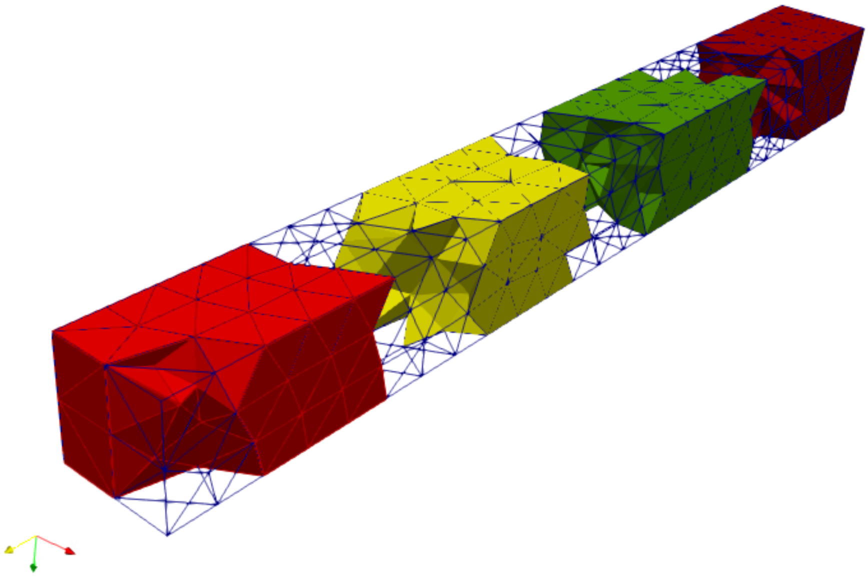
High Performance Data Traversal

Cache Aware Computing with Space Filling Curve

by

Sagar Dolas

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday August 22, 2017 at 10:00 am.



Student number:	4593065	
Project duration:	November 1, 2016 – August 22, 2017	
Thesis committee:	Prof. dr. ir. K. Vuik,	Supervisor, TU Delft
	Prof. dr. ir. H. X. Lin	Professor, Mathematical Physics, TU Delft
	Dr. Matthias Möller,	Supervisor, TU Delft
	Dr. Vahid Galavi,	Supervisor, Deltares

Preface

Firstly, I would like to thank my thesis supervisor Dr Matthias Möller for his guidance, support, patience and advice. I would like to express my gratitude towards him for providing me the freedom to work on my concepts and help me channelise my ideas into appropriate direction. I also want to thank Dr Vahid Galavi for his guidance and the numerous fruitful discussions. Without the support of above two people, this thesis would not have been possible. I would like to thank Prof. dr. Kees Vuik for providing me the COSSE room, organising couple of COSSE workshops and all the necessary discussions. I would like to thank TU Delft and Deltares for providing me the resources to complete by master's thesis. Lastly, I would also like to extend my regards to everyone I met over the past two years in Europe.

Special acknowledgement go to the European Commission and the COSSE Consortium for providing me opportunities to travel and partially fund my master's studies in Germany and The Netherlands. Without their kind support, I did not stand a chance to leave my country and pursue higher education in Europe. Above all, I would like to express my sincere gratitude to my family in India and friends Ayesha and Sweta for their kind love and persistent belief. The moral support and love that I received from them made this journey comfortable, motivating and worthwhile.

Sagar Dolas
Delft, August 2017

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Research Focus . . . 2

1.3 Research Objectives . . . 3

1.4 Thought Process . . . 4

1.5 Thesis Outline . . . 4

2 Modern CPU Architectures **7**

2.1 Introduction . . . 7

2.2 Single Processor Architecture . . . 7

2.2.1 Modern Processor . . . 7

2.2.2 Memory Management . . . 9

2.3 Multi-core Socket Architecture . . . 13

2.3.1 Challenges for Performance on Multi-core Machines . . . 15

2.3.2 NUMA Architecture . . . 15

2.3.3 Performance Issues on NUMA Architecture . . . 15

3 Analyzing Performance Impact of Data Access on NUMA machine **17**

3.1 Introduction . . . 17

3.2 Test System Specifications . . . 17

3.3 Performance Analysis . . . 19

3.3.1 Softwares and External Libraries Used . . . 19

3.3.2 Modified Stream Benchmark . . . 19

3.3.3 Analyzing Memory Bound Kernel on ccNUMA machine . . . 20

3.3.4 Analyzing Compute Bound Kernel on ccNUMA machine . . . 22

3.3.5 Analyzing Effect of Indirect Random Access on Memory Bandwidth Limited Kernel . . . 23

3.3.6 Analyzing Effect of Indirect Random Access on Compute Bound Kernel . . . 25

3.4 Conclusion . . . 29

4 An Introduction to Space Filling Curves **31**

4.1 Introduction . . . 31

4.1.1 Mathematical Description . . . 31

4.1.2 Space Filling Curves in Use . . . 32

4.1.3 Space Filling Curve Construction for Arbitrary Mesh . . . 33

4.2 Development of Parallel C++ Code for Space Filling Curves . . . 38

4.2.1 Parallel Performance Analysis of ParSFC application . . . 38

4.2.2 Relative Error . . . 40

4.3 Visualizing Data traversal with Space Filling Curves . . . 41

4.3.1 Test Case and Details . . . 41

4.3.2 Analysis of Serial Data Traversal . . . 42

4.3.3 Analysis of Parallel Data Traversal . . . 45

5 Analysis with Space Filling Curve on Finite Element Solver **49**

5.1 Introduction . . . 49

5.1.1 The Governing Equation . . . 50

5.1.2 Review on various solvers with Space Filling Curve . . . 52

5.2 Introduction to Finite Element Solver . . . 53
5.3 Performance Analysis and Methodology . . . 56
5.4 Interpreting Analysis . . . 56
5.4.1 Interpreting Analysis on Matrix Assembly . . . 57
5.4.2 Interpreting Impact on Inverse Power Iteration . . . 57
5.5 Matrix Structure Analysis. . . . 61
5.5.1 Impact on Matrix Bandwidth . . . 61
5.5.2 Performance Impact on LU Factorization . . . 62
5.6 Conclusion . . . 65

6 An Introduction to the Material Point Method **67**
6.1 Introduction . . . 67
6.2 Governing Equations . . . 67
6.2.1 Reynold's Transport Theorm . . . 67
6.2.2 Conservation of Mass . . . 68
6.2.3 Conservation of Linear Momentum . . . 69
6.2.4 Boundary Conditions . . . 69
6.2.5 Weak Formulation . . . 70
6.3 Material Point Method Formulation . . . 70
6.3.1 Material Point Method Discretization . . . 70
6.3.2 Lagrangian Phase and Eulerian Phase . . . 71

7 Challenges for Achieving High Performance for the Material Point Method **73**
7.1 Introduction . . . 73
7.1.1 Understanding Memory Access Pattern in Particle to Grid Interaction. . . . 73
7.1.2 Tackling Particle to Grid Interaction Efficiently in MPM Simulation . . . 74
7.1.3 The Hidden Potential Challenge . . . 75
7.2 Focus of Study . . . 78

8 Performance Analysis of Explicit MPM 3D Simulation **79**
8.1 Introduction . . . 79
8.1.1 Explicit Time Integration scheme and Stability Criteria . . . 79
8.1.2 Explicit MPM Algorithm. . . . 80
8.1.3 Introduction to the Explicit MPM Solver . . . 82
8.2 Performance Analysis on Dam Collapse Simulation . . . 83
8.2.1 Test Case Geometry, Boundary Condition and Simulation Parameters . . . 83
8.2.2 Methodology . . . 84
8.2.3 Visualization . . . 85
8.2.4 Results and Discussion . . . 89
8.3 Conclusion . . . 90

9 Analysis of Implicit MPM 3D Simulation **91**
9.1 Introduction . . . 91
9.1.1 Newmark-Beta Implicit Time Integration Scheme . . . 91
9.1.2 Newton's Method to Solve NonLinear System . . . 91
9.1.3 Focus of Study in Implicit MPM Algorithm. . . . 93
9.1.4 Implicit MPM Solver . . . 95
9.2 Performance Analysis on Dam Collapse Simulation . . . 96
9.2.1 Test Case Geometry, Boundary Condition and Simulation Parameters . . . 96
9.2.2 Methodology . . . 97
9.2.3 Results and Discussions . . . 102
9.3 Conclusion . . . 103

10 Summary, Conclusions and Future Work **105**
10.1 Summary . . . 105
10.2 Conclusions. . . . 106
10.3 Future Work. . . . 107

Bibliography **109**

1 Introduction

"What Mathematics is to Physics, Data traversal is to High-performance computing"

1.1. Motivation

The world of Computational science has witnessed an exponential expansion of complex numerical algorithms in the last few decades mainly to understand minute details and solve complex physical problems. It has established itself as the third pillar of science after theory and experimentation and has been successful in gaining immense popularity as a mainstream research work among academicians and scientists working in entirely different fields. The Computational Sciences has brought together Mathematicians and Computer Scientists to work in close collaboration on the variety of interdisciplinary research problems.

The need for developing large scale numerical and scientific codes for the simulation real world problems has taken the steep curve and so the current hardware industry. The Top500 [1] list ranks world's most powerful supercomputer capable of delivering 125 PFlop/s. To effectively understand this mammoth computational power, imagine 7 billion people on this planet continuously performing an add operation for ≈450000 years to match computational intensity which Sunway TaihuLight [2] shown in Figure 1.1 delivers in one second. Still, it is impossible to fully simulate the functional behaviour of the human brain or accurately predict decades of climate change. The pivotal point here is, expensive hardware or massive computational infrastructure does not naturally invoke high-performance computing but implementation of hardware auxiliary mathematical ideas, cache efficient data traversal strategies, sensible use of parallel programming paradigms and energy aware management of computational resources on machines ranging from very grass-root level basic NUMA system to entire million core server stack does.

As the world's fastest supercomputer today stands at the horizon of PetaScale computing and looks ahead for Exascale floating point performance as clearly projected in Figure [1] 1.2, there has been a tremendous rush in the last decade to develop high performance and scalable scientific software mainly to exploit the enormous computational intensity of supercomputers judiciously. The foremost challenge to achieve high performance for computational researchers in near about every front is to optimise developed serial codes and achieve parallel scalability on thousands and millions of processing cores. The central theme to the above-mentioned problem is data traversal, data placement and memory access pattern which largely influences floating point performance and energy efficiency.

As highlighted earlier, an expensive machine is not an assurance for good performance. Careful investigation of underlying hardware and understanding ways to extract maximal

[1] https://www.top500.org

[2] http://www.nsccwx.cn

performance are indispensable keys to achieving better computational throughput. Advance development of the numerical method and its analysis definitely ensures a numerically accurate solution to underlying mathematical physics, but competent software implementation and development is absolutely essential to solve real world problems. Deep understanding of elemental micro-architecture, effort to grasp behaviour of multi-core multi socket high-end server machines, reducing communication overhead and reducing energy consumption will play an important role in designing and implementing scalable numerical algorithms.

In this master's thesis we will first focus on investigating the impact of data traversal patterns on the performance of several micro-benchmarks on NUMA machine. In the second part we will implement an advanced data traversal scheme designed to improve cache utilisation for two numerical methods and analyse performance impact.

INNOVATION COOPERATION SHARING EXCELLENCE
神威
太湖之光
THE SUNWAY TAIHULIGHT SYSTEM IS THE WORLD' S FIRST SUPERCOMPUTER WITH PEAK PERFORMANCE OVER 100PFLOPS.
THE ENTIRE COMPUTING SYSTEM IS BASED ON THE SW26010 MANY-CORE PROCESSOR.

Figure 1.1: World's fastest supercomputer

1.2. Research Focus

The Data traversal is the soul of high-performance computing. Indeed it is the backbone, the way data travels to the CPU from main memory largely influences the performance of particular kernel on specific machine architecture. The majority of modern machines are designed to deliver high performance if data traversal can utilize maximum bandwidth to main memory (DRAM) and make efficient use of hierarchical memory structure. Thus, a hardware optimal data access pattern should be designed to take advantage of the underlying hardware to scale and achieve performance and that forms the central theme of this work.

In this master's thesis, the focus will be on implementing strategy for efficient data arrangement and data reuse. This thesis work will try to explore challenges of achieving high performance in advance numerical methods by delving deep into their required memory access pattern.

CPUs processing power and memory access speed to DRAM have taken different curves over past few decades and it appears to widen in near future. In today's HPC scenario, most of the scientific kernels are poorly addressing their memory requirements and therefore there is dire need to include mechanisms which help to mollify it and reduce the load on memory controllers. In this thesis, there is an attempt to implement a recipe for improved data access pattern and explore it on matrix assembly and linear system solver for Finite Element code,

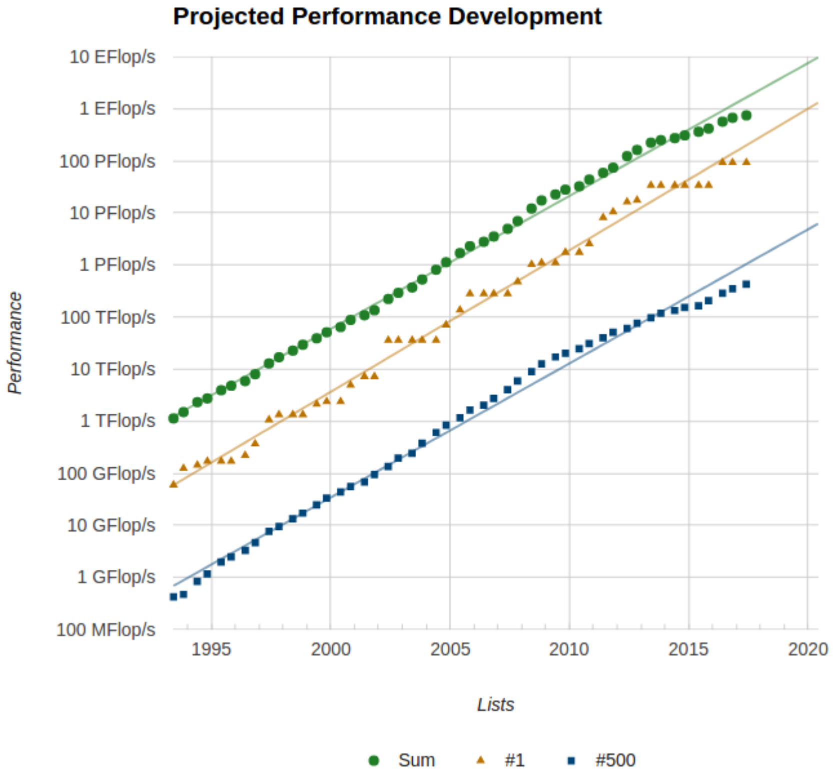


Figure 1.2: Projected performance development of world's 500 fastest supercomputers

and overall performance analysis of two different Material Point Method codes.

1.3. Research Objectives

This section highlights the research direction of the Master's thesis and precisely layouts the research objectives discretely.

Objectives

1. To understand the theoretical background of high-performance computing on modern CPU architectures and it's evolution in time.
2. To understand general performance bottlenecks of computational kernels in terms of limited hardware resources and programming paradigms.
3. To investigate and compare performance impact of data traversal schemes on simple modified stream benchmarks with respect to floating point performance and memory bandwidth.
4. To understand the mathematics behind numerical techniques such as finite element method, material point method and explore challenges in these methods to achieve high floating point performance.
5. To understand and choose advanced data traversal algorithms which may act as a unifying approach towards superior domain decomposition, generating cache oblivious data layout for NUMA machines, and comes packed with minimal computational overhead.
6. To investigate and compare performance improvements in scientific codes working with above mentioned numerical methods with the inclusion of new data traversal scheme.

1.4. Thought Process

This master's thesis starts with the theoretical understanding of high-performance computing by closely examining the changes in modern micro-architecture and multi-core multi-socket server machines. The first set of the investigation aims at understanding the available memory bandwidth and computational power of a typical dual socket server machine. Data traversal and memory access pattern are very basic yet important aspects of any scientific kernel and play an important role in achieving good performance specifically on modern machines where fetching data from DRAM can be up to ≈2-3x more expensive compared to on-chip cache and up to ≈5x from other NUMA nodes. A modified stream benchmark for NUMA machine was implemented as is used to study importance of data traversal and data placement on bandwidth utilization and computational power.

The FEM is widely used approach for approximate solution of PDE problems. As the name suggests, Finite element codes mainly involve global matrix assembly and solution of system of equations both for linear and nonlinear problems either with direct or iterative solvers. The elemental wise traversal of the underlying grid for sparse matrix assembly typically involves indirect random access to vertices and elements which invoke lots of cache misses, redundant bus cycles and therefore constrains it to utilize maximum available bandwidth offered by the machine. The HPC machines are converging towards more cores/socket, larger memory bandwidth to DRAM and increased bandwidth between CPUs and therefore the main aim of this work is to understand the evolution of modern machines and evolve scientific kernels towards it to achieve high performance. The technique which brings together temporal and spatial locality, generating cache oblivious data layout and makes use of larger bandwidth offered by the machine should be inculcated.

Going a step further, this thesis also explores the Material Point Method which combines the FEM with discrete particles. The particles move freely within the underlying grid in each computational cycle. Each computational step involves matrix assembly similar to finite elements and this unique interaction of particles with grid naturally invokes a lot of unstructured random access which inhibits this type of particular numerical scheme to achieve high performance even on bandwidth oriented machine. There exists, a special class of data traversal, mathematical curves called Space-filling curves which are originally fractal curves. One of the most important properties of Space Filling Curves is their ability to generate data layouts which can be distributed across multiple processors in distributed memory environments with minimal communication and also across multiple cores in shared memory environment with very low synchronization points.

1.5. Thesis Outline

This section provides a brief introduction to the content of each chapter and guides the reader through the rest of this document.

Chapter 2 **Modern CPU architecture** highlights relevant bottlenecks in performance of single-core and multi-core CPUs and NUMA machine. It discusses critical components of modern processors and highlights the important functionality of memory module. It also describes performance related issues in NUMA architectures and reviews some key solution-strategies to improve computational throughput.

Chapter 3 **Analysing Performance Impact of Data Access on NUMA Machine** takes a deep dive into performance aspects of data access patterns on compute-bound and memory-bound kernels with respect to operational memory bandwidth utilisation. It also studies the impact of data traversal patterns on energy consumption and summarizes key strategies to be adopted while programming on NUMA machines for high bandwidth utilisation.

Chapter 4 **An Introduction to Space Filling Curves** introduces Space Filling Curves

(SFC) and presents them as a versatile and efficient mesh reordering scheme. It provides mathematical basis to the idea of locality and explains SFC generation for arbitrary two or three-dimensional grid in detail. It provides a brief overview of the **ParSFC** application used for generating SFC and presents the three-dimensional visualizations of data traversal on arbitrary Finite Element grid to properly grasp and understand the potential impact of data access patterns on computational performance.

Chapter 5 **Analysis of Space Filling Curve on Finite Element Solver** introduces an eigenvalue problem of beam propagation in optical waveguide governed by Helmholtz equation. It establishes Finite Element discretization and simultaneous system of linear equation to solve for lowest eigenvalue and associated eigenvector. This chapter contains deep performance analysis of matrix assembly, sparse matrix-vector multiplication on the grounds of CPU time, energy efficiency, cache utilization and effective use of memory module. This analysis helps to redraw performance expectation of Space Filling Curve as a data traversal algorithm. This chapter also analyses matrix structure with respect to different industrial strength reordering schemes and compares their performance impact on LU factorization.

Chapter 6 **The Material Point Method** introduces the numerical method, the Material Point Method (MPM). It explains the mathematical physics behind MPM and outlines both the Lagrangian step and the Eulerian step.

Chapter 7 **Challenges for Achieving High Performance for the Material Point Method** explores the source of bad memory access pattern in the particle-grid interaction. It explains the importance of coalesced memory access pattern for improving cache utilisation and how grid reordering can help to achieve it. It establishes the focus of study for following chapters and unleashes hidden computational challenge especially in the case of the Material Point Method, where the set of active elements changes frequently dynamically changing initial particle-grid interaction.

Chapter 8 **Analysis on Explicit MPM 3D Solver** first explains explicit MPM algorithm in detail and then describes a test case for experimentation. It also highlights the important features of Anura3D code developed by Deltares which is used here for analysis. The main focus of this chapter is to understand the effect of reordering of the elements and vertices of the background grid on the overall computational performance of the MPM code.

Chapter 9 **Analysis on Implicit MPM 3D Solver** first explains implicit MPM algorithm in detail and then describes a test case for experimentation. It explores the open source MPM code Kratos-Particle-Mechanics and describes its important components. The main focus of this chapter is to grasp performance impact of mesh reordering of the background grid on implicit MPM solver.

2

Modern CPU Architectures

2.1. Introduction

This chapter contains explanations of modern single and multi-core CPU architectures, discusses trends and optimisation strategies to be employed for achieving high performance on these kinds of machines. The most important point for high performance computing is applying techniques to reduce wastage of memory bandwidth and this chapter will pay plenty of attention to it. Some of the essential ideas and facts in this chapter have been taken from the book ”Introduction to High-Performance Computing” by Victor Eijkhout [6]. The reader should refer to chapter 1-5 for more detailed explanations and ideas.

2.2. Single Processor Architecture

This section focuses on minute details of single-core architecture and in particular on the movement of data between main memory and the processor and within different levels of memory in the processor. Understanding memory access and its productive usage is essential for efficient scientific computing since data movement from main memory to the processing core is an order of magnitude slower than processor's computing power in today's modern CPUs.

Highly efficient codes require an understanding of microprocessor architectures and therefore it is important and sometimes crucial to know the architecture before implementing the raw algorithm. For many problems of academic and industrial interest, the use of the parallel computer is a necessity, but before that, a proper understanding of single core and its capabilities is a must. In modern scientific computing, one of the biggest challenges is to provide data hungry processor with data efficiently, and this chapter will pay attention to that.

2.2.1. Modern Processor

Modern processors are quite complicated and this section will give very concise introduction to important and pertinent parts of modern processor. The *Von Neumann* [1] architecture models sequential instruction handling. Modern processor supports *out of order* instruction handling that is, instructions can be handled differently than specified by the program only when reordering the instruction leaves the results of the execution unchanged. More detailed understanding of instruction handling can be found in Intel Architectures Optimization Reference Manual [2]. Some of the important features and trends in instruction handling are summarised as follows :=

[1] https://en.wikipedia.org/wiki/Von_Neumann_architecture

[2] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual pdf

1. **Multiple issue** : In Modern CPUs, instructions that are independent of each other can be executed in parallel.

2. **Branch prediction and speculative execution** : Today's compilers can guess whether a conditional instruction will evaluate to true and act accordingly. The purpose of branch predictor is to improve instruction pipeline, and they play a very critical role in achieving high performance for today's x86 based microprocessors. The branch predictor attempts to avoid waste of time by guessing whether conditional jump will be taken or not. The branch that is guessed will be speculatively executed. Otherwise, instructions are disregarded and pipeline starts over again with the correct branch. For example, to improve branch prediction, Intel's Sandy Bridge microprocessor was redesigned to have Branch Target Buffer (BTB) double memory than its predecessor Nehalem micro-architecture to store more data to guess next steps, therefore, allowing CPU to load more instructions beforehand leading to improved CPU performance.

3. **Out of order execution**: Instructions can be reordered if they are not dependent on each other. The processor executes instructions in an order governed by availability of input data. In this way the processor can avoid being idle while waiting for the preceding instruction to complete to retrieve data for the next instruction in a program, instead processes the next instructions that are able to run immediately and independently. Consider the in-ordered instruction for following assembly code :

   (a) ld r1, r2 (load from r1 from memory into r2)

   (b) add r2, r1, r3 ($r1 = r2 + r3$)

   (c) add r4, r5, r3 ($r4 = r5 + r3$)

   Suppose, r3 and r5 are available but r2 has to be brought in from Level 2 cache which takes around 20 clock cycles. Until then pipeline is stalled, however second instruction is ready and can be executed. The out-of-instruction handling executes second instruction before first in order to avoid idle time.

4. **Prefetching** : Data prefetching fetches request data before it is needed to take advantage of fast caches. Instruction prefetching attempts to load instructions before they are executed to improve performance. Microprocessors based on Intel's Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell, and Broadwell support four types of hardware prefetchers for prefetching data. There are two prefetchers associated each with Level 1 and Level 2 cache which can prefetch up to 128 bytes of cache lines.

Fused Multiply Add

In scientific computing, floating point computation holds top most priority, and for this reason, cores have dedicated execution units for treating arithmetic operations. For example, Intel's Sandy Bridge micro-architecture features 15 execution units as compared to 12 execution units in its predecessor Nehalem micro-architecture. The arithmetic operations differ in a number of clock cycles required to execute them which determines computational performance.

For example, in modern CPU, a division operation can take up to 10 or 20 clock cycles limiting the performance of a numerical algorithm while multiple addition or multiplication execution units can asymptotically calculate the same result per cycle. The phenomenon of executing instruction $x \leftarrow ax + b$ in the same amount of time as separate addition or multiplication is called *Fused Multiply Add* also called as FMA. FMA can achieve an asymptotic speed of several floating point operations per clock cycles so therefore effort should be made to convert expensive operation like the division into equivalent multiplication or addition operation.

Instruction Pipelining

Figure 2.1 shows a typical sequence of serial pipelined arithmetic operations consisting of IF,ID,EX, MEM and WB stages. The computational throughput is 1 complete independent operation per 5 clocks cycles.

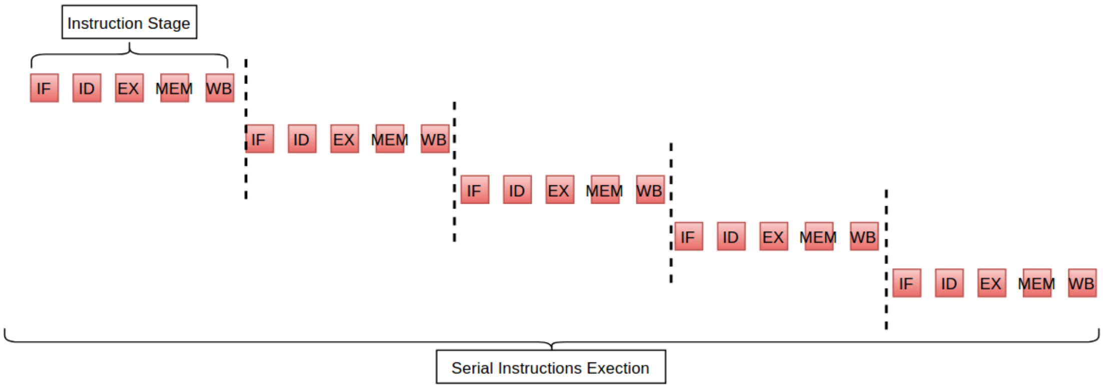


Figure 2.1: Schematic illustrations of serial pipelined operations, IF- Instruction Fetch, ID - Instruction Decode, EX- Execution, MEM- Memory Access, WB - Write back

The modern processor features instruction level parallelism schematically shown in Figure 2.2. The asymptotic computational throughput is one independent operation per clock cycle which is a speed up of 4-6 compared to non-pipelined CPU. The first Intel's Pentium processor had five pipeline stages which were increased to 31 in Pentium-4 processor and again decreased to 14 in Core i3, i5, i7 whereas, the new Intel's Nehalem and Sandy Bridge microprocessor configuration features 19-25 pipelining stages. The crucial point here is, higher pipeline staged does not guarantee higher performance because entire pipeline may have to be flushed repeatedly imposing adverse effects.

2.2.2. Memory Management

This section will mainly discuss essential ideas about memory management especially from hardware perspective and about how data travels from main memory to CPU core. This section will also point out challenges in achieving high performance specially on bandwidth limited kernels and techniques to overcome them.

