the next-generation exascale supercomputers will be heterogeneous. Therefore the focus was to design an algorithm that takes advantage of the parallelism offered by GPU processors. Two algorithms, implementing the fixed-grid method, were developed. The first was for a distributed CPU system. The second was for a heterogeneous architecture consisting of CPUs and GPUs.

For the first algorithm, from chapter four, the problem domain is partitioned among multiple CPU cores in a non-overlapping manner. Therefore the algorithm follows a SPMD programming model. The following results were obtained:

- The algorithm reaches an optimum run-time between eight and 16 cores. With an increasing number of cores the run-time increases as well.
- The scalability, speedup and iso-efficiency plots confirm that the algorithm is not scalable and quickly becomes inefficient as the number of cores increase.
- Figures 4.9 and 4.10 show that the communication cost becomes dominant as the number of cores increase.

From the results it is clear that the CPU algorithm is not scalable to massively parallel systems. This is because the calculations of the fixed-grid method are very light and any communication required will quickly trump such calculations. The solution therefore is to avoid communication of mesh point data across processor boundaries. Dividing the problem domain up among the processors in an overlapping manner can do this.

For the second algorithm from chapter five, the problem domain is partitioned in an overlapping manner such that no communication is required between processors beforehand. This represents a shared memory system. The algorithm was tested on one and two GPU processors respectively. The single GPU performance was compared to the single CPU performance to investigate the performance gain obtainable with no communication required. The performance of two GPUs were compared with that of one, to determine how scalable a multi GPU system might be. The following results were obtained:

- The single GPU outperformed the single CPU by a factor of between 30 and 36.
- A single GPU also outperforms multiple CPU cores as the relative throughput was always at least greater than eight as shown in Figure 5.1.
- Two GPUs delivered exactly twice the throughput of one GPU.

Two conclusions can be drawn from the results. Firstly, the fixed-grid algorithm is amenable to exploit the parallelism offered by a GPU processor. Therefore the algorithm delivers much more performance on a GPU processor compared to a CPU processor. On a single GPU the algorithm will outperform any CPU system size, running the algorithm from chapter four, as shown by Figure 5.1. Secondly, if the problem domain is partitioned in such a way that no communication is required it becomes very scalable. Two GPUs delivered twice the performance of one GPU.

In conclusion, the developed algorithm can take advantage of the parallelism offered by GPU processors. Furthermore, the calculations required are very light weight, therefore communication must be avoided. Dividing the problem up into regions that overlap can do this. It is the ideal to be able to run computer simulations without any limit on the size of the problem. To harness the power of next generation computers, problems will be very large and will not fit on the memory of one processor. Therefore it is important that the algorithm is scalable on a distributed system. An algorithm was designed that will be scalable and fast on massive heterogeneous systems, facilitating extreme scale simulations.

6.2 Proposed Future Work

Two additional projects are proposed: Firstly, the design of a CPU algorithm that partitions the problem in an overlapping manner so that no communication is required before

the neighbour search is done. It is believed that without communication costs the fixed-grid CPU algorithm will perform much better. The algorithm is amenable to GPU processors, but will also be very scalable on a distributed CPU system if the communication costs are nullified. Secondly, due to hardware constraints we were not able to test our GPU algorithm with more than two GPU processors. Another project would be to test the algorithm on a system comprising many GPUs to confirm that it is indeed scalable.

List of References

Al-Furaih, I., Aluru, S., Goil, S. and Ranka, S. (2000). Parallel Construction of Multidimensional Binary Search Trees. *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, pp. 136–148. ISSN 10459219.

Allahdadi, F.A., Carney, T.C., Hipp, J.R., Libersky, L.D. and Petschek, A.G. (1993). High Strain Lagrangian Hydrodynamics: A Three Dimensional SPH Code for Dynamic Material Response. Tech. Rep., DTIC Document.

Andreica, M. and Tapus, N. (2012). Sequential and MapReduce-based Algorithms for Constructing an In-Place Multidimensional Quad-Tree Index for Answering Fixed-Radius Nearest Neighbor Queries. *Acta Universitatis Apulensis*, , no. 29, pp. 131–151.

Aref, W. and Barbara, D. (1995). Efficient Processing of Proximity Queries for Large Databases. *Data Engineering*, pp. 147–154.

Artemova, S., Grudinin, S. and Redon, S. (2011 October). A Comparison of Neighbor Search Algorithms for Large Rigid Molecules. *Journal of Computational Chemistry*, vol. 32, no. 13, pp. 2865–77. ISSN 1096-987X.

Atluri, S.N. and Zhu, T. (1998). A New Meshless Local Petrov-Galerkin (MLPG) Approach in Computational Mechanics. *Computational Mechanics*, vol. 22, no. 2, pp. 117–127.

Barenblatt, G.I. (1962). The Mathematical Theory of Equilibrium Cracks in Brittle Fracture. *Advances in Applied Mechanics*, vol. 7, no. 1, pp. 55–129.