Bus Structure

The bus structure is mainly responsible for moving data around the computer especially from main memory (DRAM) and CPU. The most important part is *Front Side Bus (FSB)* which connects processor to the main memory through memory controller hub as shown in Figure 2.3. The northbridge or fast connect typically handles communication among CPU sockets, from cores to main memory (DRAM) , cores to video cards. However, Intel's Sandy Bridge processor configuration introduces full integration of northbridge functionality onto CPU chip decreasing overall size as shown in Figure 2.7.

Bus speed is typically much lower than the CPU frequency and therefore the rate at which the CPU can process data is much faster than the rate which data can be delivered to it. The introduction of fast and fat caches has alternatively solved this problem to some extent but has invoked other challenges which will be discussed later. Bus speed can not be increased indefinitely, but the number of bus channels can be increased ultimately increasing the volume of data to CPU.

For example, Intel's 5600 Nehalem series processor configuration featured support for maximum 3 memory channels, but on the other hand its successor Intel's 2600 Sandy Bridge series processor configuration features support for 4 memory channels increasing maximum

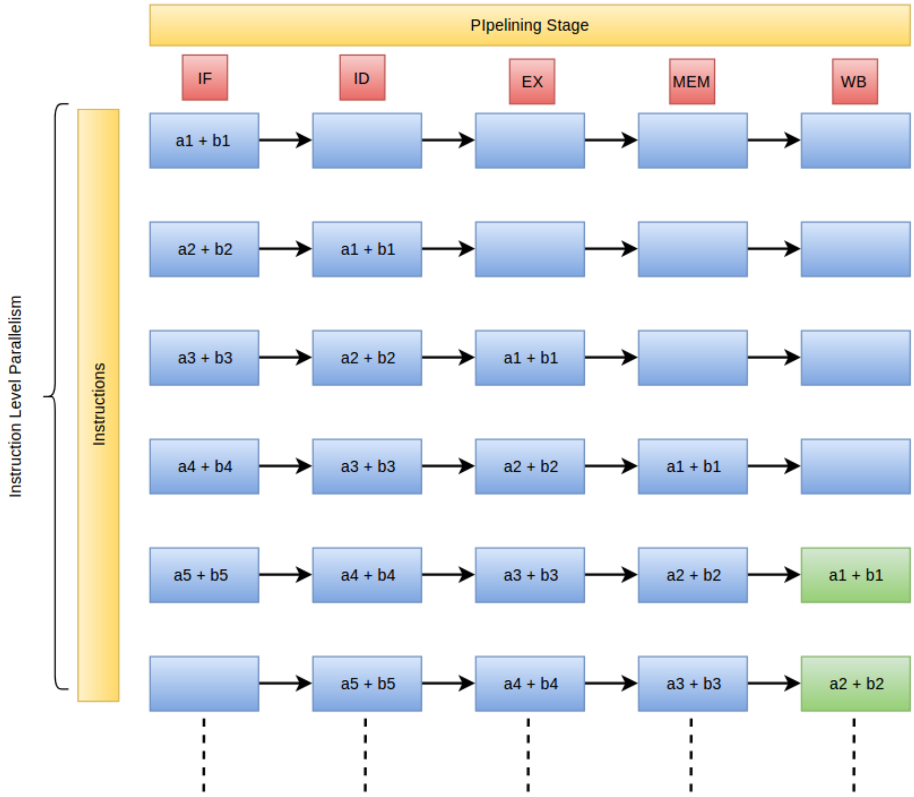


Figure 2.2: Schematic illustrations of parallel pipelined operations, IF- Instruction Fetch, ID - Instruction Decode, EX- Execution, MEM- Memory Access, WB - Write back

operational memory bandwidth by 33 %. This example suggests that trend in processor configuration is moving towards more operational bandwidth and therefore trying to solve the problem of low memory speed by supplying more data. However, increasing the number of memory channels will not help to achieve high performance if data access patterns does not adhere to the underlying hardware layout. In the next sections we will investigate this problem from hardware and programmer's perspective.

Caches

Caches are low-latency high-bandwidth memory where data can reside for a moderate amount of time. Data from the main memory travels through the caches up to the registers and CPU core. The principal advantage of cache memory is data reusability. If data is reused shortly after it was first needed, it will still be in the cache, and therefore can be accessed much faster than if it would have to be brought in again from the main memory.

Figure 2.4 shows schematic layout of the cache hierarchy for Intel's Sandy Bridge processor configuration. Loading data from register is so fast that, it virtually involves no latency at all and therefore does not contribute to any limitations whatsoever. The different level of caches are called *Level 1* (L1) and *Level 2* (L2) and nowadays modern processors also have *Level 3* (L3) cache. The L1 and L2 cache are private and on chip memory whereas in most recent processor configurations L3 is off-chip. The L1 cache is small, typically around 16Kbytes to 32Kbytes but it is much faster and sustains a bandwidth of 32 bytes per cycle. The L2 cache is bigger around hundreds of Kbytes and roughly 8 times bigger than L1 cache and has 3 times more latency as compared to L1 cache. L3 is typically around Mbytes approximately 10 times bigger than L2 cache and sustains 2x latency shared among multiple cores.

Data needed in some operations gets copied into various levels of caches up to the main

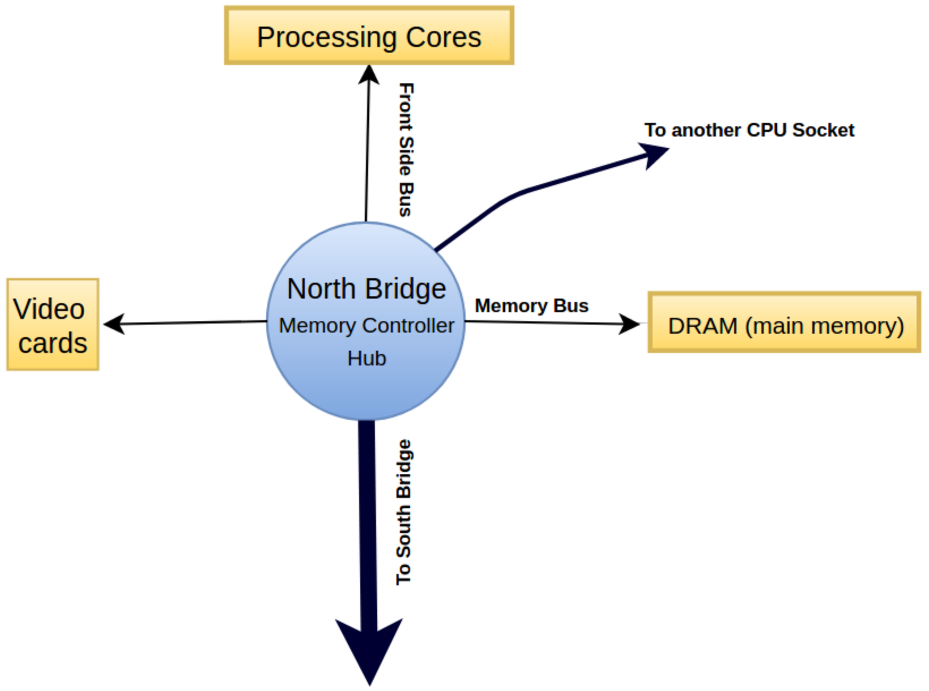


Figure 2.3: Schematic diagram of memory-bus structure

processor, if some instructions later, a data item is needed again, it is first searched in L1 cache, if it is not found there it is searched in L2 cache and if it also not found there it is searched in L3 cache or the main memory. Main memory access has a latency of more than 200 cycles and bandwidth of 4.5 Bytes per cycle, which is about 1/7th of the L1 bandwidth. However this is again shared by multiple cores of a processor chip effectively reducing the operational bandwidth. There are three types of cache misses as summarized below.

1. **Compulsory miss** : This occurs during first time reference to data, and is unavoidable.

2. **Capacity miss** : This type of cache misses is due to the size of the working set and is caused by data having been overwritten because the cache simply cannot contain all data. This type of cache miss can be avoided by partitioning data into chunks that are small enough to easily stay in cache for sufficient amount of time providing spatial and temporal locality upon repetitive memory access.

3. **Conflict miss** : This type of cache miss occurs when one data gets mapped to the same cache address as another data, while both are still needed for computation.

The typical time required to retrieve data if not found in cache is as follows :

1. L1 Cache : 1-2 Clock cycles

2. L2 Cache : 5-20 Clock Cycles

3. L3 Cache : 50 - 100 Clock cycles

4. Main memory : 300 - 500 Clock Cycles

In this master's thesis, the focus is on avoiding capacity cache misses by travelling or accessing data in an efficient way.

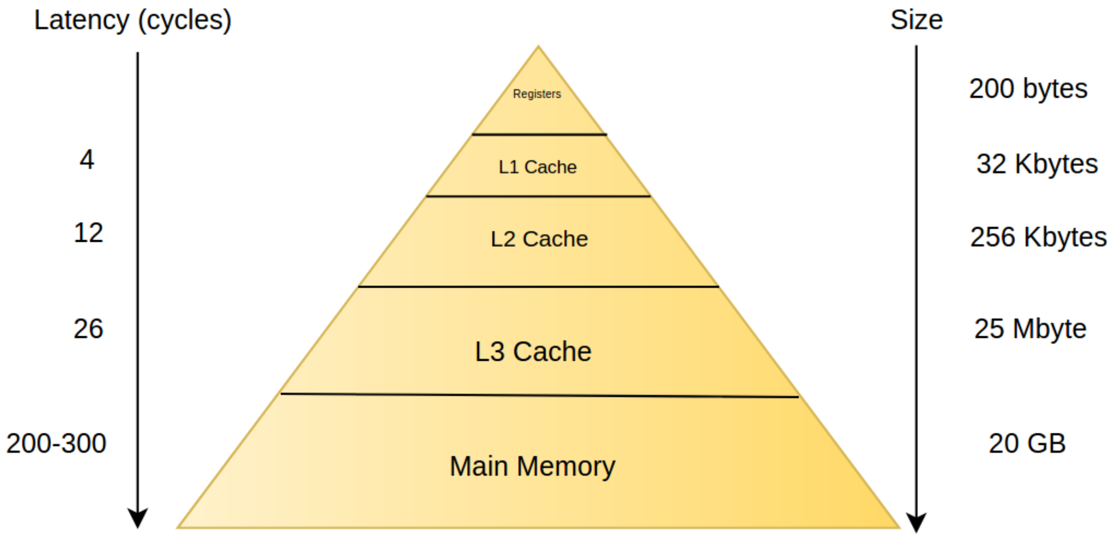


Figure 2.4: Schematic diagram of cache hierarchy

Data Movement between Main Memory and Processor

Data movement between memory and cache, or between caches is not done in single bytes, or even words. Instead, the smallest unit of data moves is called a cache line, sometimes referred as cache block. A typical cache line size can be 32 or 64 or 128 bytes which in context of scientific computing implies 4 or 8 or 16 double-precision floating-point numbers. It is important to understand the importance of cache line, since any memory access costs the transfer of several words. An efficient program then tries to use the other items in the cache line, since access to them is effectively free. This phenomenon is visible in codes that access arrays by unit stride. This is also one of the key-points of the next chapter. Typical description of cache line is illustrated in Figure 2.5.

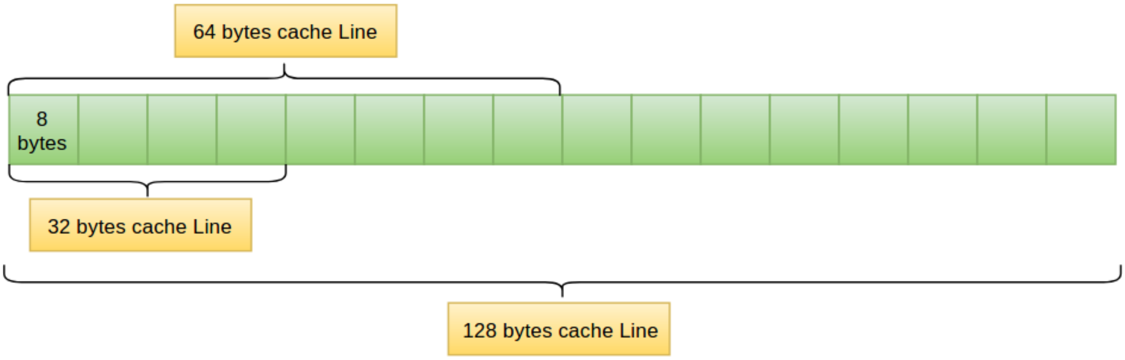


Figure 2.5: Schematic illustration of cache line

2.3. Multi-core Socket Architecture

Over the decade, multicore architectures have surfaced and now dominate most of the commodity hardware industry. The main reasons for this development are as follows :=

1. Clock frequency can not be increased further due to two main reasons, firstly : Energy consumption have increased drastically leading to dissipation of more heat generation and secondly : To increase computational power, more transistors have to be assembled in small amount of chip area, but transistors gate have reached their fundamental limits of few atomic layers restricting to further narrow down size thus ultimately restraining the number of transistors on chip.

2. It is not possible to extract more Instruction Level Parallelism (ILP) from codes due to compiler limitations.

One of the ways to further increase the computational power is to move from traditional single-core architecture to multi-core chips because two cores of lower frequency can have the same asymptotic computational throughput as a single core at a higher frequency: hence reduction in energy consumption. A typical multicore chip is shown in Figure 2.6 [3]

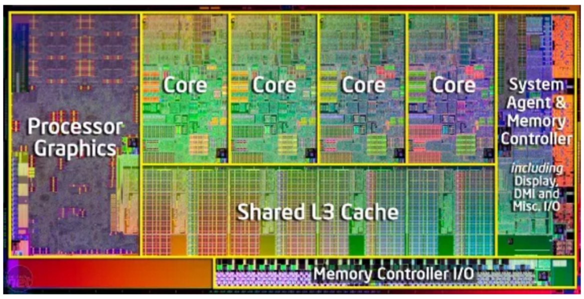


Figure 2.6: A typical multi-core chip

In this section, a brief introduction to Intel's Xeon E5-2600 multi-core processor series is presented along with relevant minute details to understand a classic multi-core multi-socket processor configuration. Figure 2.7 shows a typical Intel Xeon E5 2600 series processor configuration having 8 processing cores. Each processing core has private L1 and L2 cache and shared L3 cache with each core its share of L3 bucket represented in grey colour in Figure 2.7. Cores are connected to each other via bidirectional ring bus architecture which provides high scalability and low latency bandwidth to L3 cache.

These processors support up to 4 memory channels per CPU socket and up to 3 DIMM slots per channel with a maximum of 32 GB memory per DIMM slot cumulating to 768 GB of RAM at most. All memory channel supports up to maximum memory speed of 1600 MHz which amounts to the maximum bandwidth of ≈102.4 GB/s. The Quick Path Interconnect (QPI) offers 8 GHz/link high bandwidth connection between CPUs as shown in Figure 2.8. In today's scenario, the focus of hardware development has shifted towards enlarged bandwidth-oriented architectures especially to mitigate the gap between memory speed and processor's computing power. On one hand, multi-core machines are moving towards increased memory bandwidth and more cores/ CPU-sockets so, on the other hand, complicated programming challenges have emerged making it difficult to efficiently utilise these machines. In the next section, there will be a brief discussion about some of the important challenges to consider while programming for high performance.

[3] https://www.cnet.com/news/what-became-of-multi-core-programming-problems/

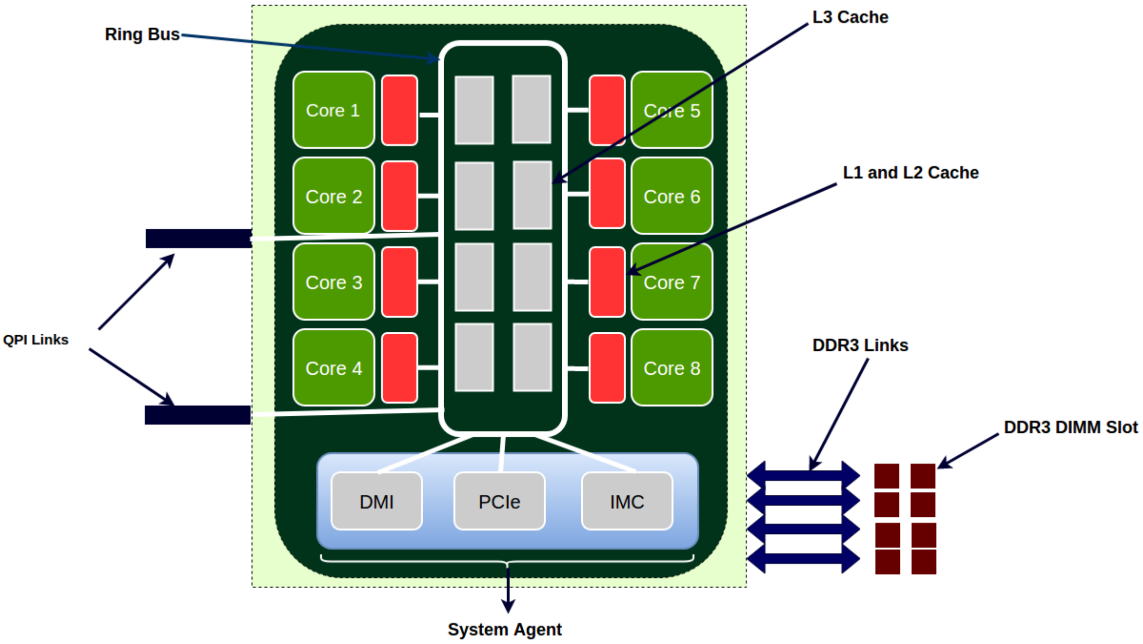


Figure 2.7: Intel Xeon E5-2600 series processor configuration

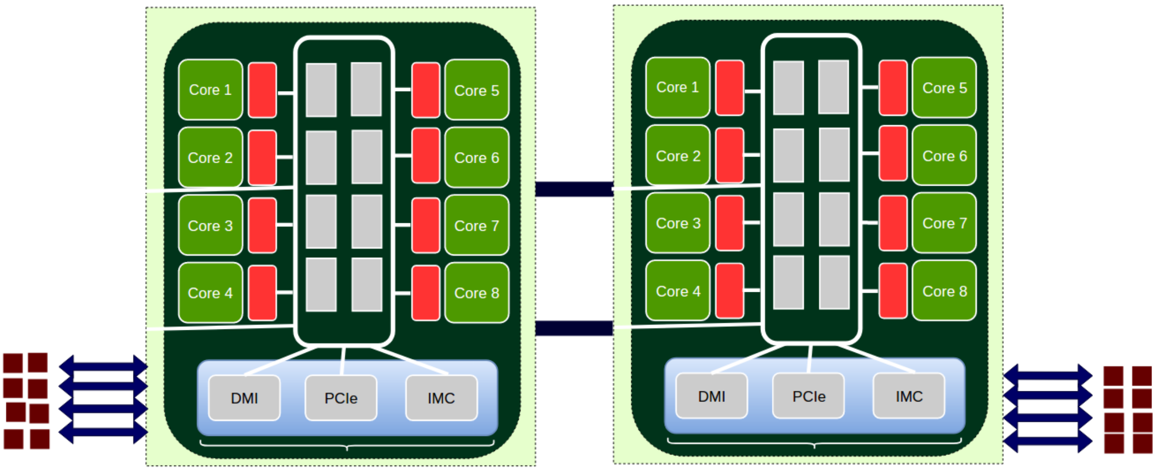


Figure 2.8: Intel Xeon E5-2600 series dual-processor server

2.3.1. Challenges for Performance on Multi-core Machines

This section will concisely explain very basic yet one of the most important performance issues in multi-core machines.

Cache Coherency and False Sharing

In multi-core machines, there is a huge potential for conflict if several processing cores have the same copy of data in their cache. *Cache coherence* is the problem of ensuring that all cached data are an exact copy of main memory. For example two cores have a copy of the same data in their private L1 cache, and one modifies its copy. Now the other core has cached data which is no longer valid or has inaccurate copy of the counterpart : the processor will invalidate the copy of the item, and in fact the whole cacheline and therefore this process of updating or invalidating cachelines is known as maintaining *cache coherency*. This phenomenon will waste bandwidth, which otherwise could have been used for load or store instructions.

The cache coherence problem can even occur if the core accesses different items, but they fall on the same cacheline and this problem is specifically known as *False Sharing*. The most common case of false sharing occurs when processing cores update consecutive locations of an array. The continous update of cacheline by multiple cores has a huge impact on performance and therefore should be avoided.

2.3.2. NUMA Architecture

This section briefly explains Non Uniform Memory Access (NUMA) architecture and highlights challenges associated with it to achieve high performance. Figure 2.8 shows an actual Intel Xeon E5-2600 dual-CPU server NUMA machine and Figure 2.9 schematically shows typical working of NUMA machine. The name NUMA suggests that memory access time for a certain set of CPUs is different from another set of CPUs. NUMA machines have been derived from their counterpart UMA machines where memory access time is the same for all CPUs. The main limitation of UMA architecture is low bandwidth scalability due to congestion of memory controllers upon increasing number of CPUs on one chip.

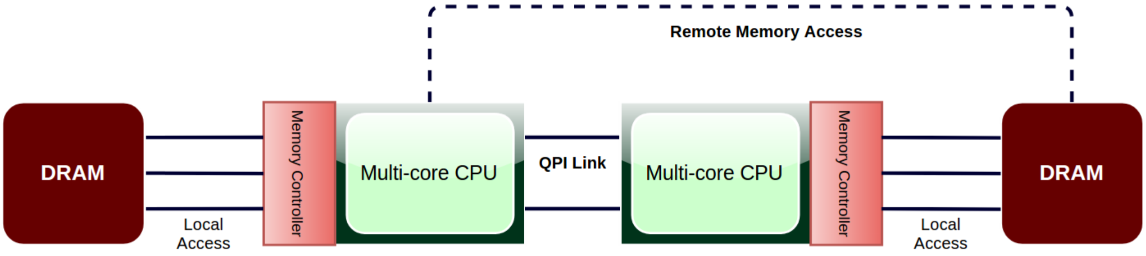


Figure 2.9: NUMA architecture

NUMA moves away from a centralized pool of memory and introduces topological properties. The NUMA architecture allows bandwidth scalability by first separating cores on different chips and then adding multiple independent memory controllers allowing each CPU to have their own memory address space thus reducing the load on a single memory controller and scaling bandwidth. The Concept of local and remote memory is depicted in Figure 2.9. Remote memory access has additional latency overhead as compared to local memory, as it has to traverse an interconnect and connect to the remote memory controller.

2.3.3. Performance Issues on NUMA Architecture

Optimizing codes on NUMA machines is not easy and involves delicate and careful treatment of cores and memory subsystems. This section will highlight some of the key issues for sub-optimal performance and plausible solutions for improving code performance on NUMA systems. Memory allocation in NUMA typically demands more attention and knowledge of operating system's memory placement policies. Special attention has to be given to applications that span over multiple NUMA nodes for optimal code performance.

Remote Memory Access

Remote memory access is one of the main reason for sub-optimal performance. Table 2.1 shows the penalty caused by remote memory access on Intel's Xeon E5-2600 processor according to NUMA distance. It is clear from table 2.1 that remote memory access is twice is expensive as local memory access.

Table 2.1: Remote memory access penalty factor

Numa Node	0	1
0	1x	2x
1	2x	1x

Remote memory access also occurs, if memory allocated by the master thread on one node is accessed from worker threads on another node.

Thread Migration

Modern operating systems assign application threads to processor cores using schedulers. A given thread will execute on an associated core for some period before being swapped out by OS scheduler as other threads are given a chance to execute. Thread migration from one core to another poses a problem for NUMA architectures because it can disassociate a thread from its local memory pool causing remote memory access and the significant increase in total computational time. The complexity of thread increases upon increasing number of CPU sockets.

Solution - Data Placement and Thread Pinning

Thread pinning and data placement are key components in achieving good performance on NUMA architecture. Thread pinning or processor's affinity is the endurance of particular thread with a set of resource instance despite the availability of other instances. Exercising processor affinity or thread pinning avoids thread migration to another CPU socket and back memory allocations local to processing cores. Data placement is another important aspect of high-performance scientific computing. The more often that data can effectively be placed in memory local to the processing core, the more overall access time will be benefited. Thread pinning and data locality are somewhat related to each other. By forcing thread pinning or processor affinity, local data allocation can be ensured.

Thread pinning and local data placement are just enzymes to the path towards high-performance computing. The real challenge is to design data traversal patterns which match the design of underlying hardware. Data traversal patterns decide usage of memory subsystems, therefore, controlling code performance in these architectures. The next chapter will focus on analysing performance impact of data traversal on simple experimental modified stream benchmarks.

3

Analyzing Performance Impact of Data Access on NUMA machine

3.1. Introduction

Application performance is largely governed by the arithmetic intensity and the memory access pattern of the computational kernel. The gap between CPU's processing power and memory speed is a major roadblock to achieving high performance for bandwidth limited kernels. The primary resource limitation is the memory controller and the L3 cache which rapidly gets saturated and inhibits high performance. The multicore architecture features complex cache coherent mechanism which may have the severe performance impact and therefore it is important to develop strategies to exploit all levels of memory subsystem up to the highest level of cache efficiently for good performance. This chapter focuses on performance bottlenecks and strategies to reduce the overall computational time and the energy consumption of simple kernels on high-end cache coherent NUMA machines, unveil valuable insights into serial and parallel behaviour and summarise strategies that can be adopted for high bandwidth utilisation and high floating point performance.

STREAM [1] is a popular and well established memory bandwidth benchmark, but it lacks NUMA aware support which makes it useless for analysis on cache coherent NUMA machine. Therefore a modified stream benchmark has been developed for an in depth analysis and experimentation on wide range of performance parameters.

3.2. Test System Specifications

This section gives a very concise introduction to the state of art x86 server HP Z280. The HP machine supports Intel Xeon E5 2687w v2 processor architecture shown in Figure 3.1. The HPC system based on Intel's Xeon E5 2600 product family provides 80 % improvement in performance and 50 % improvement in power efficiency compared to its previous generation Intel Xeon 5600 series.

Processor Micro-architecture

The Intel Xeon E5 2600 family supports Sandy Bridge EP microarchitecture which is an evolution of Nehalem EP microarchitecture. Each processing core has dedicated 32KB of L1 data cache, 256KB of L2 cache and shared L3 cache. The Sandy Bridge microarchitecture also introduces Intel Advanced Vector Instructions(AVX), a 256-bit instruction set to Intel SSE which can support up to 16 single-precision or 8 double-precision floating-point operations per cycle. Sandy Bridge microarchitecture also supports Running Average Power Limit (RAPL) interface to power measurements and analysis. The RAPL exposes performance

[1] http://www.cs.virginia.edu/stream/ref.html

Table 3.1: Test-system specifications

Processor Family	Intel Xeon E5 2600 v2
Micro-processor Architecture	Sandy Bridge EP
Cores	8
Processor Frequency	3.1 GHz
L3 Cache	20 MB
DRAM	16 GB
DDR3(1600 Mhz) Memory Channel(Active)	2
Max Memory Bandwidth	25.1 GB/s
NUMA Penalty	2x

counters to measure the energy consumed by CPU cores and the memory module.

The Intel Xeon E5 product family also improves high bandwidth and scalable interconnect ring bus liking cores, last level cache, PCIe and Integrated Memory Controllers. The shared L3 cache of 20 MB per CPU socket is divided into slices of 2.5 MB per core although each core can address the entire L3 cache which increases its overall bandwidth to L3 cache. The latency to the L3 cache is significantly reduced from 36 cycles in the previous generation to 26-31 cycles. Ring architecture for L3 cache brings high bandwidth and scalability which is very crucial for bandwidth limited kernels. The current processor has a maximum support of 4 memory channels out of which only two are active in the test system provided by Deltares, each comprising of 8 GB 1600 MHz DDR3 DIMM slot providing a total bandwidth of 25.1 GB/s per CPU socket to 16 GB of DRAM per CPU and the 2 Quick Path Interconnect (QPI) links between two processors provide overall bandwidth of 32 GB/s. Table 3.1 shows the technical specification of one CPU socket.

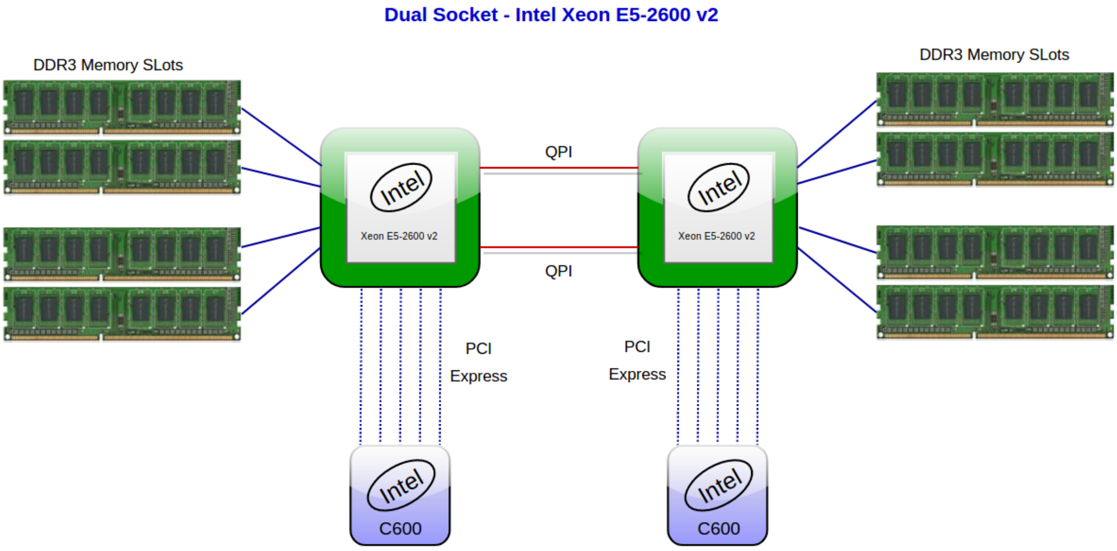


Figure 3.1: Dual-socket Intel Xeon E5 2600 v2 series architecture

There has been a lot of improvements and evolution over time on this machine in terms of bigger and faster cache, more cores/socket, and increased memory bandwidth. Since we are converging towards increasing parallelism within a processor and more bandwidth to DRAM, the majority of scientific kernels should be adapted to take that advantage. This chapter and coming sections will try to peek into the machine and understand performance optimizations on simple modified stream benchmarks.

3.3. Performance Analysis

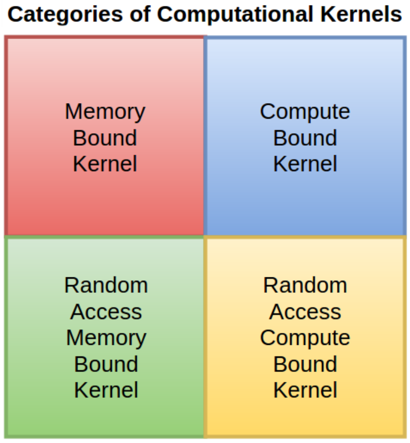

Categories of Computational Kernels
Memory Bound Kernel
Compute Bound Kernel
Random Access Memory Bound Kernel
Random Access Compute Bound Kernel

Figure 3.2: Categories of computational kernels

This section will explore performance bottlenecks in terms of limited hardware resources and CPU's processing power by studying four different computational kernels as summarized in Figure 3.2. The whole analysis provides the complete picture of strategies to be adopted for achieving maximum bandwidth utilization and maximum floating-point operation. This section will also provide a brief introduction to software tools used for profiling, debugging and performance measurements.

3.3.1. Softwares and External Libraries Used

Perf

Perf [2] tool is used for measuring performance counters under linux OS. Performanc counters in CPU are hardware registers that keep track of instructions executed, cache-misses suffered and branches miss-predicts. In this chapter, the Perf tool is used to measure energy consumption depending on different data access pattern. The Intel processor of the test system features Running Average Power Limit (RAPL) Model Specific Registers [5] which collects energy consumed by CPU-core and memory module in multiple of 15.3 µJ.

Numactl

The numactl [3] tool is used to bind specific threads to a set of processing cores and memory region to optimize data locality, avoid thread migration and avoid unnecessary OS scheduling. Memory locality or affinity is very important for high performance in NUMA machines, where remote memory access can be quite expensive compared to local memory access. The commands used for pinning threads to cores and allocating memory locally or remotely are summarised at web page of *IBM Knowledge Center* [4]

3.3.2. Modified Stream Benchmark

The modified stream benchmark [5] is developed in C++ and parallelized with OpenMP for the purpose of testing and experimenting the above explained dual socket server machine. The NUMA-aware memory allocation by the first touch principle ensures that data is allocated local to processing cores, pinning of threads is enforced to fully avoid thread migration and hyper-threading is turned off to achieve performance. This stream benchmark is used to understand the behaviour of computational kernels under different data traversal schemes.

[2] https://perf.wiki.kernel.org/index.php/Main_Page

[3] https://linux.die.net/man/8/numactl

[4] https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaai.hpctune/cpuandmemorybinding.htm

[5] https://github.com/computingdolas/Stream_Benchnmark

Compilation and Execution

The following code kernels are compiled with gcc version 5.4.0 and with std=c++11 -O3 -fopenmp flags.

3.3.3. Analyzing Memory Bound Kernel on ccNUMA machine

The parallel ADD operation shown in Kernel-1 involves very little computation and is bandwidth limited.

Kernel-1

```
for (int i = 0 ; i < K_times ; ++i){
    // Parallel Region
    #pragma omp parallel shared(a,b,c,N) num_threads(<numthreads>)
    {
            double wtime = omp_get_wtime() ;
            #pragma omp for schedule(static)
            for (uint64_t k =0 ; k < N ; ++k){
            // ADD Operation
                a[k] = b[k] + c[k]  ;
            }
            wtime = omp_get_wtime() - wtime ;
            #pragma omp master
                    global_time += wtime ;
    }
```

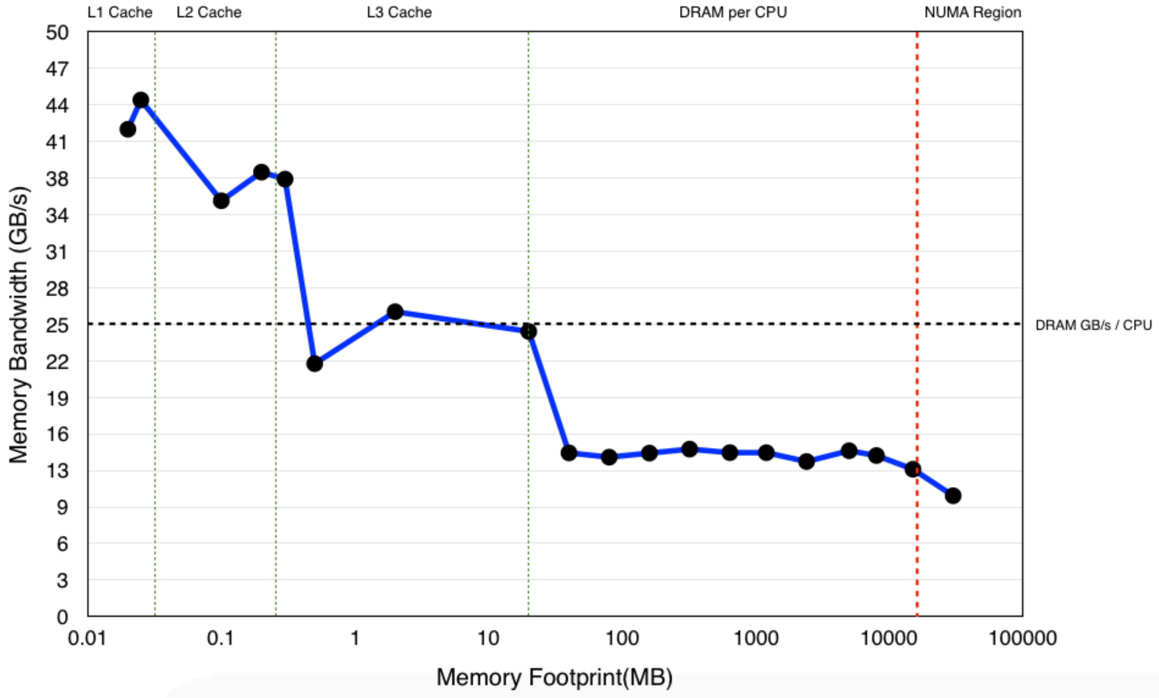


Figure 3.3: Analysis of memory-bound Kernel-1 on Single Core

The aim of this particular analysis is to clearly capture serial and parallel performance of bandwidth-limited Kernel-1 on the dual socket server machine as shown in Figure 3.3 and 3.4. Figure 3.3 shows single-core performance of Kernel-1 for entire memory footprint. It is clear that on-chip memory offers ≈3x-4x increase in bandwidth which steadily decreases

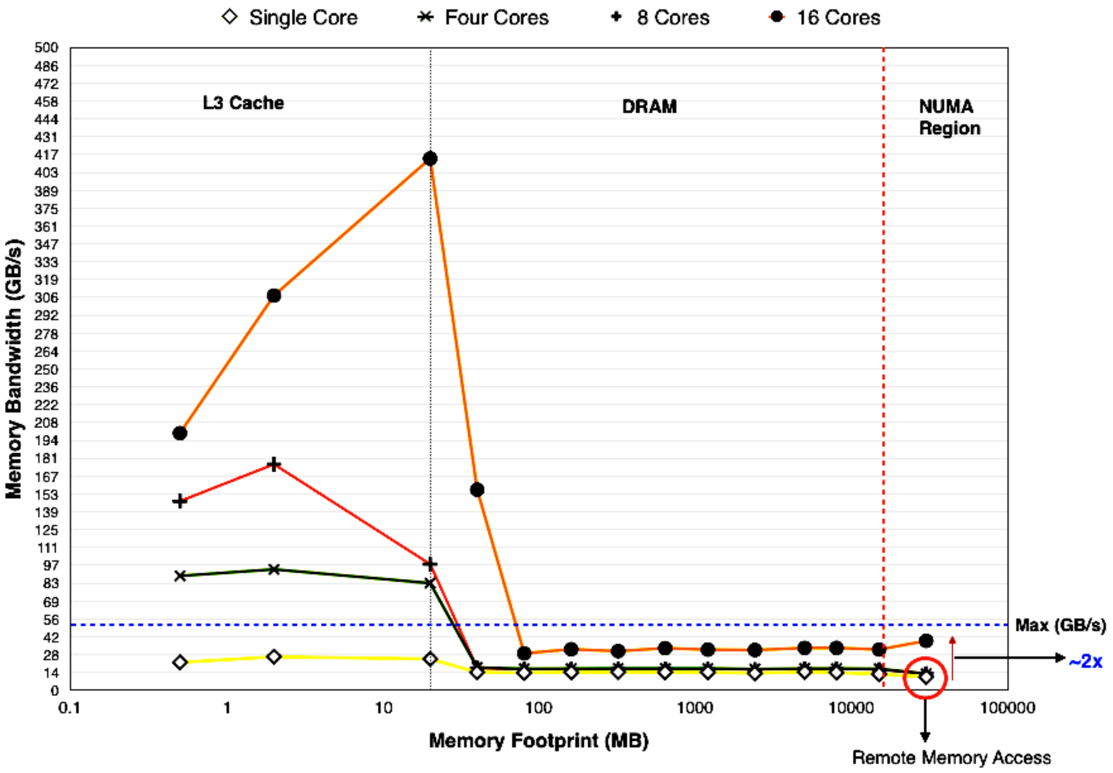


Figure 3.4: Parallel performance analysis of memory-bound Kernel-1 on 16 cores

until data is accessed from main memory (DRAM). The analysis also shows that maximum bandwidth achieved by Kernel-1 is ≈16 GB/s which is 65% of theoretical maximum. The crucial point here is that on-chip memory offers low latency and higher bandwidth compared to main memory (DRAM) and therefore programming effort should be invested to align data as close as possible to processing cores

On the other hand, the bandwidth limited kernels do not gain speed-up or scale by increasing the number of active cores especially when accessing data from main memory (DRAM) as shown in Figure 3.4, since it depends on memory access pattern, memory types (DDR3 1600), and number of active DIMM slots/ channels all of which combined determine theoretical maximum. In Figure 3.4 the perfect linear scaling in bandwidth is observed upon increasing core count from 1 to 16 until data is accessed from L3 cache because each core has private 32 KB of L1 data cache and 256 KB of L2 cache and 20MB L3 cache logically shared and physically separated for one CPU which enhances the core's ability to access the data independently thus scaling the bandwidth. As the processing cores start to access data from DRAM, there is a sudden drop in bandwidth, which is limited by the maximum theoretical bandwidth of 25.1 GB/s per CPU socket. In Figure 3.4 the maximum bandwidth achieved by Kernel-1 is 19 GB/s which is ≈75 % of theoretical peak.

The Figure 3.3 shows a sudden drop in bandwidth upon accessing memory in NUMA region because of the fact that, on the current machine remote memory access is twice as expensive as local memory access and on the other hand orange line in Figure 3.4 represents 16 cores operating on Kernel-1. There is ≈2x increase in bandwidth to 40 GB/s due to two more operational memory channels from another CPU. This is one of the major advantages of NUMA machines, the memory bandwidth to DRAM scales linearly if the number of CPU socket is increased. This analysis concludes that even though remote memory access can be quite expensive, NUMA nodes if operated independently can scale bandwidth linearly.

3.3.4. Analyzing Compute Bound Kernel on ccNUMA machine

The ADD operation shown in Kernel-2 involves expensive sin() and cos() operation and therefore is compute bound.

Kernel-2

```
for (int i = 0 ; i < K_times ; ++i){
    // Parallel Region
    #pragma omp parallel shared(a,b,c,N) num_threads(1)
    {
            double wtime = omp_get_wtime() ;
            #pragma omp for schedule(static)
            for (uint64_t k =0 ; k < N ; ++k){
            // ADD Operation
                a[k] = sin([k]) + sin(c[k])  ;
            }
            wtime = omp_get_wtime() - wtime ;
            #pragma omp master
                    global_time += wtime ;
    }
```

Figure 3.5 shows bandwidth drop factor due to various combinations of compute bound kernels. It is clear from the analysis shown in Figure 3.5 that data is stalled for the maximum of ≈300x compared to add operation in Kernel-1, drastically reducing its potential to utilize maximum operational bandwidth and thus leading to an increase in overall computational time. This type of analysis gives the complete opposite picture of bandwidth limited kernel. The more the complexity, the more is the drop in bandwidth. The speed of memory now does not make a difference at least for unit stride access.

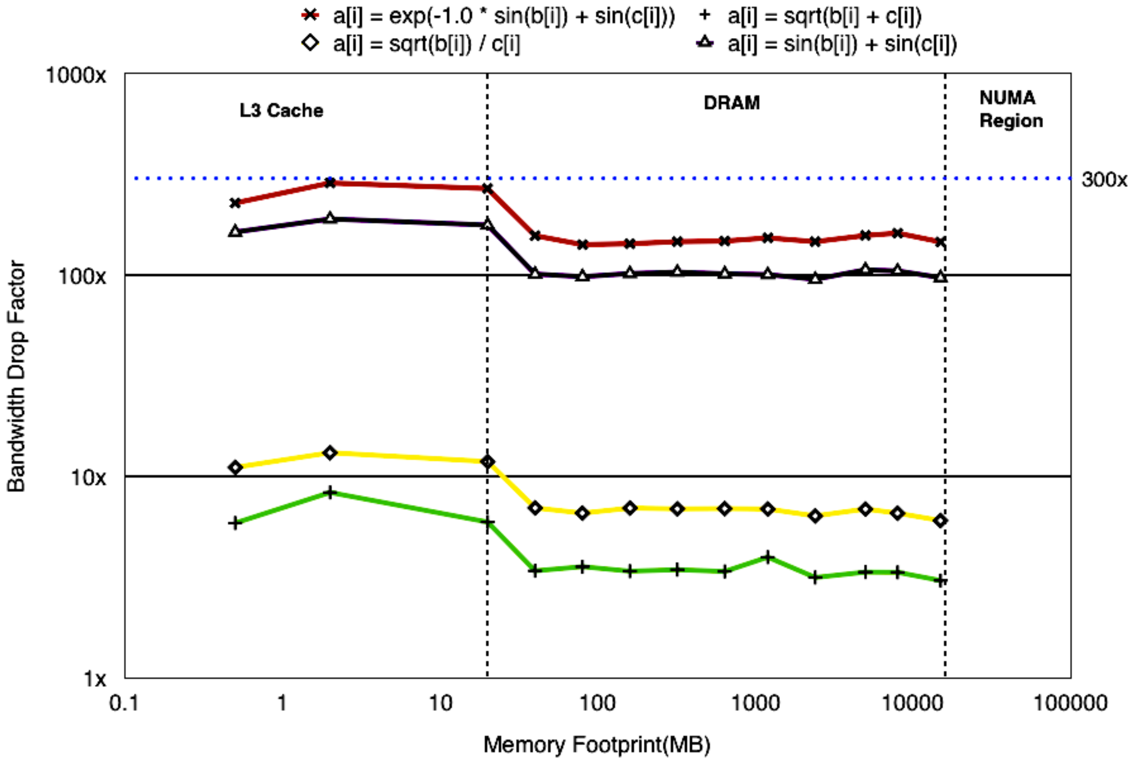


Figure 3.5: Single-core performance analysis of compute-bound Kernel-2

Figure 3.6 shows the parallel behavior and scalability of the compute-bound kernel with unit stride access on 8 and 16 cores respectively. The compute-bound kernel achieves the maximum speed of 7.3x with 8 active cores and ≈14x on utilizing 16 cores of the test system. Comparing to the bandwidth-limited analysis shown in Figure 3.4, it is clear that although the compute bound kernel is extremely expensive due to memory stalls, it can become scalable on multiple cores easily. Data distribution and data placement are very important steps and by the virtue of NUMA aware data allocation, it is possible to maintain data locality which is the main thrust to achieve high performance

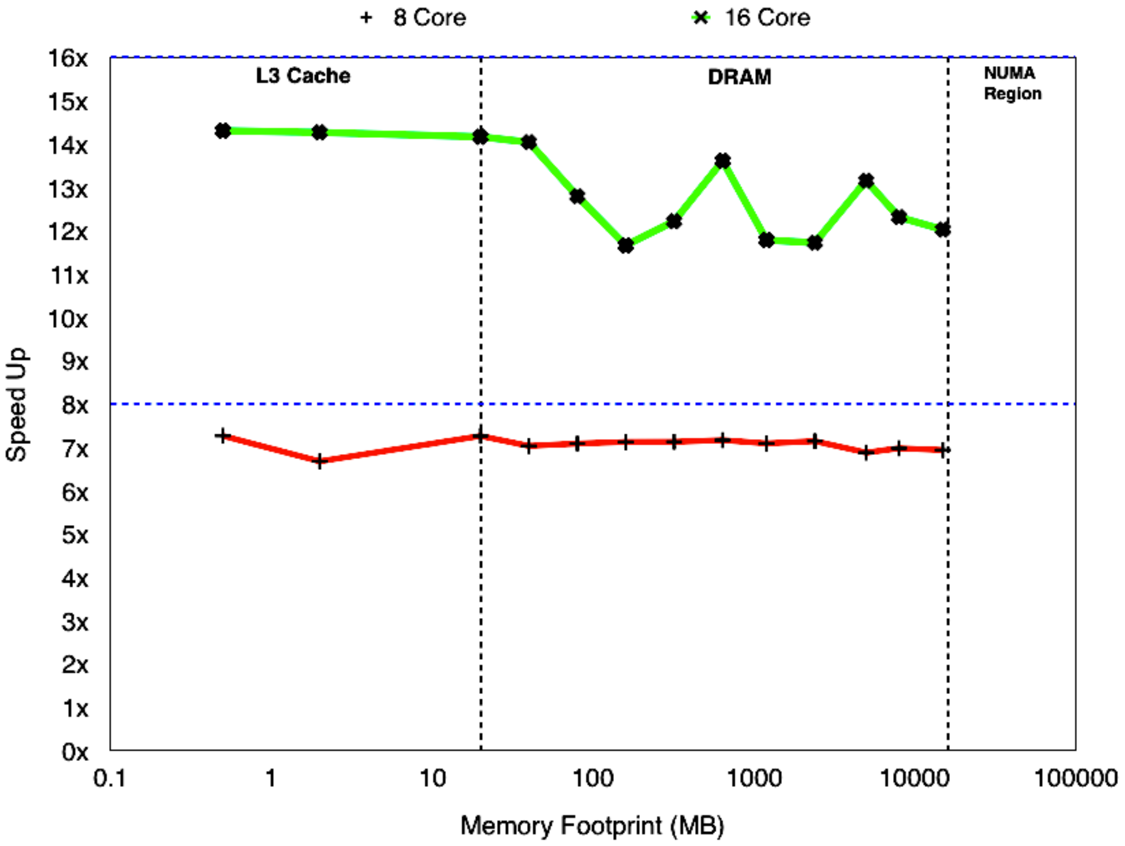


Figure 3.6: Speed-up of compute-bound Kernel-2 on 8 and 16 cores

3.3.5. Analyzing Effect of Indirect Random Access on Memory Bandwidth Limited Kernel

This analysis forms the central theme of the master thesis. Analyzing impact of random access on simple ADD kernel gives insight into drop in computational performance for both serial and parallel code. In Kernel-3 a **random[]** array is initialized and shuffled so as to have completely unpredictable behavior. Figures 3.7 and 3.8 depict the serial and parallel analysis respectively. From the serial analysis shown in Figure 3.7 one can see that, random access causes a drop in bandwidth to 0.4 GB/s from maximum of ≈15 GB/s. Another compelling point to understand is that unit stride access maintains certain bandwidth drop across each memory levels, but random access drops non-linearly across memory access to DRAM.

Kernel-3

```
for (int i = 0 ; i < K_times ; ++i){
    // Parallel Region
    #pragma omp parallel shared(a,b,c,N) num_threads(<numthreads>)
    {
            double wtime = omp_get_wtime() ;
            #pragma omp for schedule(static)
            for (uint64_t k =0 ; k < N ; ++k){
            // ADD Operation
                a[random[[k]] = b[random[k]] +
                                c[random[k]]   ;
            }
            wtime = omp_get_wtime() − wtime ;
            #pragma omp master
                    global_time += wtime ;
    }
```

Figure 3.8 shows parallel analysis of random access on 8 cores. There is a huge drop in bandwidth for in-core memory and a significant drop for memory accesses to DRAM. It is clearly visible that random access completely inhibits particular kernel to achieve operational maximum bandwidth and degrades performance. Bandwidth in Figure 3.8 is larger than in Figure 3.7, which is due to the fact that data is processed in parallel by processing cores.

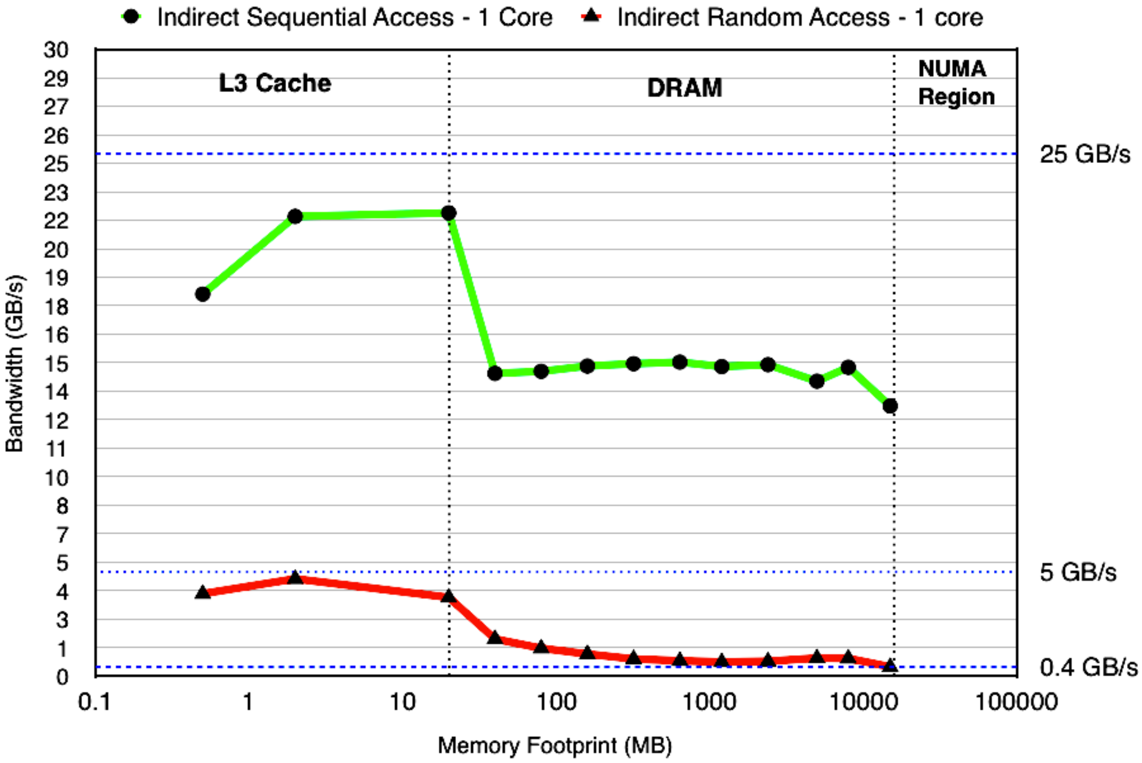


Figure 3.7: Single-core performance analysis of random access on Kernel-3

Energy Studies on Bandwidth Limited Kernel

Energy consumption is one of the major performance parameters which should also be studied because reduction in energy usage is as important as reduction in computational time.