Beckmann, N., Kriegel, H.-P., Schneider, R. and Seeger, B. (1990). *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*, vol. 19. ACM.

Belytschko, T., Lu, Y.Y. and Gu, L. (1994). Element-free Galerkin Methods. *International Journal for Numerical Methods in Engineering*, vol. 37, no. 2, pp. 229–256.

Bentley, J. (1990). K-d Trees for Semidynamic Point Sets. *Proceedings of the sixth annual symposium on Computational Geometry*.

Bentley, J. and Shamos, M. (1976). Divide and Conquer in Multidimensional Space. *Proceedings of the eighth annual ACM Symposium on Theory of Computing*, pp. 220–230.

Bentley, J., Stanat, D. and Jr, E.W. (1977). The Complexity of Finding Fixed-Radius Near Neighbors. *Information processing letters*, vol. 6, no. December.

Bentley, J.L. (1975 September). Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, vol. 18, no. 9, pp. 509–517. ISSN 00010782.

Bentley, J.L. and Friedman, J.H. (1979 December). Data Structures for Range Searching. *ACM Computing Surveys*, vol. 11, no. 4, pp. 397–409. ISSN 03600300.

Bentley, J.L. and Ottmann, T.A. (1979). Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 643–647.

Bustos, B., Deussen, O., Hiller, S. and Keim, D. (2006). *A Graphics Hardware Accelerated Algorithm for Nearest Neighbor Search*. Springer.

Castelli, V. (2004). Multidimensional Indexing Structures for Content-based Retrieval. *Image Databases: Search and Retrieval of Digital Imagery*, vol. 22208, no. 98723.

Chávez, E. and Navarro, G. (2001). Searching in Metric Spaces. *ACM Computing Surveys*, vol. 33, no. 3, pp. 273–321.

Chazelle, B. (1983). An Improved Algorithm for the Fixed-Radius Neighbour Problem. *Information Processing Letters*, vol. 16, no. May, pp. 193–198.

Dickerson, M.T. and Drysdale, R.S. (1990). Fixed-radius Near Neighbors Search Algorithms for Points and Segments. *Information Processing Letters*, vol. 35, pp. 269–273. ISSN 00200190.

Duarte, C.A. and Oden, J.T. (1996). Hp Clouds - An Hp Meshless Method. *Numerical Methods for Partial Differential Equations*, vol. 12, no. 6, pp. 673–706.

Dugdale, D.S. (1960). Yielding of Steel Sheets Containing Slits. *Journal of the Mechanics and Physics of Solids*, vol. 8, no. 2, pp. 100–104.

Edelsbrunner, H. (1980). A Time and Space-Optimal Solution for the Planar All Intersecting Rectangles Problem. *Technical University of Graz, Institut f{ü}r Informationsverarbeitung, Report F*, vol. 50.

Eringen, A.C., Speziale, C.G. and Kim, B.S. (1977). Crack-tip Problem in Non-local Elasticity. *Journal of the Mechanics and Physics of Solids*, vol. 25, no. 5, pp. 339–355.

Fatahalian, K., Sugerman, J. and Hanrahan, P. (2004). Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 133–137. ACM.

Flynn, M.J. and Rudd, K.W. (1996). Parallel Architectures. *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 67–70.

Galoppo, N., Govindaraju, N.K., Henson, M. and Manocha, D. (2005). LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In: *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 3. IEEE Computer Society.

Gingold, R.A. and Monaghan, J.J. (1977). Smoothed Particle Hydrodynamics: Theory and Application to Non-Spherical Stars. *Monthly Notices of the Royal Astronomical Society*, vol. 181, no. 3, pp. 375–389.

Govindaraju, N.K., Henson, M., Lin, M.C. and Manocha, D. (2005). Interactive Visibility Ordering and Transparency Computations Among Geometric Primitives in Complex Environments. In: *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 49–56. ACM.

Griffith, A.A. (1921). The Phenomena of Rupture and Flow in Solids. *Philosophical Transactions of the Royal Society of London*, pp. 163–198.

Guttman, A. (1984). R-Trees - A Dynamic Index Structure for Spatial Searching. *Proceedings of the ACM*.

Hillerborg, A., Modéer, M. and Petersson, P.-E. (1976). Analysis of Crack Formation and Crack Growth in Concrete by Means of Fracture Mechanics and Finite Elements. *Cement and Concrete Research*, vol. 6, no. 6, pp. 773–781.

Hinrichs, K. and Nievergelt, J. (1983). The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects. Tech. Rep., DTIC Document.

Kanda, T. and Sugihara, K. (2005). Two-dimensional Range Search Based on the Voronoi Diagram. *International Journal of Computational Geometry and Applications*.

Klein, P.A., Foulk, J.W., Chen, E.P., Wimmer, S.A. and Gao, H.J. (2001). Physics-based Modeling of Brittle Fracture: Cohesive Formulations and the Application of Meshfree Methods. *Theoretical and Applied Fracture Mechanics*, vol. 37, no. 1, pp. 99–166.