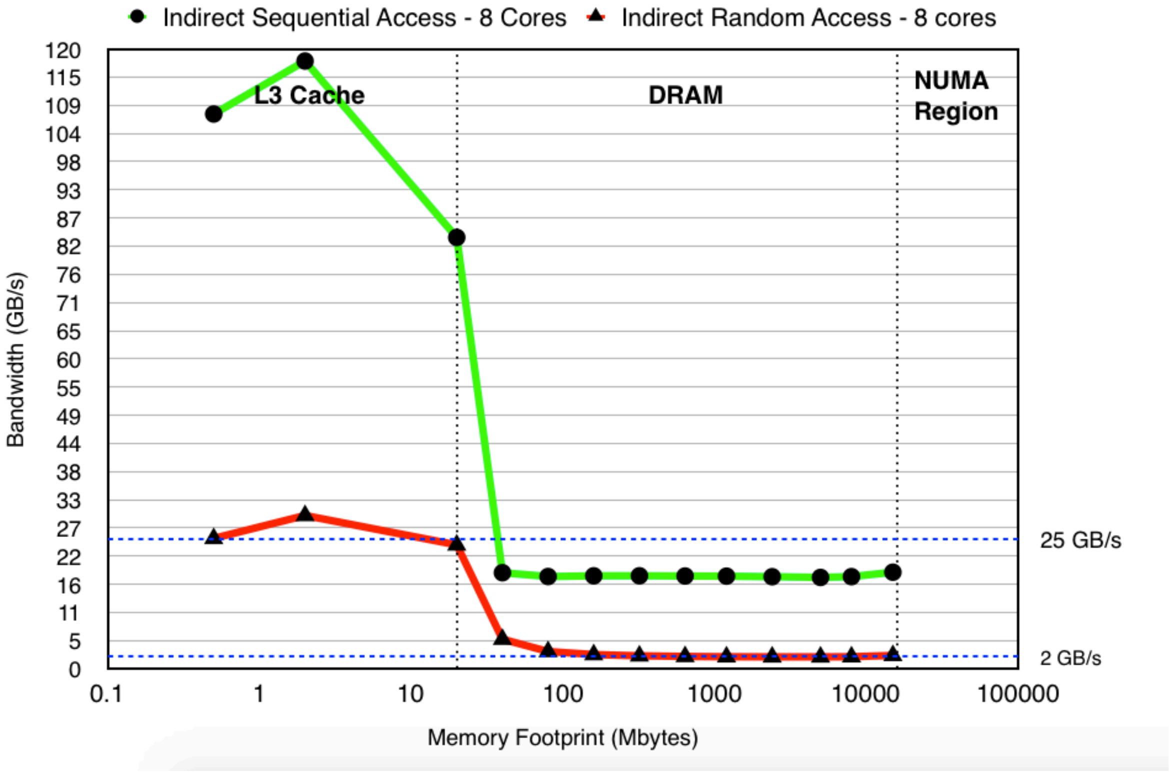


Figure 3.8: Parallel performance analysis of random access on Kernel-3

This study will analyse the impact of random access on energy/cpu-core and energy/memory module and will peek into energy consumption on bandwidth limited Kernel-3 shown above by measuring hardware performance counters using perf tool.

Figure 3.9 shows the impact of random access on energy consumed by a processing core per iteration. It is clearly visible that energy consumed by a core per iteration for random access is one order of magnitude higher than the sequential access. The energy consumption increases with increase in array size which is dictated by the memory footprint. The crucial point here is to understand that high-performance computing is not only about efficient usage of processing cores for reduction in computational time but also ensuring judicial consumption of energy. Random access has devastating effect on energy consumption and should be avoided for sustainability.

Figure 3.10 shows the impact of random access on energy consumption of memory module. It is very clear that the energy consumed by DRAM is not as large as consumed by a CPU core but it makes significant difference to overall energy consumption. It is certain that energy consumed by memory module is very low until data is accessed from L3 cache, but it increases exponential as soon as memory footprint crosses L3 cache. Also energy consumed by memory module upon random access is significantly higher due to non-linear increase in memory traffic. It is also one order of magnitude higher than the sequential access.

3.3.6. Analyzing Effect of Indirect Random Access on Compute Bound Kernel

This section investigates the impact of random access on computational performance of compute bound kernel shown in Kernel-4.

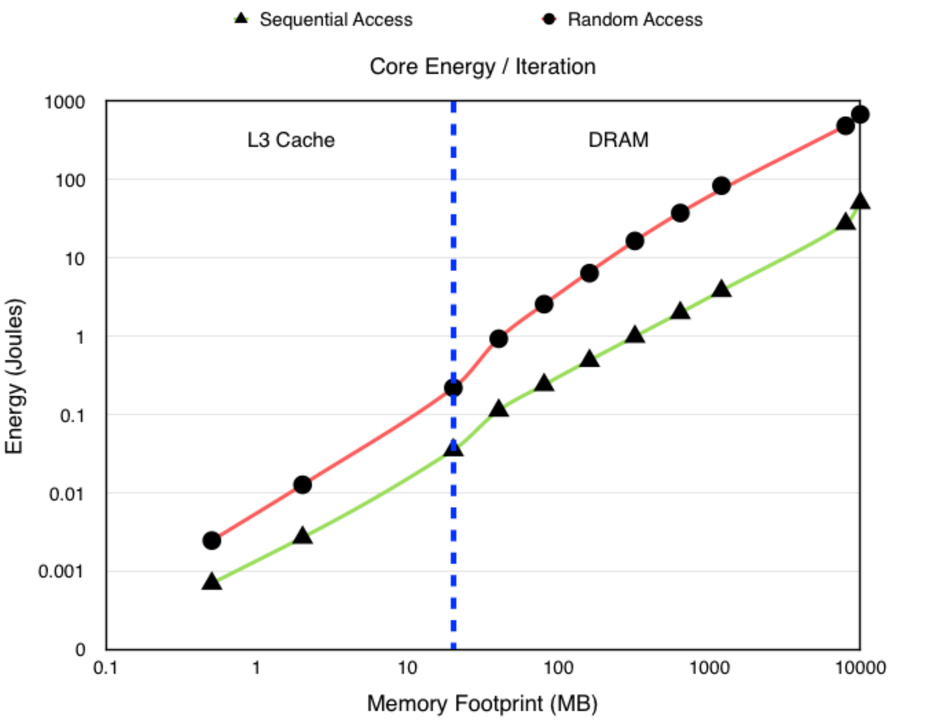


Figure 3.9: Analysis of energy consumed per iteration by a CPU core on bandwidth limited Kernel-3

Kernel-4

```
for (int i = 0 ; i < K_times ; ++i){
    // Parallel Region
    #pragma omp parallel shared(a,b,c,N) num_threads(<numthreads>)
    {
            double wtime = omp_get_wtime() ;
            #pragma omp for schedule(static)
            for (uint64_t k =0 ; k < N ; ++k){
            // ADD Operation
                a[random[[k]] = sin(b[random[k]]) +
                                sin(c[random[k]])  ;
            }
            wtime = omp_get_wtime() − wtime ;
            #pragma omp master
                    global_time += wtime ;
    }
```

Figure 3.11 indicates that, there is a maximum drop of ≈2.2x when the array length reaches maximum DRAM limit of 16 GB per CPU socket. There is no loss until data is accessed from L3 cache because of the simple fact that, the compute bound kernel takes much more time to process data and since L3 cache provides high bandwidth and low latency so data reaches as fast as possible to processing cores. As soon as array length starts crossing L3 cache limit there is visible speed drop increasing nonlinearly due to the fact that processing cores cannot find next set of data sets immediately and has to request data from DRAM which degrades its computational performance.

Figure 3.12 shows the parallel performance of random access in the compute bound ker-

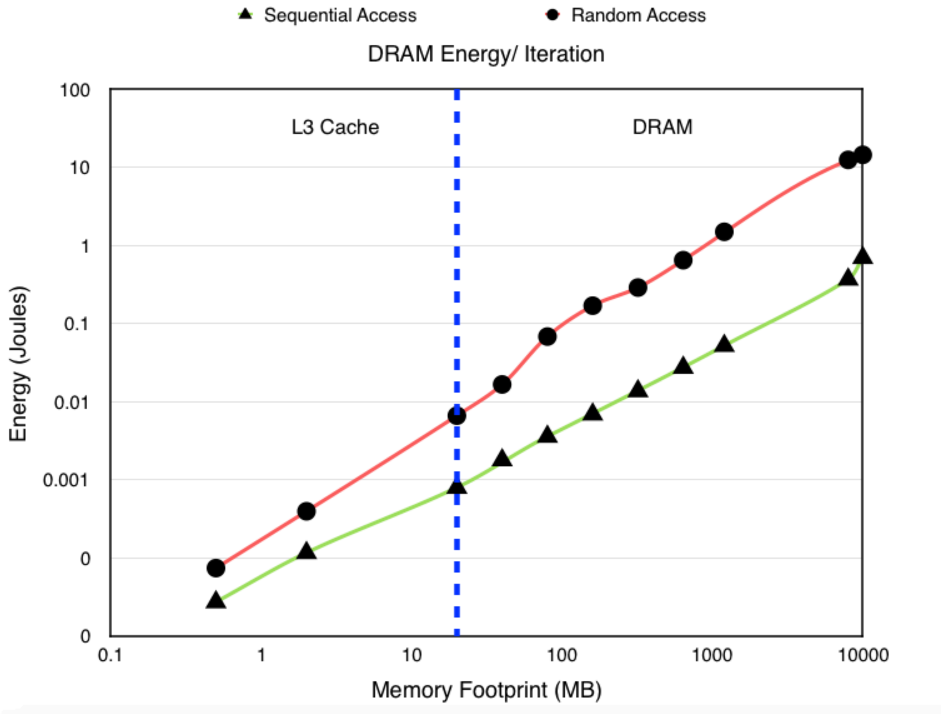


Figure 3.10: Analysis of energy consumed per iteration by memory module on a bandwidth limited Kernel-3

nel. It is surprising to see that, both random access and unit stride access appear to be scalable and achieve nearly same speed for every level of memory hierarchy. Well, there is no background literature as to why random access compute-bound kernels should achieve parallelism same as unit stride access, but there is a hidden reason of inefficient dedication of processing cores. Processors are twice as busy doing inefficient computation as clearly projected in Figure 3.11 while maintaining scalability due to the fact that Kernel-4 is so expensive that processing cores do not seem to worry about data arrival and departure but just processing it. This type performance analysis can be sometimes misleading in a sense that achieving parallelism is not certificate for high performance but there are numerous perspectives which should be looked in carefully for optimization.

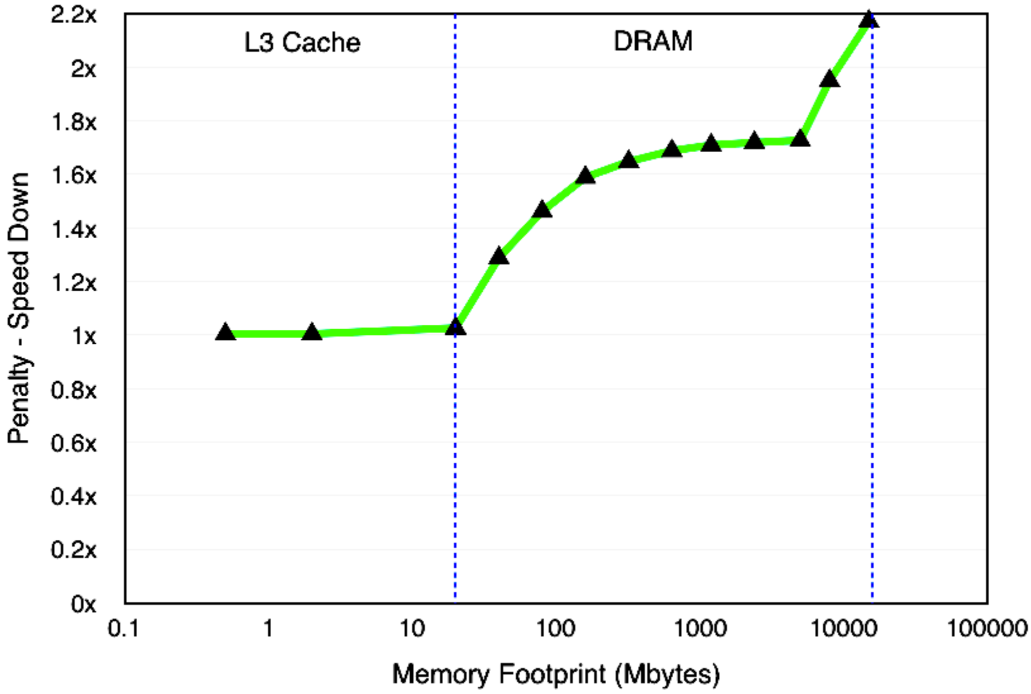


Figure 3.11: Drop in speed due to random access in compute-bound Kernel-4

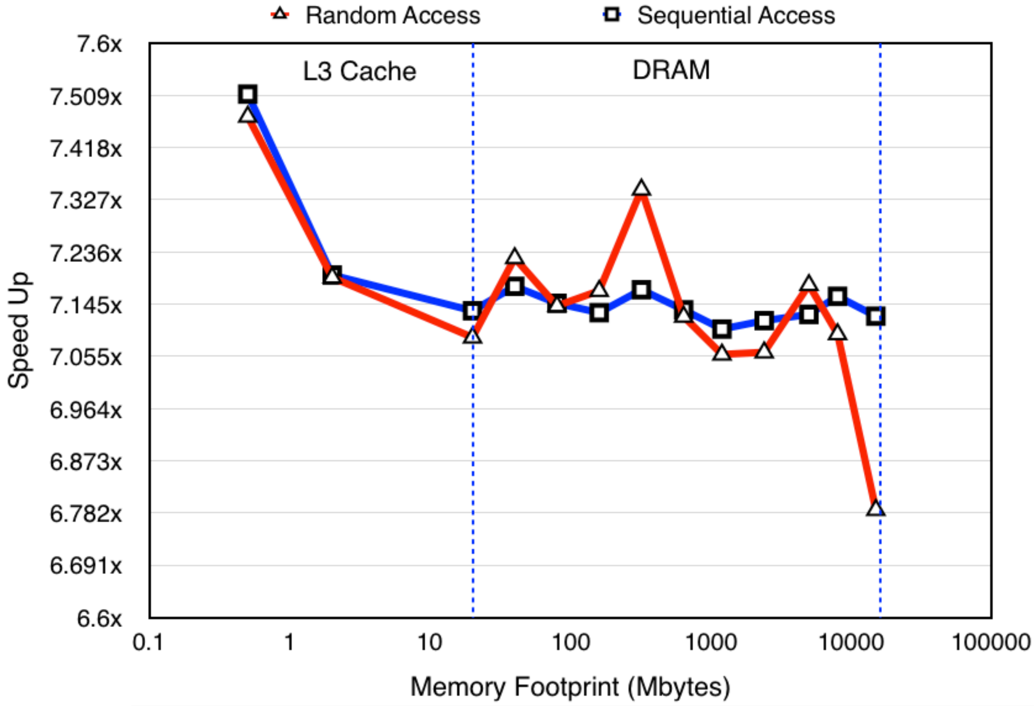


Figure 3.12: Parallel performance analysis of unit stride and random access on Kernel-4

3.4. Conclusion

This chapter started with understanding technical specifications of the test hardware and assessing performance potential of simple kernels with regards to memory bandwidth utilisation, computational time and energy consumption. The main goal of this chapter was to observe theoretical basis of high-performance computing experimentally on particular machine architecture and establish the importance of data traversal by analysing the performance of simple stream benchmarks. This section will highlight key points of analysis of memory and compute bound kernels.

The performance of memory-bound kernel is limited by the rate at which data can be delivered to the processing cores or technically speaking it is bounded by the maximum bandwidth offered by the underlying hardware. So increasing the core count on one particular processor will certainly scale up the bandwidth linearly if data fits into the L3 cache but will not improve the DRAM access performance since shared resources of the memory subsystem get saturated very quickly. Therefore, increasing the number of CPUs or NUMA nodes will only be able to scale up the overall bandwidth used by the kernel, allowing memory controllers of different NUMA nodes to access data independently. The attempt should be made to utilize cache data as efficiently as possible because it offers low energy, low latency and high bandwidth access to the data.

This chapter also discussed performance impact of random access on bandwidth utilization and energy consumption. The conclusion is random access causes increase in DRAM traffic, therefore, leading to a huge drop in operational bandwidth utilization and a significant increase in energy consumption both for processing cores and memory modules. Programming effort should be invested in aligning or accessing data as per underlying hardware to maximize operational efficiency of cache hierarchy and memory subsystems.

This chapter also analyses compute bound kernels. Compute bound kernels are limited by the ability of the computing core to process data and is no longer controlled by the speed of the memory module. Compute bound kernels are scalable to a large extent since data processing takes much longer compared to its arrival and therefore individual cores can then process data seamlessly without waiting for it. Random access of data in compute bound kernel do not have a devastating effect on performance since a penalty due to cache miss or redundant memory reference to DRAM is comparatively much less time consuming than data processing itself. Therefore the conclusion is that performance impact of random access to data in compute bound kernel is not acutely visible but still plays an important factor in achieving high performance.

4

An Introduction to Space Filling Curves

4.1. Introduction

This chapter introduces the concept of Space Filling Curves as versatile tool for efficient mesh data reordering algorithms for dynamic load balancing, mesh partitioning, domain decomposition and cache aware computing. The Space Filling Curves (SFC) have proven to be absolutely remarkable alternatives to travel spatial data structures and encourages energy efficient high-performance computing with the minimal amount of overhead involved.

There exists a vast number of algorithms for improving cache performance, performing domain decomposition, solving Molecular dynamics and N-body problems on distributed and shared memory computing architectures. Reordering algorithms like Reverse Cuthill-Mckee, Recursive Spectral Bisection or Multilevel Nested dissection are very robust and efficient in performing fast search for spatial data structures. However above-mentioned techniques require a large amount of investment in terms of programming and maintenance but none of them offer unifying approach towards domain decomposition and improvement in cache performance along with easy implementation and minimal computational cost [1].

Space Filling Curves on the other hand, offer easy implementation and embarrassingly parallel construction of three-dimensional index in space. The construction of Space Filling Curve involves finding a unique index and sorting of data structures relative to that index. The asymptotic complexity of constructing the curve is therefore bounded by the sort algorithm which orders the mesh along the curve. This can be accomplished by using standard serial quick sort with $O(NlogN)$ complexity.

Computational scientists have understood the importance of Space Filling Curves and these curves are already part of scalable scientific codes specifically for plasma simulations, fluid structure interaction, N-body problems and multi-physic Finite Element codes. Space Filling Curves apart from high-performance computing are also utilised for analysis in big data, graph analysis, geospatial analysis and communications. SFC have not been able to establish themselves as a cache friendly algorithms in the majority of high-performance scalable softwares. This section and later text will provide the comprehensive understanding of Morton-order Space Filling Curve and highlight its importance for cache aware high-performance computing on modern machines.

4.1.1. Mathematical Description

Mathematically speaking Space Filling Curves belong to family of fractal curves which recursively duplicate coarser design patterns on an infinite scale in any number of dimensions. They are mappings from a $\boldsymbol{d}$ dimensional space to 1-dimensional space [13].

$$c : (1 \cdots n)^d \rightarrow (1 \cdots n^d) \tag{4.1}$$

Space Filling Curves do not intersect themselves and therefore, a unique ordering of the points in $\boldsymbol{d}$ dimensional space is obtained. These curves strongly enjoy locality property which makes them good enough for high-performance computing in the wide range of topics in computational science and engineering. A mathematical description of 2D and 3D locality of Hilbert-order SFC is given in detail in [10]. For 2D Hilbert curve authors in [10] propose

$$|c(i) - c(j)| < \sqrt[2]{6(|i - j|)} \quad \forall \quad i, j \in \mathbb{N} \tag{4.2}$$

and for 3D order Hilbert curve as

$$|c(i) - c(j)| < \sqrt[3]{33.2(|i - j|)} \quad \forall \quad i, j \in \mathbb{N} \tag{4.3}$$

The origin of SFCs can be attributed to the development of Cantor sets [1]. The Hilbert-order SFC or Morton order SFC for the square or cube can be generated by recursion. Space Filling Curves are not always generated explicitly, instead their mathematical description is used to calculate a spatial index associated with each entity. In d dimensions and p as the depth of recursion, the SFC provides an ordered enumeration of an imaginary bounding Cartesian grid having 2^{dp} cubic cells in which the computational domain is fully embedded. Each cubic domain has unique SFC index and is duplicated on the entity which lies inside it. The algorithm used to generate SFC index for unstructured meshes is described in detail in Section 4.1.3 .

4.1.2. Space Filling Curves in Use

Dynamic Load Balancing and Efficient Domain Partitioning

Load imbalance is one of the important roadblocks for scientific codes to become scalable on very large HPC systems. An efficient mesh partitioning becomes an essential component for scalable design of numerical applications. Space Filling Curves have been used for dynamic load balancing and efficient partitioning as documented in Campbell et al. [3] and Vo et al. [15] .

On the other hand, there are many popular software such as ParMetis [2] which have been used to solve problem of dynamic load balancing, but SFCs as a better alternative is presented and argued in [9]. Authors observe that, SFCs provide memory efficient alternative and easy local implementation as opposed to techniques available in ParMetis which requires global knowledge of connectivity matrix. Authors have applied Space Filling Curves to balance dynamically changing workloads on static but unstructured mesh for an adaptable flow solver. Authors in [9] concluded that, ParMetis was not able to scale beyond 65,536 processors, in contrast to Space filling curves which scales well upto 294,912 processors with only 1- 2 % drop in quality of partitions as compared to ParMetis.

Energy Aware Computing

Energy aware computing is one of the most important research topics in today's high-performance computing systems and it is important thrust for economically operating HPC systems adding another yet challenge to run and maintain scalable codes on big HPC systems.

Energy efficiency analysis using Space Filling Curves has been studied in [12]. The authors investigated locality effects of Hilbert-order and Morton-order SFC on dense matrix-matrix multiplication. It was observed that tiling behaviour of SFC is one of the most important artefacts enabling cache-aware computing. It was shown that energy consumption is exclusively proportional to CPU's execution time and increasing memory traffic to DRAM also leads to an increase in energy consumption. It was also observed that amount of energy consumed by CPU is much larger as compared to memory and finally it was concluded that the gap between memory speed and CPU performance has significant impact on energy efficiency and therefore efficient utilization of memory hierarchy is as important as minimizing

[1] https://en.wikipedia.org/wiki/Cantor_set

[2] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

execution time.

The authors in [12] also observe that the overhead of generating Morton-order SFC balances the speed up and energy saving for dense matrix matrix multiplication as opposed to Hilbert-order SFC which only provides moderate improvement in locality properties as compared to Morton-order SFC. This is one of the foremost reasons to choose Morton-order SFC for data traversal as opposed to Hilbert-order in this work.

4.1.3. Space Filling Curve Construction for Arbitrary Mesh

Space Filling Curves can be constructed for many type of data objects. This section and coming ones will specifically describe their construction for grid based objects consisting of vertices and elements. Space Filling Curves reorder elements and vertices to preserve spatial locality leading to a significant drop in the cache misses and redundant bus cycle to the DRAM.

Understanding Recursion Depth

In this section, quadtree [3] data structure will be highlighted for a very simple 2D structured grid. A quadtree is a two dimensional analogue of the octree [4], both are used to partition two and three dimensional space by recursively dividing computational domain into four or eight parts.

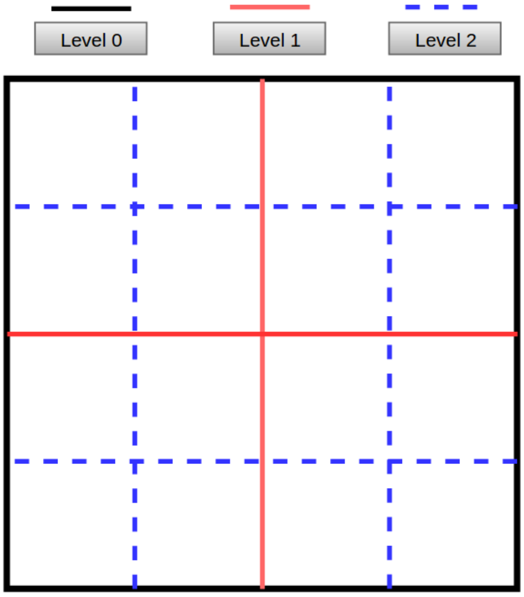


Figure 4.1: Level 3 recursion depth

Figure 4.1 illustrates a two-dimensional grid. The black square corresponds to zero recursion depth and similarly red and blue partitions correspond to the first and second level of recursion respectively. The equivalent quadtree representation is shown in Figure 4.2 where the head node is followed by four red leaf nodes and so on. This type of data structure is widely used to partition two dimensional structured computational domains, but for generating a cache oblivious mesh layout with Morton-order SFC, this type of arrangement is superimposed on existing mesh as shown in Figure 4.3 and SFC index is calculated based on position of mesh element in the Cartesian grid.

[3] https://en.wikipedia.org/wiki/Quadtree
[4] https://en.wikipedia.org/wiki/Octree

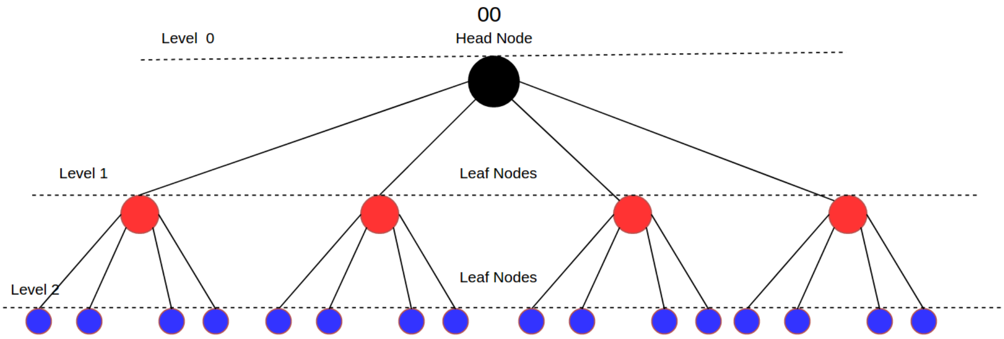


Figure 4.2: Quadtree illustration of level 3 recursion-depth

Calculating Space Filling Curve Index

The centroids of triangular elements in 2D and tetrahedral elements in 3D are calculated and an implicit Octtree of depth p is constructed where the root of the tree is bounding Cartesian imaginary grid and its leaf nodes are ordered 2^{pd} cells. The centroids represent each element uniquely and its SFC index is calculated by finding the cell index the centroid falls in, thus implicitly finding out element index. The more the depth of recursion, the more is the accuracy of the centroids to have unique cell indexes.

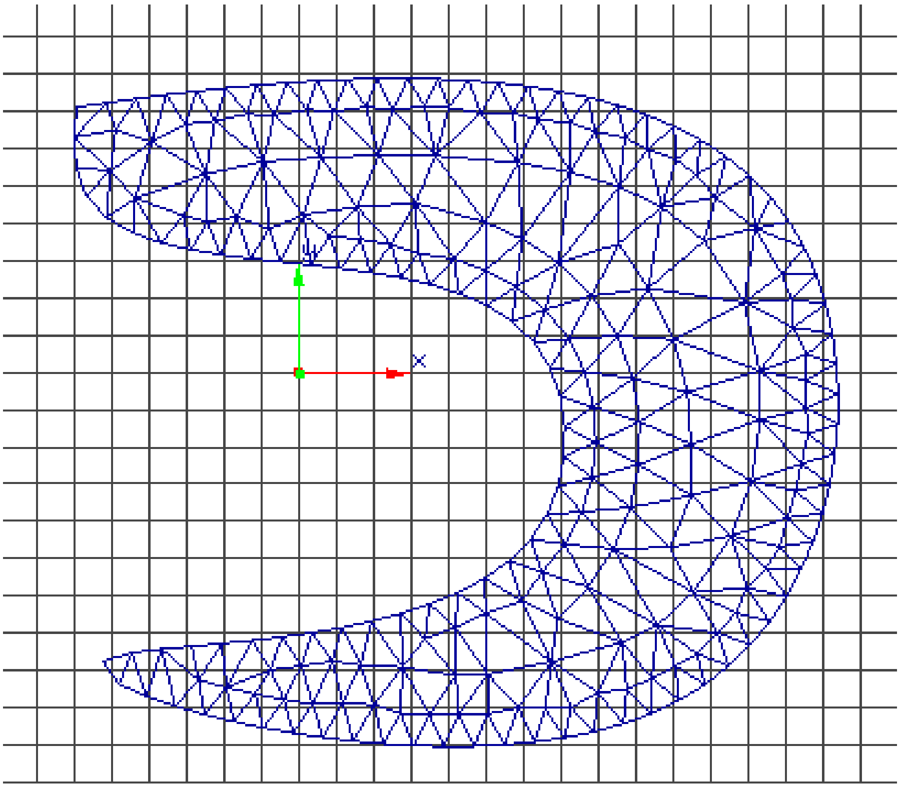


Figure 4.3: Superimposition of cartesian grid over arbitrary FE mesh

Algorithm 1 briefly describes the generation of the Morton-order SFC and Algorithm 2 describes bit inter-leaving technique used for calculating SFC index. The detailed description of these algorithms and the comparison with other optimal cache obvious data layouts (COML) which are based on NP-hard optimization problem are explained in Vo et al. [15].

```
Algorithm 1: Parallel Space Filling Curve generation
  Input  : Mesh file
     1. Store element and vertices data into appropriate data structures
     2. Compute imaginary Cartesian bounding box
  In Parallel foreach element in the mesh do
      Calculate Centroid
  end
  /* Initialize the number of recursion levels - N                  */
  In Parallel foreach element in the mesh do
      /* Pass the element, bounding box, number of levels          */
      MortonOrderCurve(e,boundingbox,N) /* O(N)                    */
  end
  /* Sort the Elements according to their SFC index                */
  parallelsort(element.begin(),element.end())
```

Bit Interleaving with Recursion Depth

This section explains bit interleaving with recursion depth to grasp SFC index calculation for regular and adaptive grids.

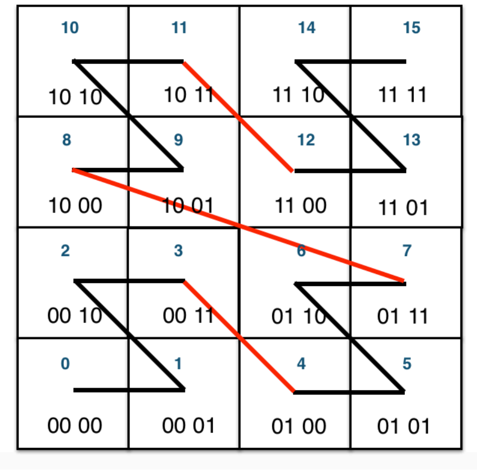


Figure 4.4: Construction of Morton-order Space Filling Curve for 2D regular grid and bit interleaving

Figure 4.4 illustrates a two dimensional grid with two levels of recursion and its equivalent quadtree representation in Figure in 4.5. Looking at any arbitrary cell of the two dimensional grid shown in Figure 4.4 it is clear that the first two digits represent the first level of recursion and the next two represent the second level of recursion. Concatenating these four bits together as shown in Figure 4.4 or concatenating bit indexes of each leaf node along with its parent node up till the head node as shown in Figure 4.5 and their equivalent decimal number corresponds to a SFC index. Similar technique can be applied to adaptive meshes also shown in Figure 4.6 and its equivalent quadtree in Figure 4.7. This approach can handle, an order of 2^{32} and 2^{21} number of cells in 2D and 3D respectively with machine supporting 64 bits of integer precision.

Algorithm 2: Morton-order SFC index

```
Result: Morton-order SFC index for each element
Input  : element, bounding box , number of Levels
Function:
MortonOrder(Element e,BoooundingBox box,uint64 N )
$index \leftarrow 0$ ;
foreach level upto N do
    $index \leftarrow index << 3$ ; /* Concatenation with each level, 2 for two
        dimension                                                        */
    /* Set the octant using Morton order                                 */
    if e.centroidx > box.center.x then
        $index \leftarrow index + 1$
    end
    if e.centroidy > box.center.y then
        $index \leftarrow index + 2$
    end
    /* For 3D Case                                                       */
    if e.centroidz > box.center.z then
        $index \leftarrow index + 4$
    end
    /* Update th bounding box to the appropriate octant                  */
    foreach dimension in bounding box do
        if e.component > box.center.component then
            box.min.component = box.center.component
        else
            box.max.component = box.center.component
    end
end
```

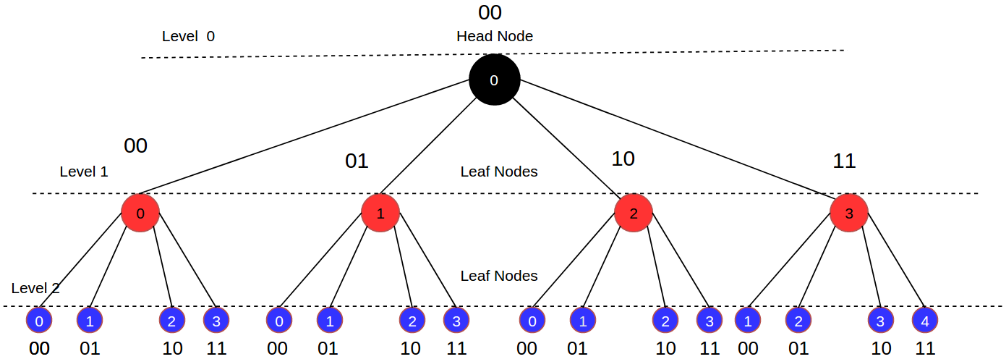


Figure 4.5: Quadtree illustration of Z-order Space Filling Curve ordering

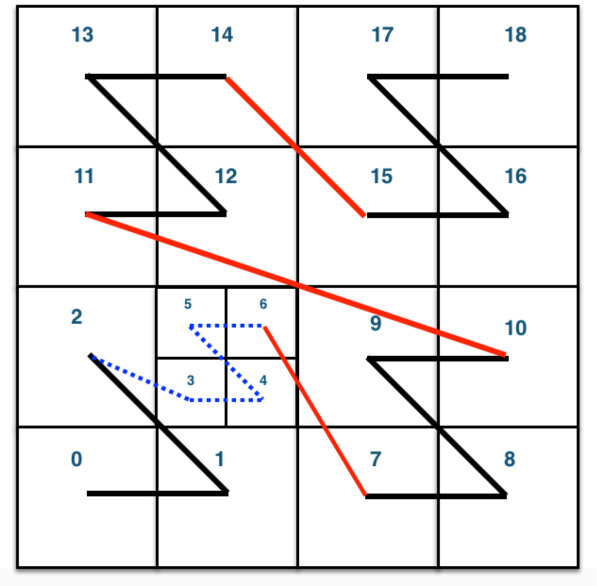


Figure 4.6: Construction of Morton-order Space Filling Curve for 2D adaptive grid

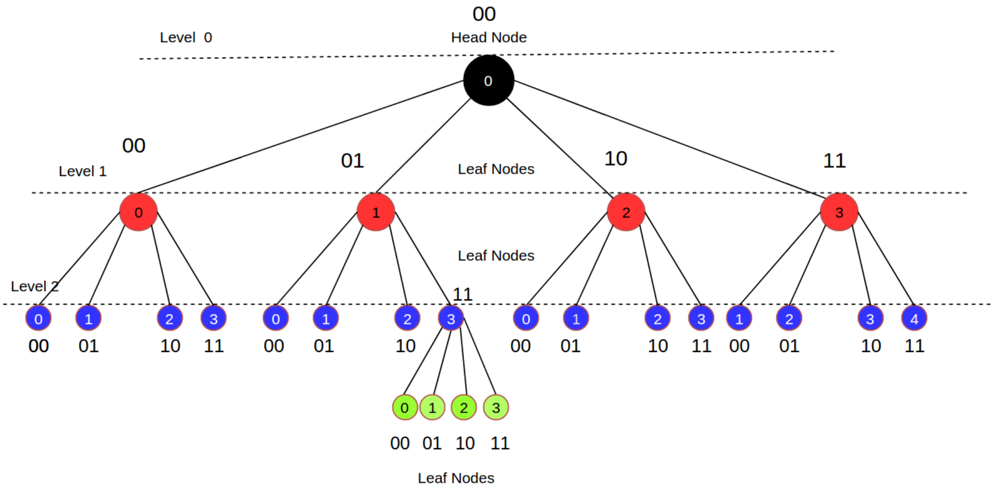


Figure 4.7: Quad-tree illustration of Morton-order Space Filling Curve with adaptive grid

4.2. Development of Parallel C++ Code for Space Filling Curves

The parallel code for Space Filling Curve generation has been developed in C++ during the period of the master thesis. The design strategies and the implementation are discussed in this section and useful results for one test case are shown to highlight efficiency and usability of the code. The parallel code enables computing Morton-order Space Filling Curve for very large meshes in a reasonable amount of time. The design flexibility of the code encourages the easy inclusion of other mathematical curves as well, for example, the Hilbert-order SFC [13] or the Peano Curves [13]. Figure 4.8 illustrates basic workflow of the ParSFC [5] application available online.

External Libraries used and Compilation

The parallel sort algorithm from Intel TBB [6] is used for parallel sorting operation in ParSFC application. Intel TBB is a thread building block library for parallel algorithms, data structures and task scheduling.

The ParSFC has been compiled with gcc version 5.4.0 and -std=c++11 -fopenmp -O3 as compiler flags and -tbb against the Intel TBB library.

Execution

Numactl is another library for thread pinning and local memory allocation. It is used for pinning threads to cores. The following command is used for execution

```
numactl –physcpubind=0-7 –localalloc ./<name-of-executable> <file-name-of-geometry>
```

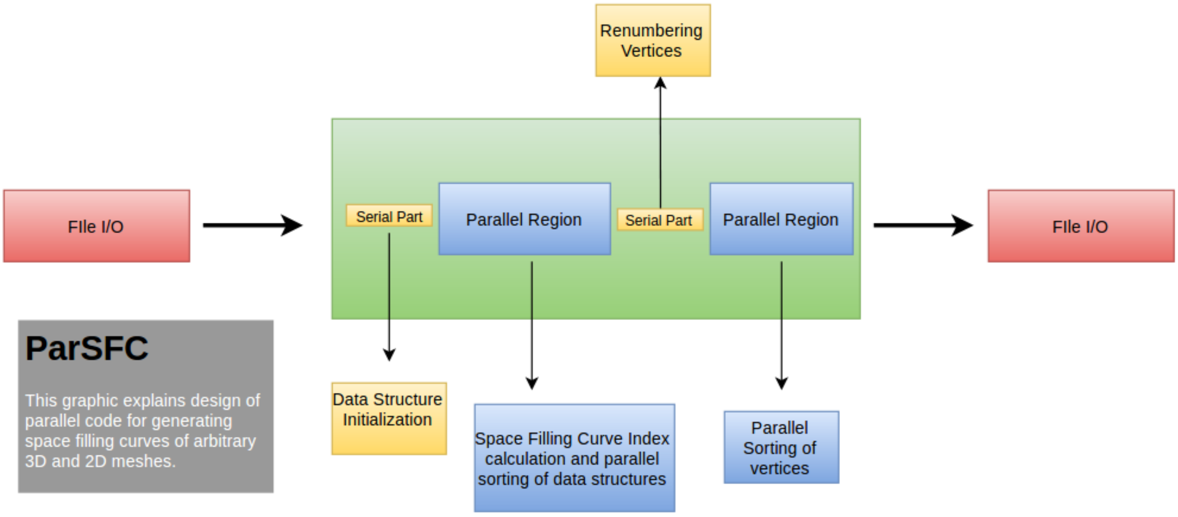

Renumbering Vertices
FIle I/O
Serial Part
Parallel Region
Serial Part
Parallel Region
FIle I/O
ParSFC
This graphic explains design of parallel code for generating space filling curves of arbitrary 3D and 2D meshes.
Data Structure Initialization
Space Filling Curve Index calculation and parallel sorting of data structures
Parallel Sorting of vertices

Figure 4.8: Pipeline of ParSFC C++ code

4.2.1. Parallel Performance Analysis of ParSFC application

The parallel performance of the SFC curve generation for recursion depth of 20 levels is studied for the test case shown in Figure **??** to understand scalability issue in generating Space Filling Curve. Table 4.1 shows the simulation parameters. Figures 4.10 and 4.11 are practical implications of Algorithms 1 and 2. Upon observing Figure 4.10 it is clear that Morton-order Space Filling Curve generation speeds up to a maximum of ≈3.2x on 8 core machine for the test case mesh containing ≈49 million vertexes. The not so impressive speed-up

[5] https://github.com/computingdolas/Parallel-Space-Filling-Curve
[6] https://www.threadingbuildingblocks.org/

is due the presence of a serial part which suppresses perfect scalability. The major obstruction to parallelize the serial part is maintaining global numbering of vertices which if updated by multiple-cores simultaneously will cause huge penalties due to the involvement of atomic updates.

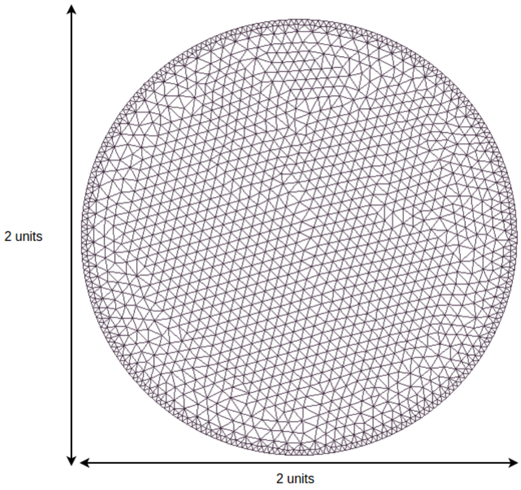


Figure 4.9: Test case

Table 4.1: Simulation parameters

Number of Triangles	100 million
Number of Vertices	49 million
Recursion Depth	20
Number of Cores	8

Figure 4.11 shows the scalability for different numbers of vertices on a 8 core machine. It is clearly visible that speed-up decreases linearly due to increase in the number of vertices. This is because of the presence of serial part which linearly grows as the number of vertices is increased. This problem can be solved with generating Space Filling Curve over virtual distributed memory computing environment but this is not explored in this work.

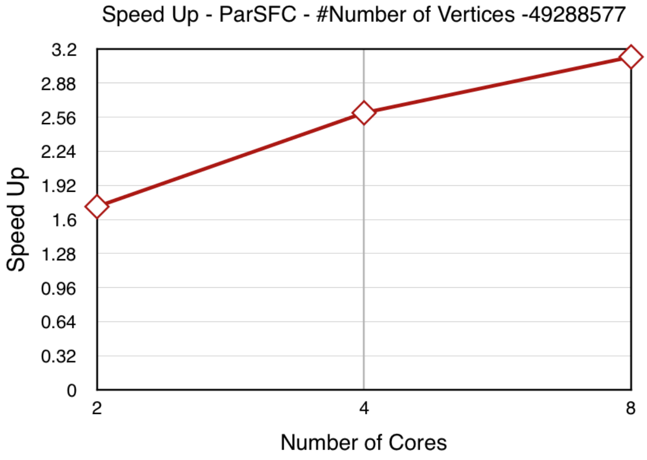


Figure 4.10: Speed of ParSFC C++ code for increasing number of cores.

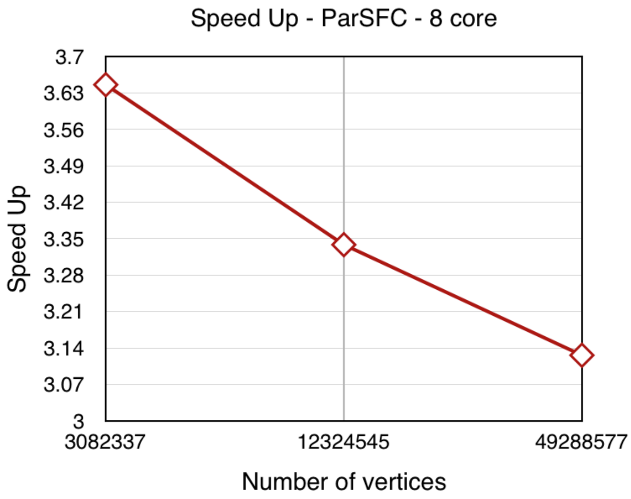


Figure 4.11: Speed-up of ParSFC C++ code for different number of vertices .

4.2.2. Relative Error

This section will explain the relative error in implicit octree implementation for Space Filling Curves. The relative error is defined in equation 4.4 as the fraction of the number of elements having the same SFC index over the total number of elements.

$$Relative\ Error(\%) = \frac{Number\ of\ Elements\ with\ same\ SFC\ Index\ x\ 100}{Total\ Number\ of\ Elements} \tag{4.4}$$

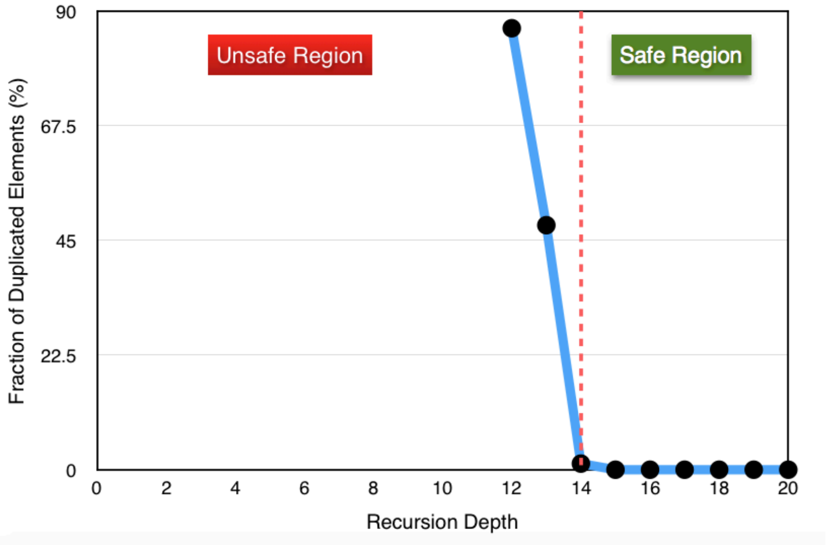


Figure 4.12: Fraction of duplicated elements in a mesh containing ≈98 million elements

Figure 4.12 shows the fraction of elements having same the SFC index with respect to varying recursion depths on a test mesh containing ≈98 million elements. Obviously a recursion depth below 14 is impractical and will lead to unpredictable results since the fraction of elements having the same SFC index increases exponentially leaving a major chunk of elements unordered. It is crucial to recognize operating region for a specific mesh in case of implicit octree implementation.

4.3. Visualizing Data traversal with Space Filling Curves

Visualising data traversal with Morton-order Space Filling Curve helps to understand its working principle and locality property. This section visualises Morton-order SFC dynamically by simulating serial and parallel data traversal on the 3D arbitrary mesh. Element data traversal involves a lot of indirect memory accesses, and therefore it is vulnerable to the inefficient use of memory subsystems. This type of visualisations helps to understand how space filling curve layout can improve performance.

4.3.1. Test Case and Details

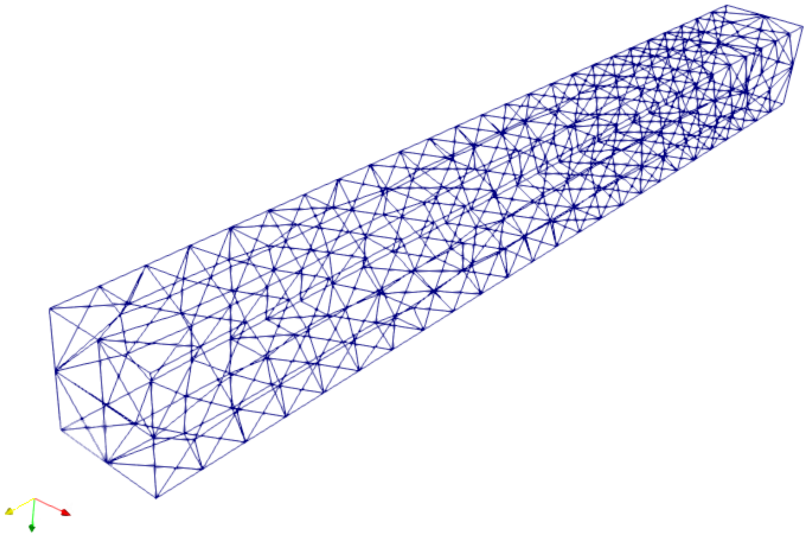


Figure 4.13: 3D test case for visualizing data traversal by Space Filling Curve

Table 4.2: Simulation details

Number of Elements	1471
Number of Vertices	2602
Simulation Post-processor	Paraview

Table 4.2 shows the properties of the test case mesh and details of the post-processing tool to create the visualization. The next pages show the simulation of data traversal on test case 3D geometry illustrated in Figure 4.13. The snapshots are set up within a gap of 300 elements traversal and captured both for the serial and parallel case with natural ordering and SFC ordering. They illustrate the order in which elements are visited in a for loop of a typical finite element code.

4.3.2. Analysis of Serial Data Traversal

The evolution or progression of colours in following figures typical represents data accessed.

Analysing Serial Data Traversal with Natural Ordering

The serial data traversal with natural ordering is shown in Figures 4.14 to 4.19. It is quite clear that data traversal does not follow a regular pattern. Spatial distribution of travelled elements appear to be scattered and random. The major performance impact is an inefficient utilisation of cache hierarchy due to increased capacity and conflict misses. It is well known that data from main memory to CPU travels in cache blocks. Utilising these cache blocks can give impetus to code performance. However, it is apparent that data traversal as shown in Figures 4.14 to 4.19 cannot encourage cache reuse and leads to repeated number of redundant DRAM memory accesses degrading performance.

Looking from the perspective of modern bandwidth oriented machine, this type of data access pattern can utilise the underlying hardware as it will always be busy doing redundant memory accesses and inefficient computation. Energy consumption both for CPU-core and DRAM has been studied for similar data traversal pattern in the previous chapter. It is clear that this type of data traversal will lead to increased energy consumption for CPU and memory module as well.

Analysing Serial Data Traversal with SFC Ordering

The serial data traversal with SFC ordering is shown in Figures 4.20 to4.25. It is quite clear that data traversal follows a blocking pattern and spatial distribution is clustered in each snapshot from Figures 4.20 to 4.25. The major performance impact is an efficient utilisation of cache hierarchy and cache reuse. This type of data traversal supports spatial and temporal locality in a way that redundant bus cycles to DRAM can be reduced to a large extent when accessing vertex data in typical for the loop.

Serial Data Traversal with Natural Ordering

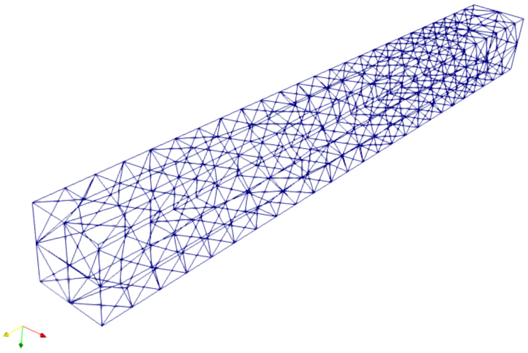


Figure 4.14: 0 Elements travelled

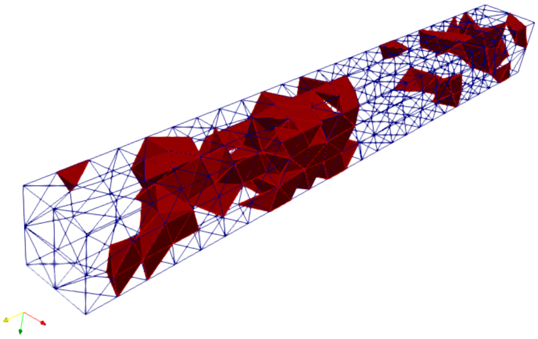


Figure 4.15: 300 Elements travelled

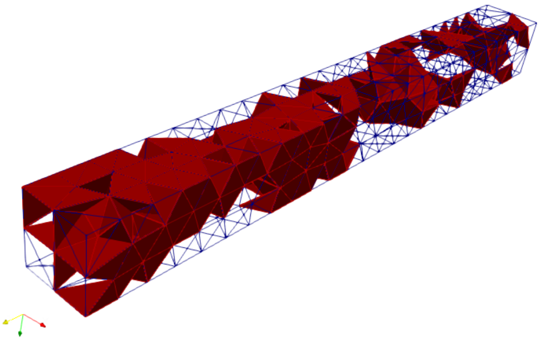


Figure 4.16: 600 Elements travelled

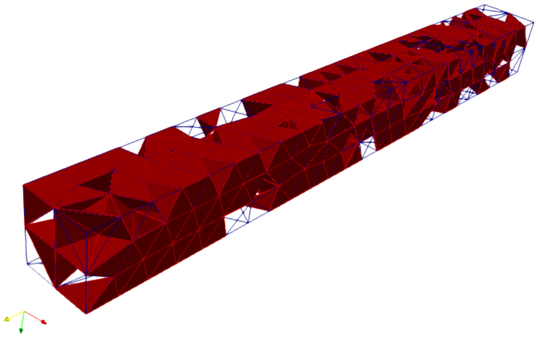


Figure 4.17: 900 Elements travelled

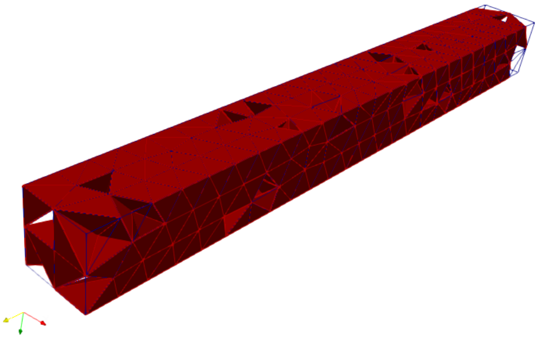


Figure 4.18: 1200 Elements travelled

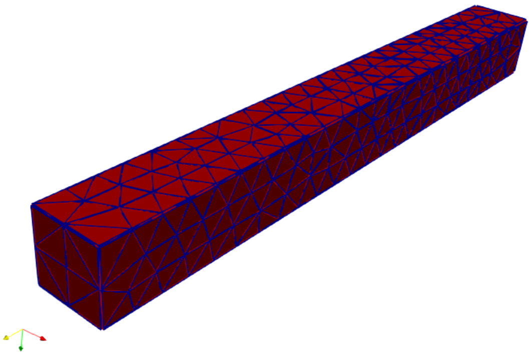


Figure 4.19: 1500 Elements travelled

Serial Data Traversal with SFC Ordering

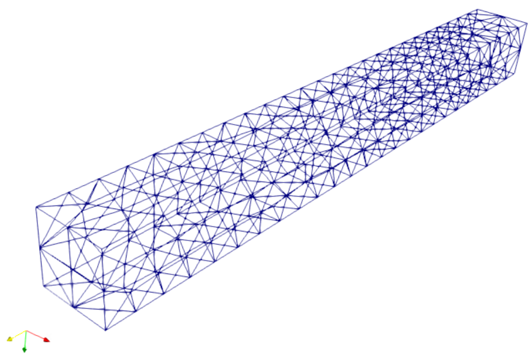


Figure 4.20: 0 Elements travelled

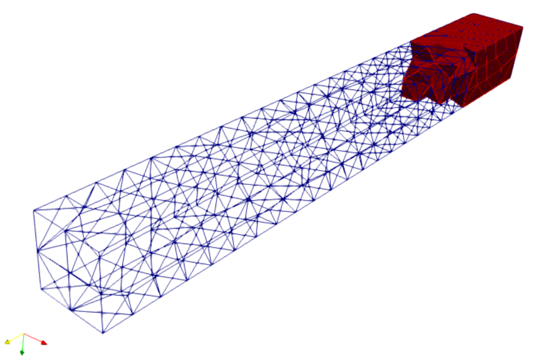


Figure 4.21: 300 Elements travelled

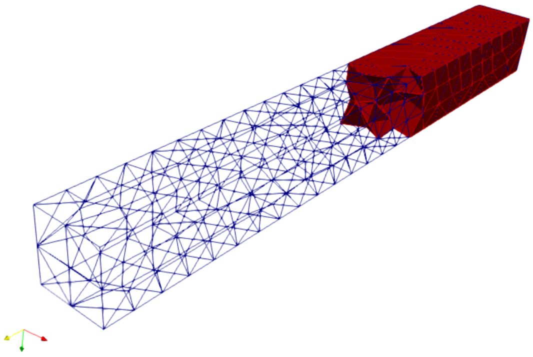


Figure 4.22: 600 Elements travelled

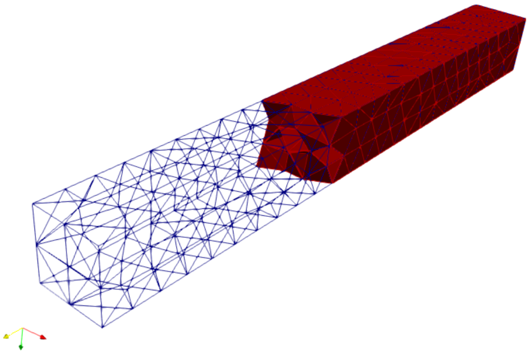


Figure 4.23: 900 Elements travelled

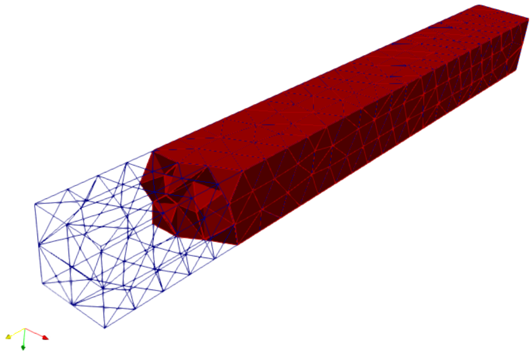


Figure 4.24: 1200 Elements travelled

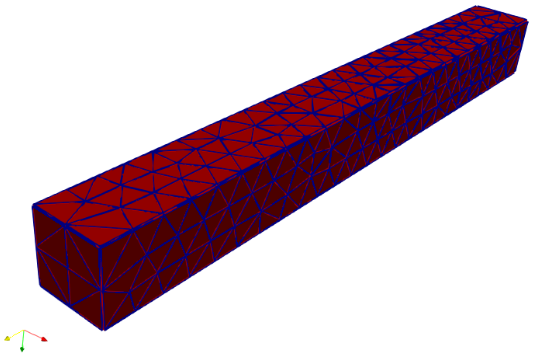


Figure 4.25: 1500 Elements travelled

4.3.3. Analysis of Parallel Data Traversal

Parallel data traversal is complicated as it engages multiple processing cores to operate on data. Typically parallel loops involve synchronisation points, and in turn, these synchronisation points commonly require atomic read or write instructions to update or access data accurately. The number of synchronisation points heavily affects the performance since atomic read or write allows the only single core to update or access data at a time resulting in multiple cores waiting in queue for the same operation.