Knuth, D.E. (1973). Sorting and Searching (The Art of Computer Programming volume 3).

Krüger, J. and Westermann, R. (2003). Linear Algebra Operators for GPU Implementation of Numerical Algorithms. In: *ACM Transactions on Graphics (TOG)*, vol. 22, pp. 908–916. ACM.

Liszka, T.J., Duarte, C.A.M. and Tworzydlo, W.W. (1996). < i> hp</i>-Meshless Cloud Method. *Computer Methods in Applied Mechanics and Engineering*, vol. 139, no. 1, pp. 263–288.

Liu, W.K., Jun, S. and Zhang, Y.F. (1995). Reproducing Kernel Particle Methods. *International journal for Numerical Methods in Fluids*, vol. 20, no. 8-9, pp. 1081–1106.

Löhner, R., Sacco, C., Onate, E. and Idelsohn, S. (2002). A Finite Point Method for Compressible Flow. *International Journal for Numerical Methods in Engineering*, vol. 53, no. 8, pp. 1765–1779.

Lucy, L.B. (1977). A Numerical Approach to the Testing of the Fission Hypothesis. *The Astronomical Journal*, vol. 82, pp. 1013–1024.

McCreight, E.M. (1980). Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles. Tech. Rep..

Melenk, J.M. and Babuška, I. (1996). The Partition of Unity Finite Element Method: Basic Theory and Applications. *Computer Methods in Applied Mechanics and Engineering*, vol. 139, no. 1, pp. 289–314.

Morton, G.M. (1966). *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company.

Nguyen, V.P., Rabczuk, T., Bordas, S. and Duflot, M. (2008 December). Meshless Methods: A Review and Computer Implementation Aspects. *Mathematics and Computers in Simulation*, vol. 79, no. 3, pp. 763–813. ISSN 03784754.

Nievergelt, J., Hinterberger, H. and Sevcik, K.C. (1984). The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 1, pp. 38–71.

Noske, A. (2004). *Efficient Algorithms for Molecular Dynamics Simulations and Other Dynamic Spatial Join Queries*. Ph.D. thesis, James Cook University.

Oñate, E. and Idelsohn, S. (1998). A Mesh-free Finite Point Method for Advective-Diffusive Transport and Fluid Flow Problems. *Computational Mechanics*, vol. 21, no. 4-5, pp. 283–292.

Onate, E., Idelsohn, S., Zienkiewicz, O.C. and Taylor, R.L. (1996). A Finite Point Method in Computational Mechanics. Applications to Convective Transport and Fluid Flow. *International Journal for Numerical Methods in Engineering*, vol. 39, no. 22, pp. 3839–3866.

Preparata, F. (1981). A New Approach to Planar Point Location. *SIAM Journal on Computing*, vol. 10, no. 3, pp. 473–483.

Sagan, H. (1994). *Space-Filling Curves*, vol. 18. Springer-Verlag New York.

Samet, H. (1984). The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260.

Sellis, T., Roussopoulos, N. and Faloutsos, C. (1987). The R+-tree: A Dynamic Index for Multi-Dimensional Objects.

Shamos, M. and Hoey, D. (1975). Closest Point Problems. *16th Annual IEEE Symposium on Foundations of Computer Science, 1975*, pp. 151–162.

Sharifzadeh, M. and Shahabi, C. (2010). VoR-Tree : R-trees with Voronoi Diagrams for Efficient Processing of Spatial Nearest Neighbor Queries. *Proceedings of the VLDB Endowment*, pp. 1231–1242.

Shen, S.N.A..S. (2002). The Meshless Local Petrov-Galerkin (MLPG) Method: A Simple & Less-Costly Alternative to the Finite Element and Boundary Element Methods. *Computer Modeling in Engineering & Sciences*, vol. 3, pp. 11–51.

Silling, S. (2000 January). Reformulation of Elasticity Theory for Discontinuities and Long-Range Forces. *Journal of the Mechanics and Physics of Solids*, vol. 48, no. 1, pp. 175–209. ISSN 00225096.

Tamminen, M. (1982). The Extendible Cell Method for Closest Point Problems. *BIT Numerical Mathematics*, vol. 22, no. 1, pp. 27–41.

Turau, V. (1991). Fixed-Radius Near Neighbors Search. *Information Processing Letters*, vol. 39, no. August, pp. 201–203.

Xu, X.-P. and Needleman, A. (1994). Numerical Simulations of Fast Crack Growth in Brittle Solids. *Journal of the Mechanics and Physics of Solids*, vol. 42, no. 9, pp. 1397–1434.

Zhou, K., Hou, Q., Wang, R. and Guo, B. (2008). Real-time KD-tree Construction on Graphics Hardware. In: *ACM Transactions on Graphics (TOG)*, vol. 27, p. 126. ACM.