Synchronisation points depend on shared-memory addresses among multiple processing cores which in turn depend on how data is divided among processing cores or depends on the type of data layout multiple cores are operating. In the next pages, parallel data traversal with four cores represented by different colours is shown to emphasise data traversal in parallel with multiple cores. The evolution or progression of colours typical presents data access with individual cores. The appearance of different colours at the same vertex or edge represent shared memory address requiring atomic instruction to update or access correctly.

Analysing Parallel Data Traversal with Natural Ordering

The Figures 4.26 to 4.31 shows parallel data traversal with natural ordering. It is clear that this type of data access pattern involves many synchronisation points since shared-memory addresses among processing cores are huge and therefore atomic read or write instructions have to be used to correctly update or access data. The performance degradation is imminent due to the problem of false sharing and cache coherency, bad partitioning of data among processing cores and inefficient use of memory subsystem by the individual core.

Analysing Parallel Data Traversal with SFC Ordering

Parallel data traversal with SFC ordering is shown in Figures 4.32 to 4.37. Cores travel data in blocks requiring only few synchronisation points which amounts to less usage of atomic read and write instructions. The cache hierarchy can be utilised to a great extent, and the problem of false sharing and cache coherency also reduces significantly. From the perspective of modern-bandwidth oriented NUMA machines, this type of data traversal and data placement is favourable since it supports efficient data partitioning among multiple cores of even different CPU socket as well. This blocking pattern typically enhances cache aware computing.

Parallel Data Traversal by 4 Cores with Natural Ordering

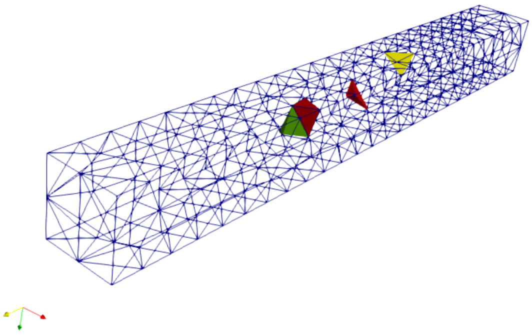


Figure 4.26: 0 Elements travelled

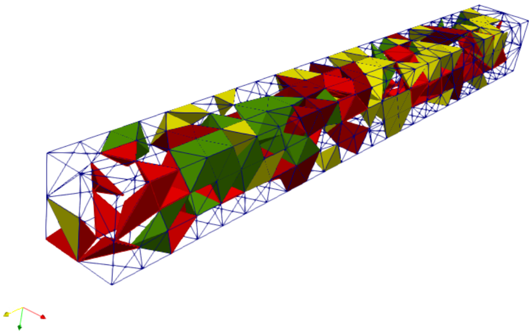


Figure 4.27: 320 Elements travelled

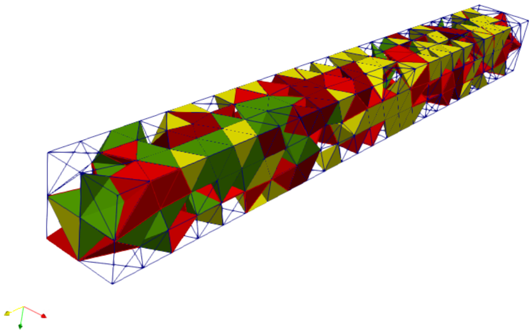


Figure 4.28: 640 Elements travelled

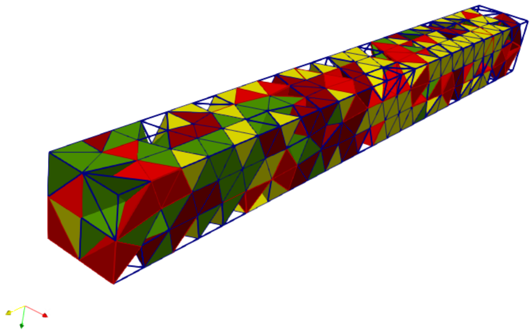


Figure 4.29: 960 Elements travelled

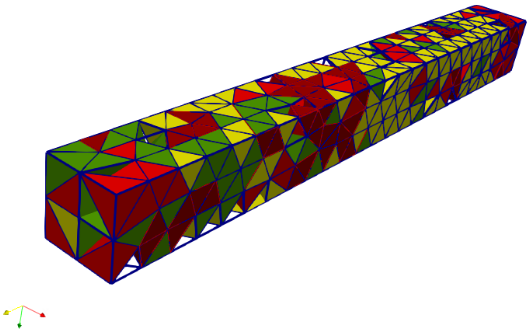


Figure 4.30: 1280 Elements travelled

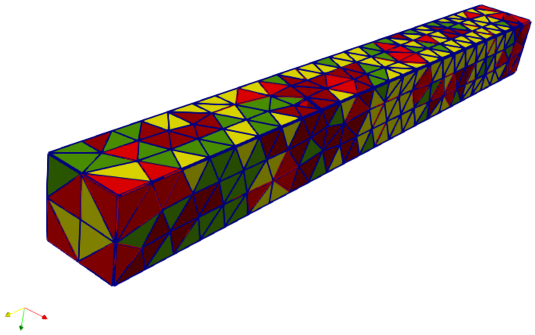


Figure 4.31: 1500 Elements travelled

Parallel Data Traversal by 4 Cores with SFC Ordering

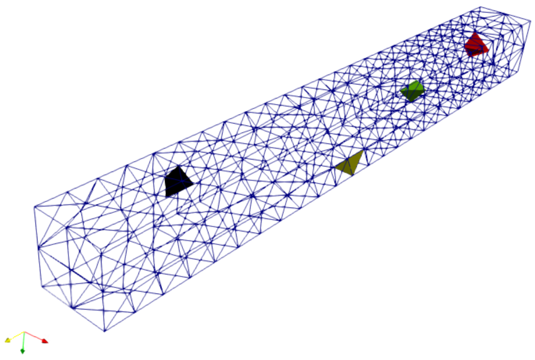


Figure 4.32: 0 Elements travelled

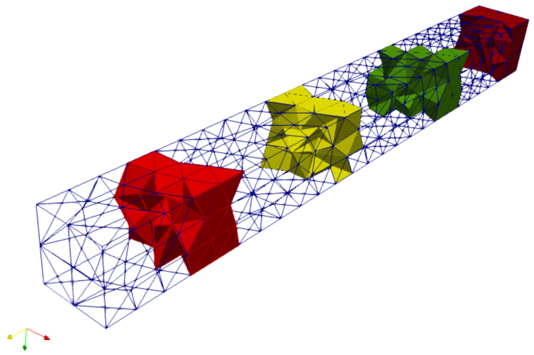


Figure 4.33: 320 Elements travelled

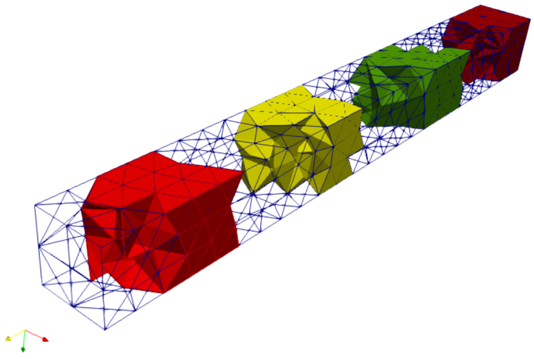


Figure 4.34: 640 Elements travelled

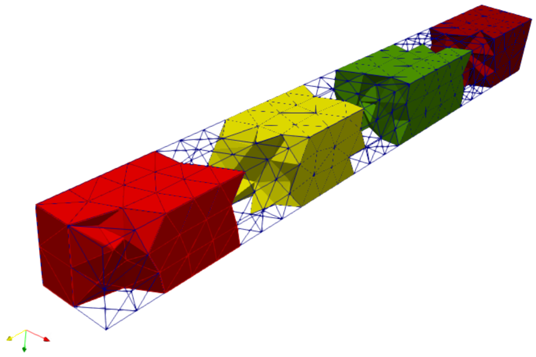


Figure 4.35: 960 Elements travelled

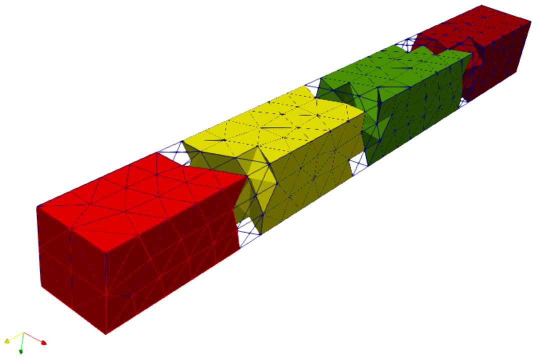


Figure 4.36: 1280 Elements travelled

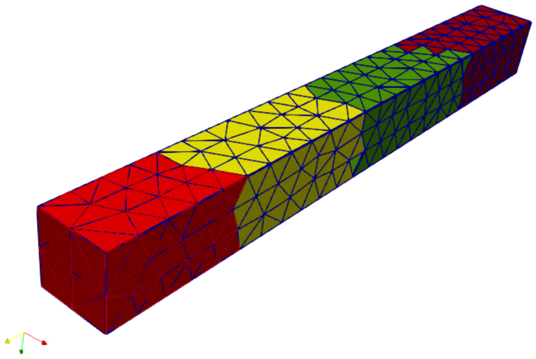


Figure 4.37: 1500 Elements travelled

5

Analysis with Space Filling Curve on Finite Element Solver

5.1. Introduction

This chapter investigates a Finite Element solver for optical beam waveguide governed by the Helmholtz equation. It contains an in depth analysis of the impact of Space Filling Curve reordering on the computational performance of matrix assembly and linear system solver. This chapter also compares the impact of Morton-order Space Filling Curve (SFC) reordering of elements and vertices on a bandwidth of Finite Element matrix and performance of its LU factorization. The chapter concludes with advantages and disadvantages of using Morton-order SFC.

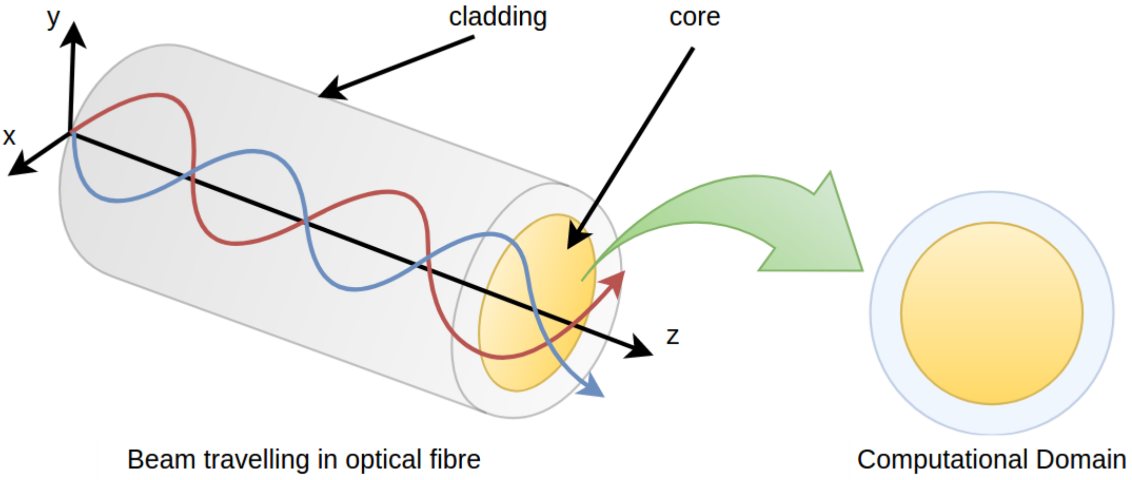


Figure 5.1: Beam travelling in optical waveguide

Figure 5.1 illustrates the propagation of beam in an optical waveguide which is mathematically described by lowest order eigenmode and eigenvalue of the associated eigenvalue problem stated by Helmholtz equation. The overall approach involves the discretization of governing equation with linear finite elements on the test mesh shown in Figure 5.2 and finally solving the linear system of equations for the lowest eigenvalue and the associated eigenvector by inverse power iteration. In this particular analysis performance impact of

Space Filling Curve reordering of mesh elements and vertices on energy efficiency and CPU time is the focus of study.

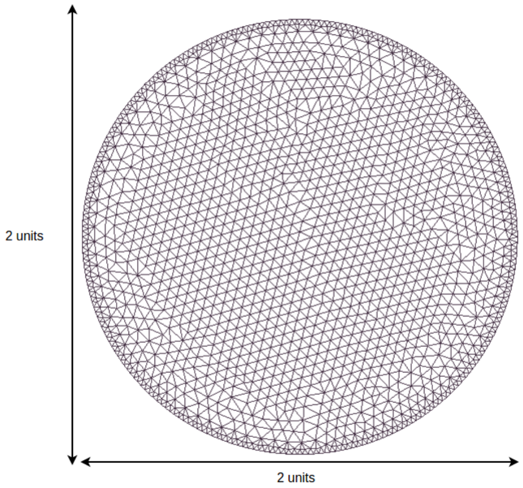


Figure 5.2: Computational domain

5.1.1. The Governing Equation

The information transport within a beam waveguide is described by the Helmholtz equation. The general three-dimensional problem can be reduced to two dimensions by using a time-harmonic approximation of the wave in propagation direction. The resulting equation is as follows :

$$-\Delta u(x,y) - k(x,y)^2 u(x,y) = f(x,y) \;\; in \;\; \Omega \tag{5.1}$$

with Neumann boundary condition

$$\frac{\partial u}{\partial \vec{n}}|_{\partial\Omega} = 0 \tag{5.2}$$

The variational formulation after integration by parts and satisfying (5.2) reads: Find $u(x,y) \in V = \{u \in H^1(\Omega)\}$ such that

$$\int_\Omega \nabla u(x,y)\nabla v(x,y) - k(x,y)^2 u(x,y)v(x,y)\mathbf{d}(x,y) = \int_\Omega f(x,y)v(x,y)\mathbf{d}(x,y) \tag{5.3}$$

for all $v(x,y) \in V$. For this particular problem, we want to compute the beam properties in the waveguide which is usually described by the lowest order eigenmode of the eigenvalue problem.

$$-\Delta u(x,y) - k(x,y)^2 u(x,y) = \lambda u(x,y) \tag{5.4}$$

Variable coefficient $k(x,y)$ depends on the refractive index of the material and the wavelength of the propagating beam. Since refractive index of the core of the waveguide is higher than the refractive index of the cladding [1] , the gradient profile is chosen to be

$$k(x,y)^2 = (100.05)e^{-50(x^2+y^2)} - 100 \tag{5.5}$$

Figure 5.3 plots the above function. The discrete form of the eigenvalue problem (5.4) can be derived by its weak formulation as :

$$\mathbf{A}_h\mathbf{u}_h = \lambda\mathbf{M}_h\mathbf{u}_h \tag{5.6}$$

[1] http://www.ciscopress.com/articles/article.asp?p=170740&seqNum=3

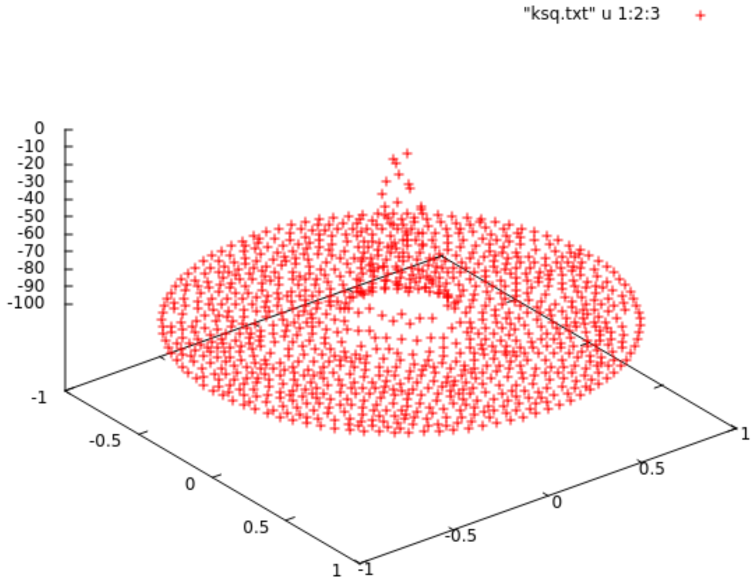


Figure 5.3: Plotting function $k(x,y)^2$

where stiffness matrix $\mathbf{A}_h$ and mass matrix $\mathbf{M}_h$ are formulated as follows :=

$$\mathbf{A_h} = \int_\Omega \nabla u(x,y) \nabla v(x,y) - k(x,y)^2 u(x,y) v(x,y) \mathbf{d}(x,y) \tag{5.7}$$

$$\mathbf{M_h} = \int_\Omega u_h(x,y) v_h(x,y) d(x,y) \tag{5.8}$$

and u_h and v_h belong to our finite element space. Inverse power iteration described in Algorithm 3 is used to solve equation 5.6 to compute the lowest eigenvalue λ and eigenvector $\mathbf{u}_h$ simultaneously.

Algorithm 3: Inverse Power Iteration

```
Data: Initialise $u_h$
Result: $u_h$ and $\lambda$
initialization of solver iteration;
while | $\frac{\lambda - \lambda_{old}}{\lambda_{old}}$ |> $10^{-8}$ do
    do
        Initialise old $\lambda$
        $\lambda_{old} = \lambda$
        Find out right hand side
        $f_h = M_h . u_h$
        Solve using CG :
        $A_h . u_h = f_h$
        $u_h = \frac{u_h}{\|u_h\|^2}$
        $\lambda = \frac{u_h^T A_h u_h}{u_h^T M_h u_h}$
    while | $\frac{\lambda - \lambda_{old}}{\lambda_{old}}$ |> $10^{-8}$;
```

The crucial point here is to note that matrix assembly using linear basis hat function is a bandwidth limited kernel and a bandwidth limited kernel scales or achieves performance upon utilizing memory subsystem efficiently as seen in Chapter 3. Space Filling Curves have the capability to reorder mesh data to preserve spatial and temporal locality and generate mesh layouts which can use memory hierarchy up to highest level of the cache as effectively. This investigation is aims at understanding the performance impact of data locality and cache oblivious data access pattern on bandwidth limited kernel.

The investigation on inverse power iteration is to understand the effect of data locality and data access pattern on compute intensive matrix-vector multiplication. This whole analysis will also compare energy consumption in different scenarios. This analysis should confirm that Space Filling Curves can act as a default choice of data traversal in finite element analysis and associated numerical methods.

5.1.2. Review on various solvers with Space Filling Curve

There have been numerous occasions, where researchers have tried solving Finite Element problems with Space Filling curve reordering and few of them are summarised below.

Hibert-order Space Filling Curves have been used to reorder mesh to improve cache utilisation for unstructured Finite Element meshes in [7]. The authors note that Space Filling Curves help to enumerate the mesh elements in a way to ensure the reduction of wastage in time for lock guarding mechanism in parallel stiffness matrix assembly. They have also observed the reduction in time and number of iterations required to solve the linear system of equations by the preconditioned conjugate gradient method. They have also observed that in certain cases, data layout generated by SFC reordering has very low number of shared vertices as compared to RCM when performing parallel matrix assembly. Overall they have observed 20 % reduction in time due to Space Filling Curve reordering on 4 socket 48 core multiprocessor system.

Automated mesh adaptation is studied in [2] which relies heavily on data localisation. The authors renumber the mesh items using Hilbert-order Space Filling Curve to preserve spatial and temporal locality and they have observed SFC to be versatile and useful. The authors have considered the wide variety of computational problems and have concluded that Space Filling Curve provides cheap and easy parallelization of problems involving mesh adaptation and also leads to significant drop in overall cache misses.

In [15] the authors present Morton-order Space Filling Curve as an technique to compute cache oblivious data layout of unstructured mesh and compare its performance with optimal cache-oblivious mesh layout (COML) for surface and volume data. They compare different algorithms documenting strengths and weakness of SFC. They observe that generating Space Filling Curve can be an order of magnitude faster than other methods while maintaining mesh layout close to optimal. They conclude that Space Filling Curve has many advantages over its competitive alternatives.

In [11] the authors investigate the impact of matrix reordering schemes on preconditioned conjugate gradient solver. They have observed that SFC provides superior performance compared to (Cuthill-McKee) RCM or METIS in some cases. They also concluded that ensuring cache reuse may be significantly more powerful than reducing communication overhead in the parallel environment. Test case studies have shown that that SFC is twice as fast as RCM for performing sparse matrix-vector multiplication.

5.2. Introduction to Finite Element Solver

This section outlines the features of the developed code WaveGuide [2] and explains important code blocks relevant to understanding assembly and solver routine. Figure 5.6 illustrates code design and it's working. It is clear from the diagram that program starts with reading the mesh file, stores the mesh data into the appropriate data structure, performs grid refinement if required, assembles global stiffness matrix and solves system of linear equations.

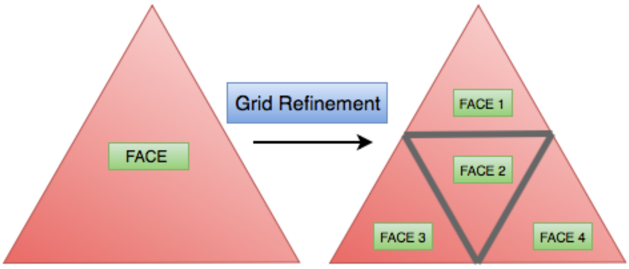


Figure 5.4: Grid refinement procedure

The data structure used for storing faces/ triangular-elements and vertices is shown in Figure 5.7. It is designed to ensure striding access to the processing core. The data transfer between main memory (DRAM) and CPU is completed in cache lines or cache blocks as shown in Figure 5.8. This type of data structure helps to calibrate performance impact with and without cache friendly data access. The process of grid refinement is shown in Figure 5.4. During grid refinement each face is divided into four equal smaller faces and the new face indices and the associated new vertex indices are appended to the end of the data structure.

Stiffness matrix A_h and mass matrix M_h as described in equation (5.6) are assembled as shown in Algorithm 4 in the data structure shown in figure 5.5. Since most of the time, prior knowledge about sparsity pattern is unavailable, this type matrix storage scheme provides an efficient way of storing sparse matrices because adding and updating matrix entries is less complicated and virtually involves no searching.

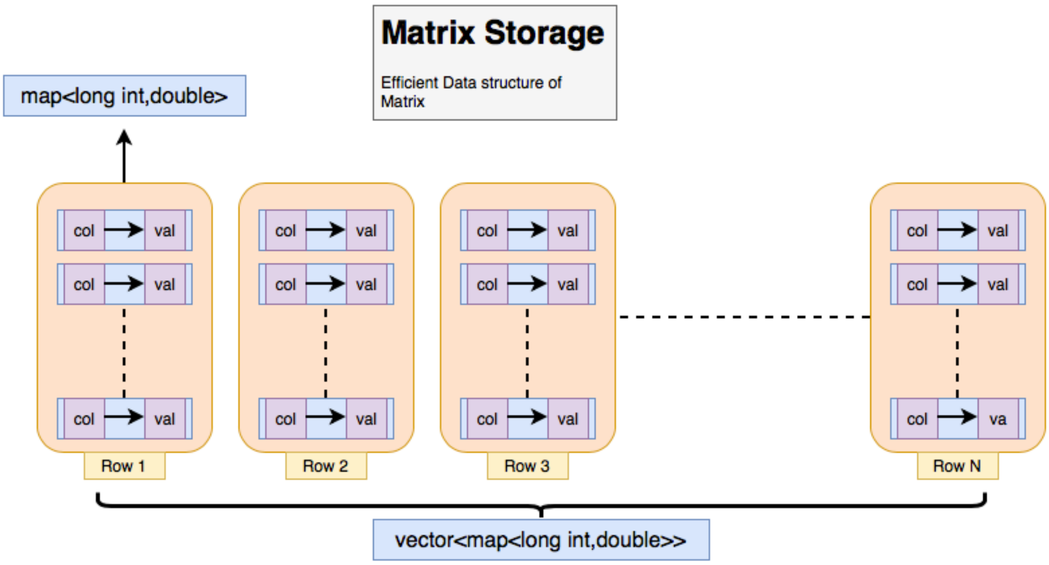


Figure 5.5: Efficient matrix data storage layout

[2] https://github.com/computingdolas/ParSFC

With the completion of assembly operation, the matrix data structure is converted into compressed row storage (CRS)[3] format so as to efficiently execute sparse matrix-vector multiplication.

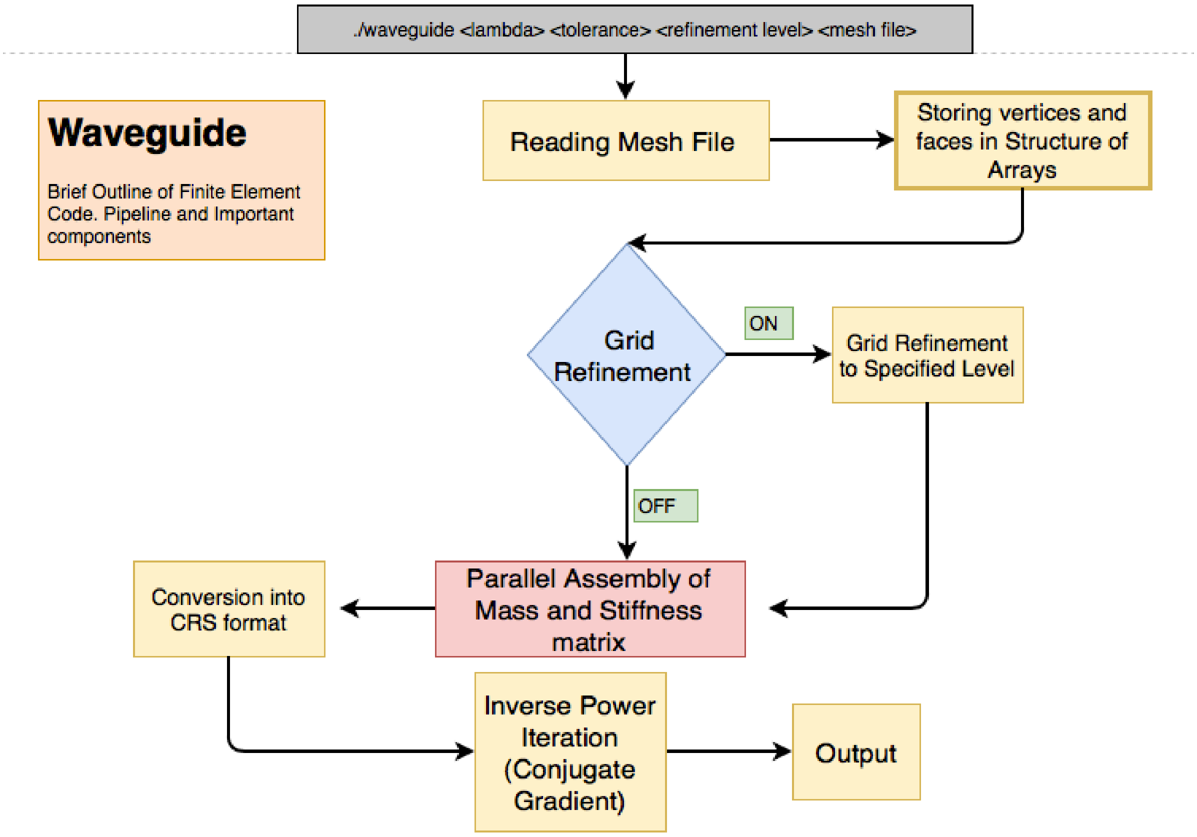

./waveguide <lambda> <tolerance> <refinement level> <mesh file>
Waveguide
Brief Outline of Finite Element Code. Pipeline and Important components
Reading Mesh File
Storing vertices and faces in Structure of Arrays
Grid Refinement
ON
Grid Refinement to Specified Level
OFF
Parallel Assembly of Mass and Stiffness matrix
Conversion into CRS format
Inverse Power Iteration (Conjugate Gradient)
Output

Figure 5.6: Flow of Waveguide C++ code

Algorithm for Matrix Assembly

The procedure for serial matrix assembly is shown in Algorithm 4.

Algorithm 4: Algorithm for matrix assembly

```
Input   : Mesh file
foreach Element in the mesh do
    Access vertices of the element
    Calculate local stiffness matrix
    Update entries in the Global stiffness matrix
end
```

[3] http://netlib.org/linalg/html_templates/node91.html

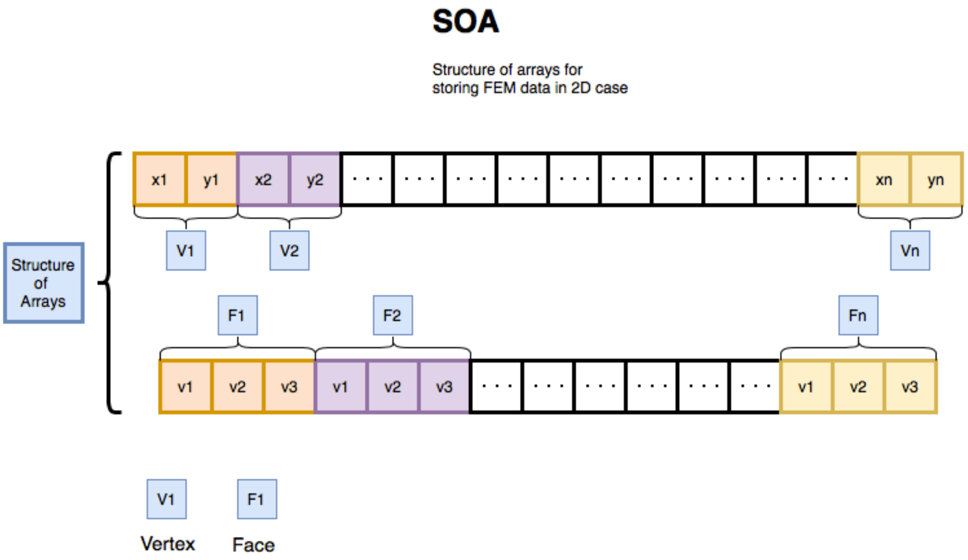


Figure 5.7: Structure of arrays for storing element and vertices data efficiently

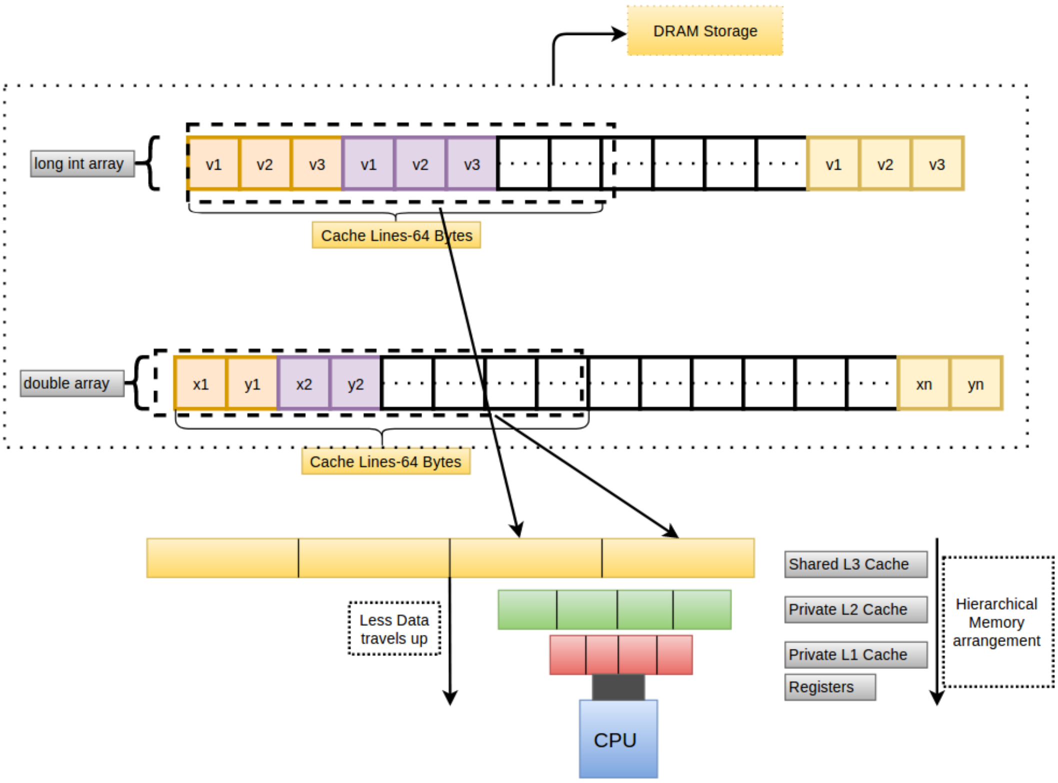


Figure 5.8: Memory access pattern of FE data in processor memory hierarchy

5.3. Performance Analysis and Methodology

The NUMA-aware memory allocation by first touch principle is ensured to designate data local to processing cores, pinning of threads is enforced to avoid thread migration and hyperthreading is turned off. The numactl library and Perf tool are used for thread pinning, memory allocation, profiling and measurements. This section will not discuss in depth about solution of the eigenvalue problem discussed above but about the methodology and in-depth performance issues with respect to computational time and energy consumption.

Simulation Details

This particular analysis is carried on mesh shown in Figure 5.2 containing ≈700,000 vertices and ≈1500,000 triangular elements. The analysis is repeated 500 times for serial matrix assembly to normalize unnecessary overhead penalties. Simulation parameters are shown in Table 5.1.

Table 5.1: Simulation parameters

Refinement Level	0
Relative Tolerance	10^{-8}
Number of Repetitions(Matrix Assembly)	500
Number of Repetitions(Inverse Power Iteration)	1
Filename(without SFC)	unit_cicrle_refine3.txt
Filename(with SFC)	unit_cicrle_sfc.txt

Compilation and Execution

The FEM code is compiled with the Intel C++ Compiler version 17.0.4 with -c -std=c++11 -O3 -Wall -pedantic compiler flags. The program can be executed with following command :=

```
./waveguide 0.05 τ refinement-level <filename>
```

Different Hardware Scenarios and their Significance

This analysis is carried out in three different scenarios which differ with respect to the significance of data placement, data access and data traversal. The three scenarios are summarized below maintaining colour code similar to results explained in section 5.3 are as follows :

1. **Scenario A** - **Remote Memory Access without SFC** := In this scenario remote memory access is investigated without any reordering. This artificial remote memory access is investigated to understand and analyse performance impact and energy consumption due to non-local data placement to the processing cores.

2. **Scenario B** - **Local Memory Access without SFC** := In this scenario data is allocated local to the processing cores but not reordered to benefit memory subsystem or memory hierarchy. This scenario helps to understand performance impact due to inefficient and redundant data access pattern on bandwidth-oriented hardware.

3. **Scenario C** - **Local Memory Access with SFC** := In this scenario, vertices and elements are reordered according to Space Filling Curve which can use memory hierarchy efficiently. This analysis helps to understand the importance of access pattern with cache-oblivious data layout on computational time and energy consumption.

5.4. Interpreting Analysis

Comprehensive performance analysis of matrix assembly and inverse power iteration is carried out in this section to understand impact of data locality and data access pattern on three principle performance parameters as follows :

1. CPU time and energy efficiency.
2. Efficient utilization of cache hierarchy.
3. Effective use of memory subsystems.

5.4.1. Interpreting Analysis on Matrix Assembly

Performance analysis of matrix assembly is shown in Figures 5.9 to 5.14. Impact of data locality and data access pattern on CPU time and energy efficiency can be understood from Figures 5.9 and 5.10 respectively. Looking at impact of data locality, scenario A increases the computational time by ≈11 % and energy consumption by ≈8 % compared to scenario B due to fetching of expensive remote memory.

The data access pattern with Morton-order SFC layout reduces both computational time and energy consumption by ≈10 % and ≈8 % respectively as seen by comparing scenarios B and C. The main reason for performance improvement is efficient utilisation of cache hierarchy and memory module. There is ≈68 % drop in LLC misses, ≈11 % drop in L1 cache misses and ≈18 % drop in bus cycles.

5.4.2. Interpreting Impact on Inverse Power Iteration

This section explores the performance impact of Morton-order Space Filling Curve reordering on the inverse power iteration. It is the inherent property of Space Filling Curves to maintain spatial locality and therefore for a sparse symmetric finite element matrix it can potentially reduce bandwidth to a great extent speeding up matrix-vector multiplication operation. This section starts with the description of the solution of the eigenvalue problem, followed by highlighting ways to accelerate sparse matrix-vector multiplication, and finally discusses the performance impact of data access pattern and data locality.

Table 5.2: Solution details

Number of Elements	1539968
Number of vertexes	771185
Solution(Eigenvalue)	88.31383493790854
Relative tolerance	1e-8
Number of Iterations to converge	48

The solution details are given in Table 5.2. The smallest eigenvalue is ≈88.3. The inverse power iteration requires 48 iterations to converge with relative tolerance of 10^{-8}. A significant part of the overall computation in Algorithm 3 is spent in spmv operations. The Figure 5.15 illustrates this process. The blue blocks represent entries in one matrix row, and red blocks represent entries in one column vector. Each row entry is multiplied to its corresponding column entry. The left part of Figure 5.15 shows sparse matrix vector multiplication with scattered randomly distributed entries which inhibit efficient utilisation of cache hierarchies due to non-unit stride access pattern, whereas on the other hand clustered entries as depicted in the right-hand side of Figure 5.15 encourages cache blocking leading to improved performance.

Table 5.3: Number of cores used

Number of Cores	1
Processor	Intel Xeon 2687w v2

Performance analysis of inverse power iteration is shown in Figures 5.16 to 5.21. Looking from the perspective of data locality, scenario A increases the computational time and CPU

Serial Matrix Assembly - Performance Analysis

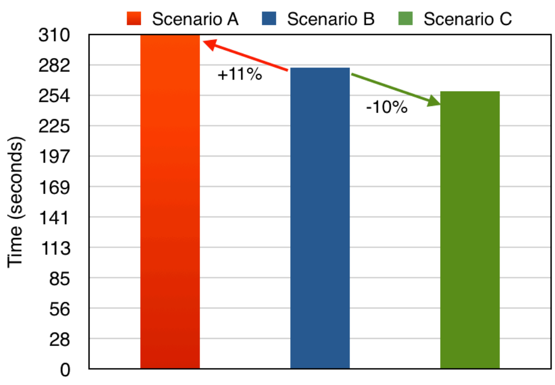


Figure 5.9: Total CPU time

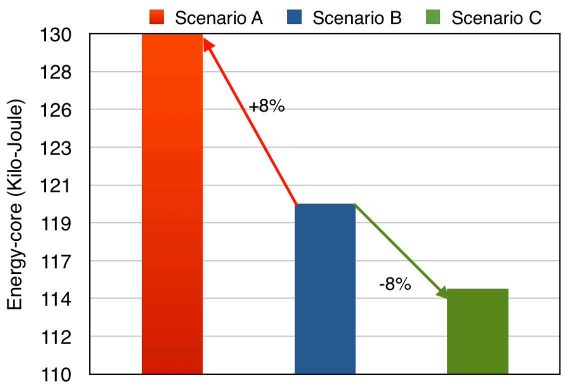


Figure 5.10: Energy-core

CPU time and Energy Efficiency

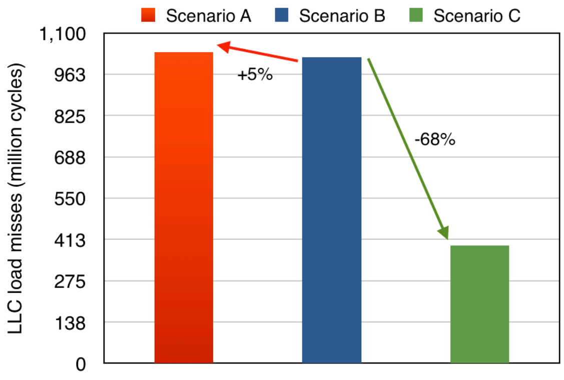


Figure 5.11: LLC load misses

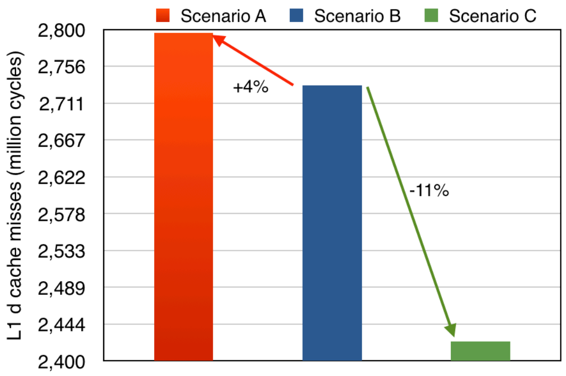


Figure 5.12: L1 d cache misses

Utilization of Cache Hierarchy

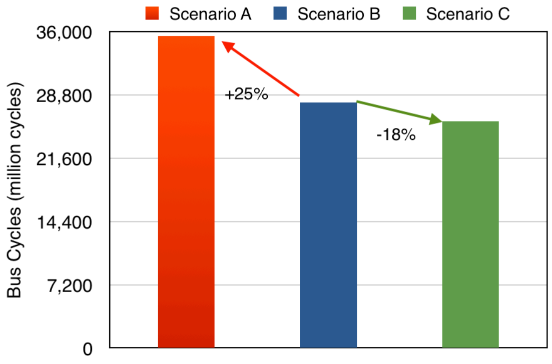


Figure 5.13: Bus cycles

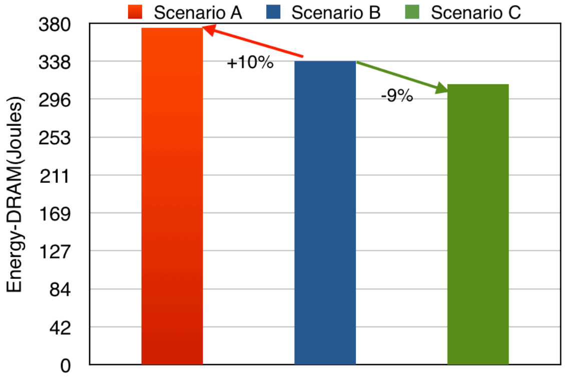


Figure 5.14: Energy-DRAM

Use of Memory Subsystem

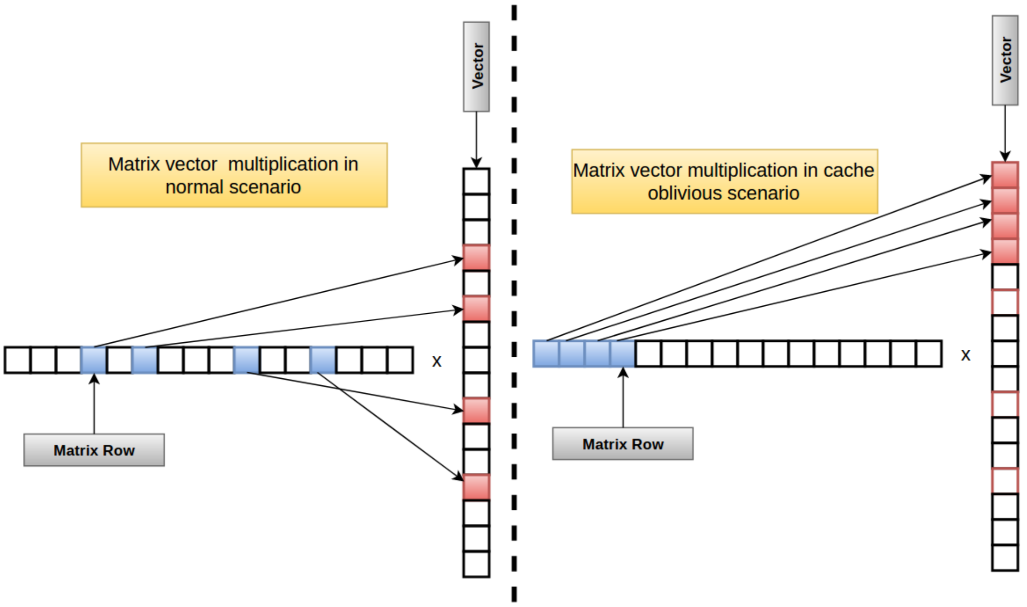

Vector
Matrix vector multiplication in normal scenario
x
Matrix Row
Vector
Matrix vector multiplication in cache oblivious scenario
x
Matrix Row

Figure 5.15: Comparison of normal and cache oblivious matrix-vector multiplication

energy consumption by ≈47 % and ≈50 % respectively compared to scenario B which proves that data locality is one major factor for compute intensive kernels. There is ≈66 % increase in DRAM traffic and ≈46 % increase in DRAM-energy consumption, signalling that fetching data from another CPU not only deteriorates the performance but also drastically increases DRAM traffic and DRAM-energy consumption.

Data traversal pattern has significant performance impact on algorithm dominated by the spmv operation. Scenario C improves computational time and core-energy consumption by ≈33 % and ≈25 % respectively. The main reason behind this substantial development is that the Morton-order Space Filling Curve decreases matrix bandwidth by large factor enabling efficient utilisation of cache hierarchies. There is ≈44 % drop in Last Level Cache misses resulting in ≈28 % drop in bus cycles and ≈40 % drop in DRAM energy consumption.

Inverse Power Iteration - Performance Analysis

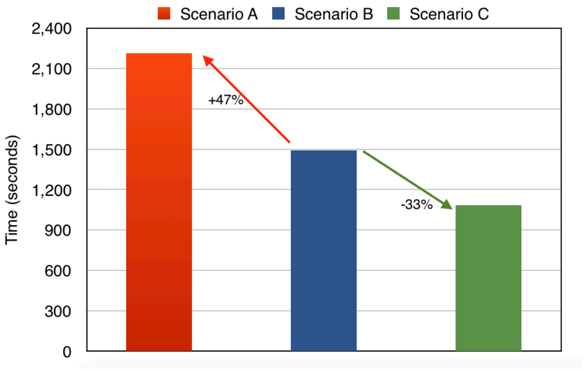


Figure 5.16: Total CPU time

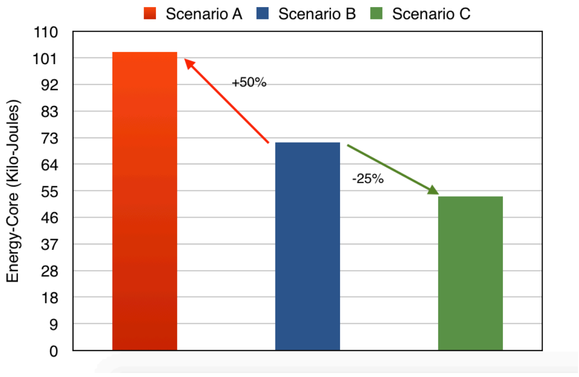


Figure 5.17: Energy-Core

CPU time and Energy Efficiency

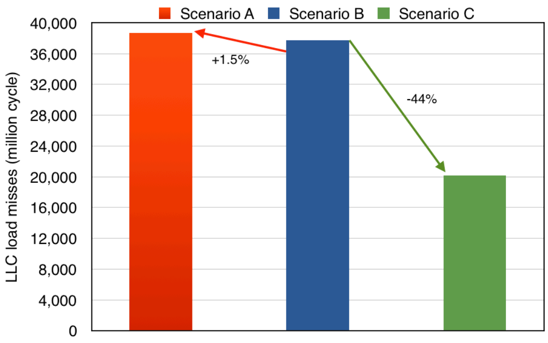


Figure 5.18: LLC load misses

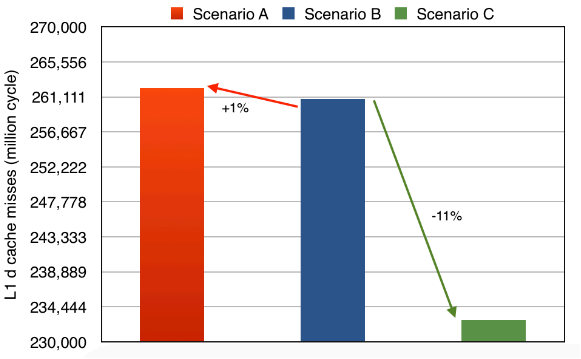


Figure 5.19: L1 d cache misses

Utilization of Cache Hierarchy

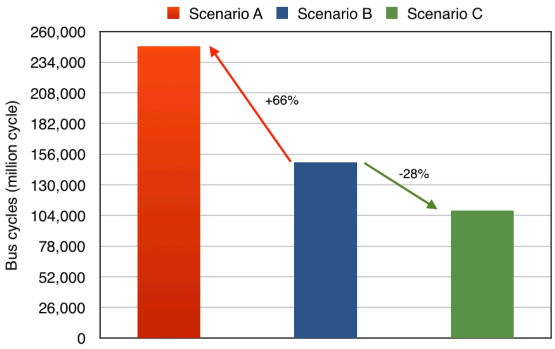


Figure 5.20: Bus Cycles

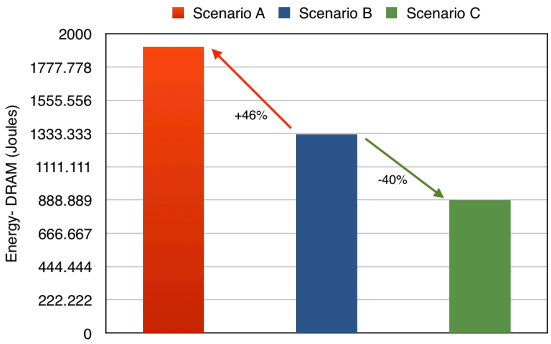


Figure 5.21: Energy DRAM

Use of Memory Subsystem

5.5. Matrix Structure Analysis

This section focuses on the impact of Space Filling Curve reordering on the structure of the global finite element matrix assembled with piecewise linear basis hat functions. Space filling curve reordering on the test case mesh containing ≈12000 vertexes and ≈24000 triangular elements will be analysed, and the results will be compared with other robust graph-based matrix reordering techniques.

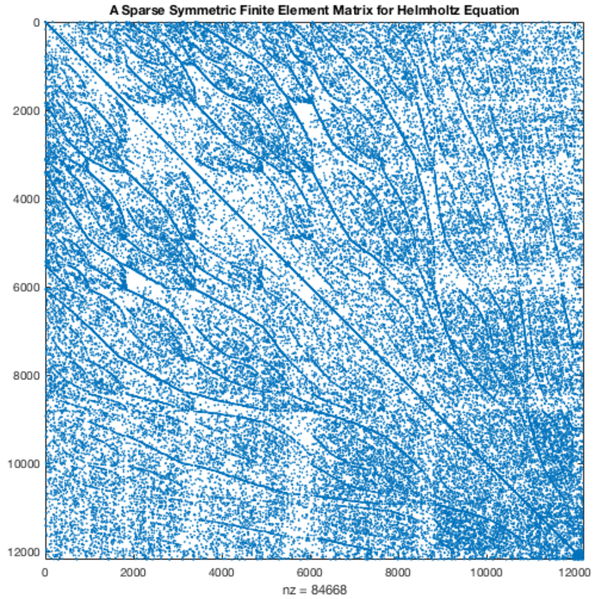


Figure 5.22: A sparse symmetric Finite Element Stiffness test matrix

Matlab code [4] is implemented for the purpose of comparing capabilities of Space Filling Curve with other industrial strength algorithms. The comparison is made on the basis of the computational time required to compute LU factorization, the number of nonzero entries after LU factorization and time needed for matrix-vector multiplication. Figure 5.22 shows raw sparse symmetric Finite Element matrix containing 0.06 % of non-zero entries and Figure 5.23 shows LU factorization of test case matrix containing 22 % of non-zero entries.

Machine Used

The matrix structure analysis is tested on MacOS operating system with Intel i3 dual core processor with each core having 32KB of L1 Cache, 256 KB of L2 Cache and 3 MB of L3 Cache. Matlab version 2017b is used for measuring performance impacts of different mesh reordering techniques.

5.5.1. Impact on Matrix Bandwidth

Figure 5.22 shows the sparsity pattern of the test Finite Element matrix and Figure 5.23 shows fill-ins after LU factorisation. Figures 5.24, 5.26 and 5.28 show matrix structure after application of Reverse Cutkill Mckee (RCM)[5], Approximate sparse minimum degree reordering (AMD) [6] and Space Filling Curve (SFC) respectively.

Reordering with Morton-order Space Filling Curve does lead to a significant drop in bandwidth but not as remarkable improvement as with RCM. The band structure of reordered

[4] https://github.com/computingdolas/Matrix_Reordering
[5] http://ciprian-zavoianu.blogspot.nl/2009/01/project-bandwidth-reduction.html
[6] http://epubs.siam.org/doi/10.1137/1031001

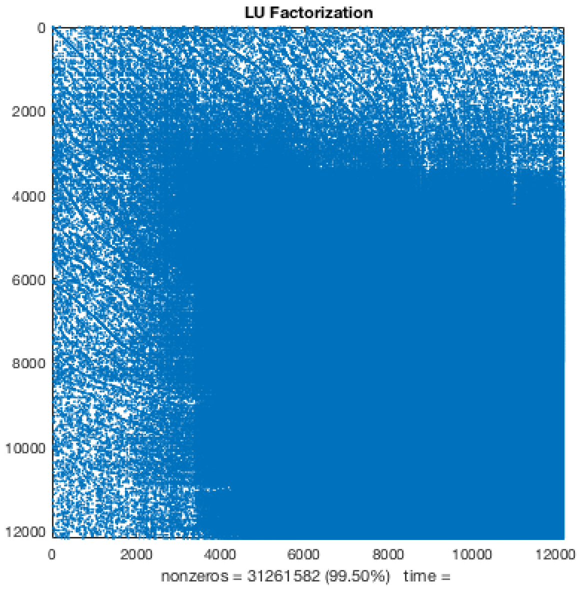


Figure 5.23: LU factorisation of Global Stiffness Test Matrix

matrix in Figure 5.28 shows the periodic presence of off diagonal entries because the Morton-order Space Filling Curve eventually generates massive jumps while connecting hierarchical blocks creating these entries in the matrix.

5.5.2. Performance Impact on LU Factorization

This section compares the performance impact of different matrix reordering schemes on time to compute LU factorization and the number of nonzero entries after LU factorization. Figures 5.25, 5.27 and 5.29 shows sparsity pattern of the LU factorization after the application of the different matrix reordering schemes. From the visual inspection, one can conclude that RCM reordering has the best impact on LU factorization followed by AMD and then lastly SFC.

It is evident from Figures 5.30 and 5.31 that RCM and SFC nearly have the same effect for time to LU factorization and number of nonzero entries after LU factorization. RCM and SFC both taken ≈50 seconds to compute LU factorization and retain nearly ≈2 % of matrix entries after LU factorization.

Matrix Structure Analysis

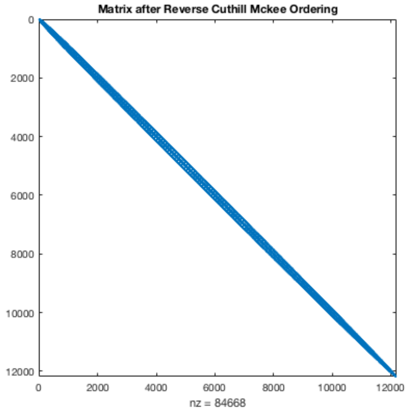


Figure 5.24: RCM reordering

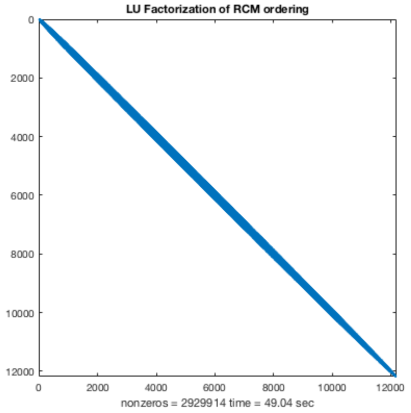


Figure 5.25: LU factorization after RCM reordering

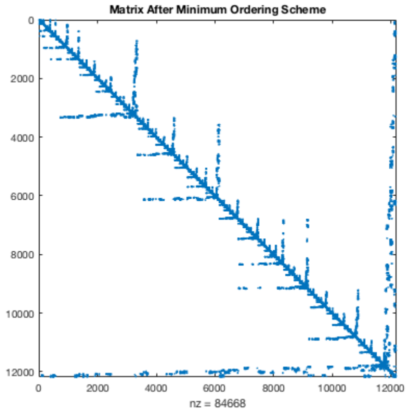


Figure 5.26: AMD reordering

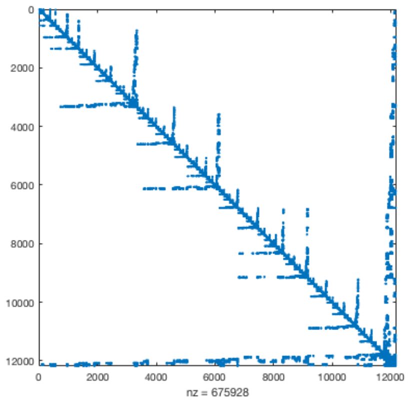


Figure 5.27: LU factorization after AMD reordering

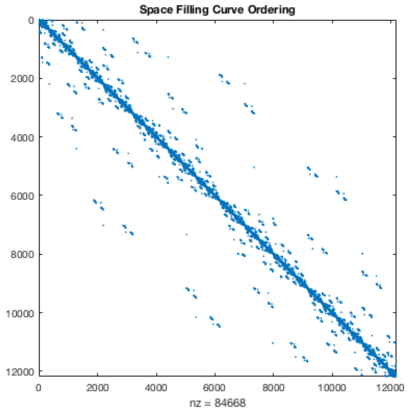


Figure 5.28: SFC reordering

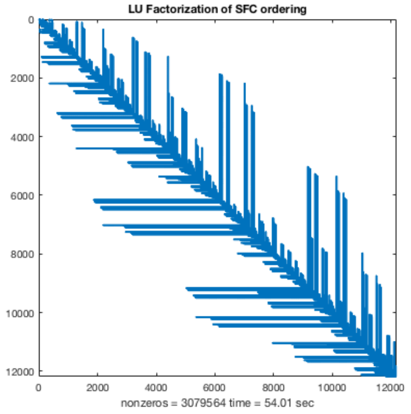


Figure 5.29: LU factorization after SFC reordering

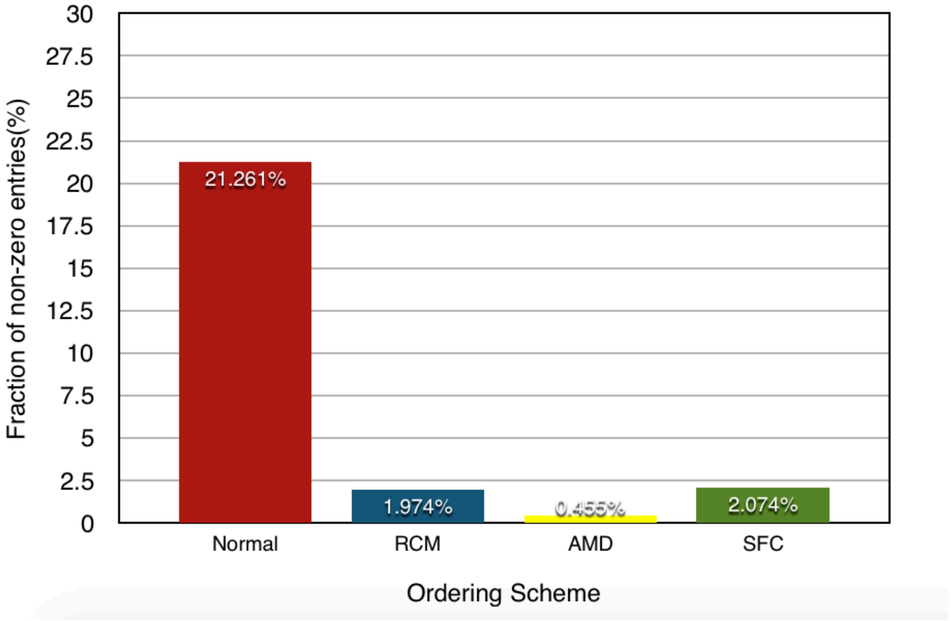


Figure 5.30: Fraction of non-zeros after LU factorization

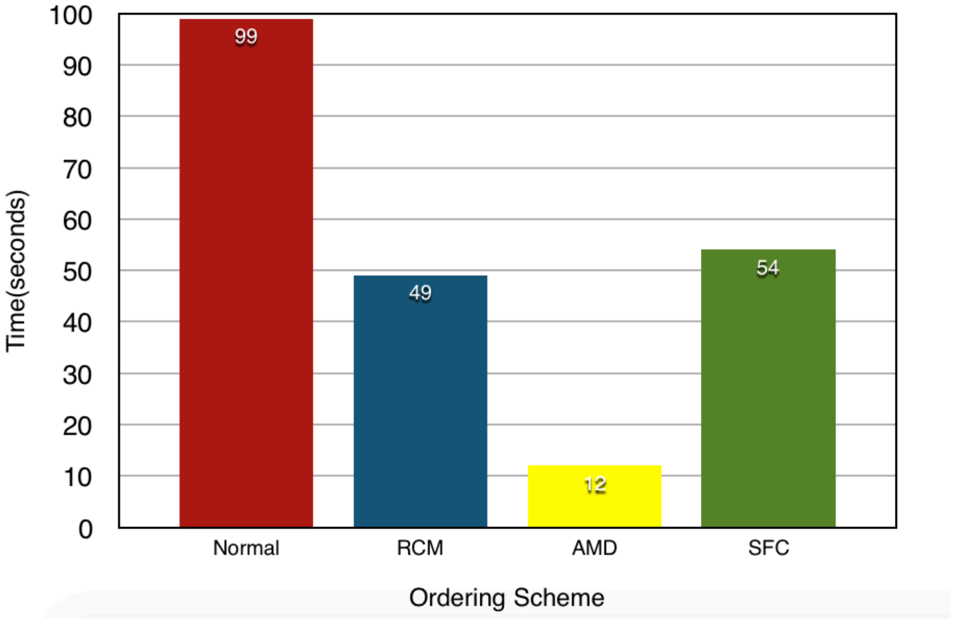


Figure 5.31: Time to compute LU factorization

5.6. Conclusion

This chapter started with an introduction to eigenvalue problem of an optical beam waveguide propagation governed by the two dimensional Helmholtz equation. The discretization of the Helmholtz equation on the two-dimensional domain was achieved by FEM leading to the solution of a linear system of equations for lowest eigenvalue and associated eigenvector. The relevance of performance analysis on matrix assembly, conjugate gradient solver, sparse matrix-vector multiplication was established followed by the review of mesh data reordering algorithms.

The aim of this chapter was to qualitatively and quantitatively grasp performance impact of data placement and data access pattern on the matrix assembly and linear system solver in the Finite Element application. Three different artificially created hardware scenarios distinguished by data locality and data access pattern were set up to capture performance impact.

Putting it all together, remote memory access does not affect cache utilisation but increases the time to access data, impacts memory subsystem significantly leading to a substantial increase in DRAM traffic, CPU idling time and associated energy consumption. On the other hand, the performance of both inverse power iteration and matrix assembly is also significantly affected by Morton-order Space Filling Curve ordering of mesh elements and vertices. It enables efficient usage of cache hierarchy due to reduced penalties from capacity and conflict cache misses. Overall, there is ≈58 % increase in CPU time and energy consumption due to remote memory access and ≈43 % and ≈33 % improvement in CPU time and energy consumption due to improved data access pattern.

Analysis of sparse symmetric Finite Element matrix structure justifies capability of the Morton-order Space Filling Curve for bandwidth reduction, reducing fill in after LU factorization. The speeding up of sparse matrix-vector multiplication concretely explains the above results.

6

An Introduction to the Material Point Method

6.1. Introduction

This chapter gives a brief overview of the fundamentals of solid mechanics and introduces the Material Point Method (MPM) as an advanced numerical scheme to solve problems in continuum mechanics similar to the Finite Element approach. The Material Point Method discretizes the continuum body into a set of material points or particles analogous to the Gauss points in the Finite Element method. The MPM [14] is an advanced and relatively young numerical technique currently used by many researchers to simulate multi phase phenomenon and large deformation problems mainly in Geotechnical Engineering.

The Material Point Method is the variant of the particle in cell method in which particles carry all physical properties of the continuum whereas the mesh carries no permanent information. Each computational cycle involves a Lagrangian phase and a Eulerian phase. In Lagrangian phase, physical variables of particles are updated while in the Eulerian phase the grid is remapped to its previous configuration leaving particle at their new positions.

MPM discretization involves superimposing a Finite Element background grid on material particles shown in Figure 6.1. In each computational step, the nodal properties are advanced either via explicit or implicit time integration scheme. Afterwards, particle-grid interaction takes place to advance particles properties such as position, stress, velocity in time.

6.2. Governing Equations

This section introduces the governing equations of solid mechanics based on the conservation of mass, momentum and energy. In this section, the updated Lagrangian formulation is discussed because it is widely used for MPM discretization. Essential mathematical ideas have been taken from Chapter 2 and 3 of the book, The Material Point Method [16].

6.2.1. Reynold's Transport Theorm

The material derivative of the volume integral of a generic function $f(\mathbf{x},t)$ over the time dependent region $\Omega(t)$ that has boundary $\partial\Omega(t)$ is calculated in [16] as follows :

$$\frac{D}{Dt}\int_{\Omega(t)} f(\mathbf{x},t)dV = \frac{D}{Dt}\int_{\Omega_0(t)} f(\mathbf{x},t)JdV_0 = \int_{\Omega_0(t)} [\dot{f}(\mathbf{x},t)J + f(\mathbf{x},t)\frac{DJ}{Dt}]dV_0 \tag{6.1}$$

where $\Omega(t)$ and $\Omega_0(t)$ represent current and initial configuration respectively and $\dot{f}(\mathbf{x},t) = \frac{Df(\mathbf{x},t)}{Dt}$ is the material derivative of the function $f(\mathbf{x},t)$ with respect to time. J is Jacobian matrix and dV satisfies the below equation

$$dV = JdV_0 \tag{6.2}$$

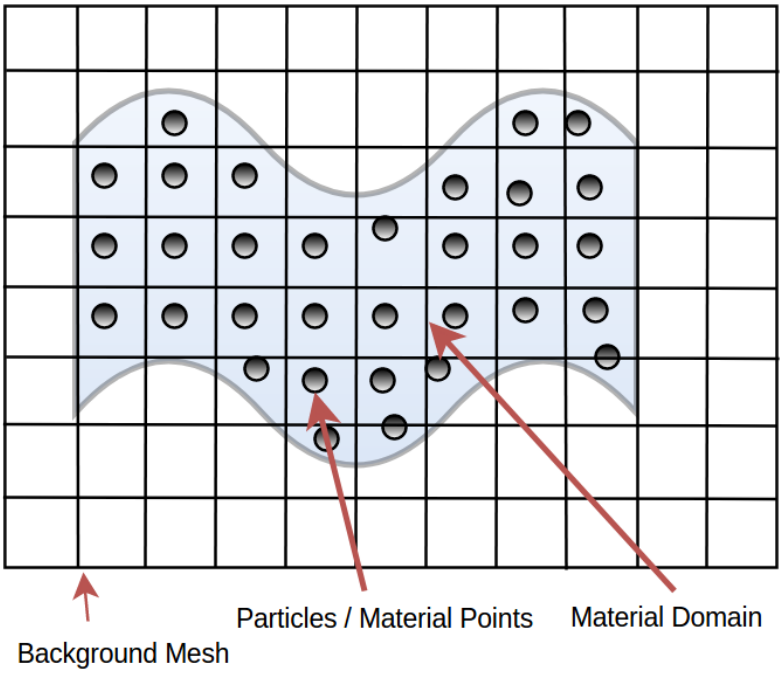


Figure 6.1: Particles descritizing material domain embedded in background grid

also illustratively shown in Figure 6.2 . It is known that,

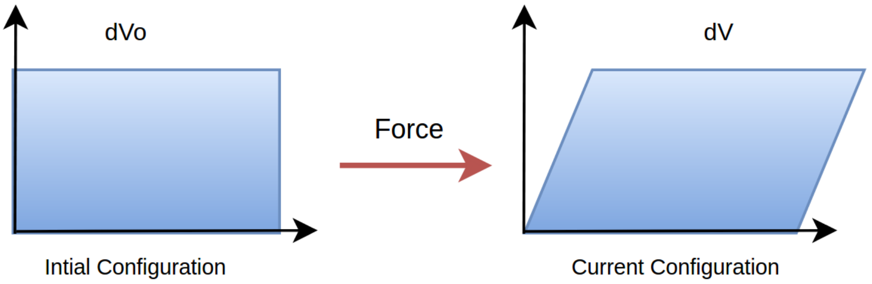


Figure 6.2: Deformation of current and initial configuration

$$\frac{DJ}{Dt} = J\nabla \cdot \mathbf{v} \tag{6.3}$$

Substituting (6.3) into (6.1) result's in Reynold's transport equation

$$\underbrace{\frac{D}{Dt}\int_{\Omega(t)} f(\mathbf{x},t)dV}_{\text{Total change in time}} = \int_{\Omega(t)} [\underbrace{\dot{f}(\mathbf{x},t)}_{\text{Temporal change}} + \underbrace{f(\mathbf{x},t)\nabla \cdot \mathbf{v}}_{\text{Spatial change}}]dV \tag{6.4}$$

6.2.2. Conservation of Mass

The total mass of the body in the current configuration $\Omega(t)$ can be described as

$$m = \int_{\Omega(t)} \rho(\mathbf{x},t)dV \tag{6.5}$$

where $\rho(\mathbf{x},t)$ is the local density. The conservation of mass requires that, material derivative of mass over time be equal to zero. Applying (6.4) on 6.5 with $f(x,t) = \rho(x,t)$ results in

$$\frac{D}{Dt}\int_{\Omega(t)} \rho(\mathbf{x},t)dV = \int_{\Omega(t)} (\dot{\rho}(\mathbf{x},t) + \rho(\mathbf{x},t)\nabla \cdot \mathbf{v})dV = 0 \tag{6.6}$$

Finally the mass conservation or continuity equation becomes

$$\dot{\rho}(\mathbf{x},t) + \rho(\mathbf{x},t)\nabla \cdot \mathbf{v} = 0 \tag{6.7}$$

6.2.3. Conservation of Linear Momentum

The conservation of linear momentum ensures that the material derivative of the linear momentum of continuum body Ω in current configuration is equal to the total force acting on it.

$$\underbrace{\frac{D}{Dt}\int_{\Omega(t)} \rho v(\mathbf{x},t)dV}_{\text{Rate of change of momentum}} = \underbrace{\int_{\Omega(t)} \rho\mathbf{b}(\mathbf{x},t)dV}_{\text{Body force}} + \underbrace{\int_{\partial\Omega(t)} \mathbf{t}(\mathbf{x},t)dA}_{\text{Traction force}} \tag{6.8}$$

where $\mathbf{b}$ is body force per unit mass and $\mathbf{t}$ is the external traction acting on the boundary $\partial\Omega(t)$. Using Reynold's transport theorm (6.4), the left hand side of (6.8) can be transformed to

$$\frac{D}{Dt}\int_{\Omega(t)} \rho v(\mathbf{x},t)dV = \int_{\Omega(t)} [\rho\dot{v} + \underbrace{(\dot{\rho} + \rho\nabla \cdot v)}_{\text{Continuity equation}}]dV \tag{6.9}$$

Using the continuity equation (6.7) from previous section,(6.9) becomes

$$\frac{D}{Dt}\int_{\Omega(t)} \rho v(\mathbf{x},t)dV = \int_{\Omega(t)} \rho\dot{\mathbf{v}}dV \tag{6.10}$$

Gauss divergence theorem is applied on the traction part in [16] on (6.8) as follows :

$$\int_{\partial\Omega(t)} \mathbf{t}(\mathbf{x},t)dA = \int_{\partial\Omega(t)} \mathbf{n}\cdot\sigma dA = \int_{\Omega(t)} \sigma\cdot\nabla dV \tag{6.11}$$

Combining (6.11), (6.8) and (6.9) leads to

$$\int_{\Omega(t)} (\rho\dot{\mathbf{v}} - \rho\mathbf{b} - \sigma\cdot\nabla)dV = 0 \tag{6.12}$$

which leads to equation of conservation for linear momentum as :

$$\rho\dot{\mathbf{v}} - \rho\mathbf{b} - \sigma\cdot\nabla = 0 \tag{6.13}$$

6.2.4. Boundary Conditions

There are two boundary conditions explained in this section. The displacement boundary condition given in (6.14) imposes displacement $\bar{\mathbf{v}}$ on displacement boundary nodes and traction boundary condition enforces $\bar{\mathbf{t}}$ on traction boundary nodes.

Displacement Boundary Condition

$$\mathbf{v}|_{\partial\Omega_u} = \bar{\mathbf{v}} \tag{6.14}$$

and

Traction Boundary Condition

$$\mathbf{n}\cdot\sigma = \bar{\mathbf{t}} \tag{6.15}$$

6.2.5. Weak Formulation

Conservation of mass is automatically ensured in the Material Point Method by construction therefore, this section will deal with discretization of the linear momentum equation over the computational domain by deriving its weak formulation. Taking test function as $\delta u_j \in \mathcal{R}^0$, $\mathcal{R}^0 = \{\delta u_j | \delta u_j \in \mathcal{C}^0, \delta u_j|_{\partial\Omega(u)} = 0\}$, combining linear momentum and traction boundary conditions, the weak formulation can be written as :

$$\int_{\Omega(t)} \delta u_j(\sigma_{ij,j} + \rho b_i - \rho \ddot{u}_i) dV = 0 \tag{6.16}$$

$$\int_{\partial\Omega(t)} \delta u_i(\sigma_{ij} n_j - \bar{t}_i) dV = 0 \tag{6.17}$$

Please note that Einstein's Summation Convention [1] is used here for simplicity. Expanding first term on the right hand side of equation 6.16 and using condition of test function $\delta u_j|_{\Gamma_u} = 0$, the resultant equation becomes :=

$$\begin{aligned}
\int_{\Omega(t)} \delta u_i \sigma_{ij,j} dV &= \int_{\Omega(t)} [(\delta u_i \sigma_{ij})_{,j} - \delta u_{i,j}\sigma_{ij}] dV \\
&= \int_{\partial\Omega(t)} \delta u_i \sigma_{ij} n_j dA - \int_{\Omega(t)} \delta u_{i,j}\sigma_{ij} dV \\
&= \int_{\partial\Omega(t)} \delta u_i \bar{t}_i dA - \int_{\Omega(t)} \delta u_{i,j}\sigma_{ij} dV
\end{aligned} \tag{6.18}$$

Using equation 6.18 into equation 6.16 we obtain

$$\underbrace{\int_{\Omega(t)} \rho \ddot{u}_i \delta u_i dV}_{\text{Inertial Term}} = \underbrace{\int_{\Omega(t)} \rho b_i \delta u_i dV - \int_{\partial\Omega(t)} \bar{t}_i \delta u_i dA}_{\text{External Force}} - \underbrace{\int_{\Omega(t)} \sigma_{ij} \delta u_{i,j} dV}_{\text{Internal Force}} \tag{6.19}$$

The weak formulation derived above is used Finite Element Analysis by taking test functions from $\mathcal{C}^0$. This complicated equation can be understood simply in terms of Newton's second law. The first term on the left hand side of (6.19) can be understood as inertial term or dynamics of the body $\Omega(t)$ whereas first and second term on the right hand side can be understood as body and traction forces respectively together as externally applied force and third term represent internal stresses. The equation (6.19) can be rewritten in form equation of motion as :

$$\mathbf{M}\ddot{\mathbf{u}}^t = \mathbf{F}^{ext} - \mathbf{F}^{int} \tag{6.20}$$

6.3. Material Point Method Formulation

The MPM formulation is similar to the Finite Element set-up but its computational step can be divided broadly into the Lagrangian phase and the Eulerian phase. In the Lagrangian phase, the particles deform with the Finite Element solution. In the Eulerian step, the grid is simply reset to its original configuration leaving particles in updated position. This is illustratively shown in Figure 6.3 and mathematically formulated and explained in next two sections.

6.3.1. Material Point Method Discretization

The MPM discretizes the continuum body $\Omega(t)$ into the set of material points moving freely through the underlying fixed Eulerian grid. The information such as mass, momentum, energy, strain, stress and internal state variables are carried by the particles for history-dependent constitutive modelling. The fixed underlying grid is used for solving the momentum equation similar to the Finite Element Analysis.

[1] https://en.wikipedia.org/wiki/Einstein_notation

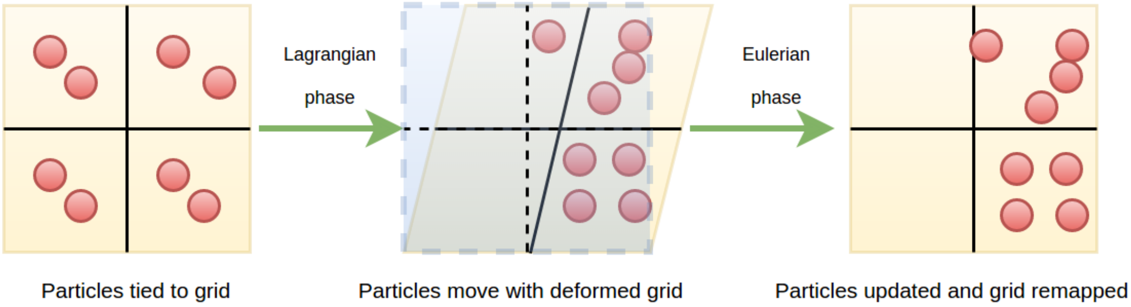

Lagrangian
phase
Eulerian
phase
Particles tied to grid
Particles move with deformed grid
Particles updated and grid remapped

Figure 6.3: Lagrangian and Eulerian Step in MPM

The Material Point Method involves the use of particles to discretize the continuum, and therefore dirac delta function can be used to approximate particles into weak formulation (6.19). The mass density is calculated as:=

$$\rho(x) = \sum_{p=1}^{np} m_p \delta(\mathbf{x} - \mathbf{x}_p) \tag{6.21}$$

where n_p is the number of particles, m_p is the mass of the particle, δ is the Dirac delta function with dimension of inverse of volume and $\mathbf{x}_p$ is the position of the particle. Substituting (6.21) into (6.19) results in

$$\sum_{p=1}^{n_p} m_p \ddot{u}_{ip} \delta u_{ip} + \sum_{p=1}^{n_p} m_p \sigma^s_{ijp} \delta u_{ip,j} - \sum_{p=1}^{n_p} m_p b_{ip} \delta u_{ip} - \sum_{p=1}^{n_p} m_p \bar{t}_{ip} h^{-1} \delta u_{ip} = 0 \tag{6.22}$$

where subscript p indicates evaluation of particular function at spatial position $\mathbf{x}_p$.

6.3.2. Lagrangian Phase and Eulerian Phase

In Lagrangian phase, particles move when grid deforms upon application of internal stress and external force. The position of particles can be interpolated from grid nodal position as

$$x_{ip} = N_{Ip} x_{iI}, I = 1 \cdots n_g \tag{6.23}$$

where n_g is number of the nodes. Similarly virtual displacement δu_{ip}, particle displacement u_{ip} and its derivative $u_{ip,j}$ can also be approximated from their nodal counterparts in the following way

$$\begin{aligned} \delta u_{ip} &= N_{Ip} \delta_{iI} \\ u_{ip} &= N_{Ip} u_{iI} \\ u_{ip,j} &= N_{Ip,j} u_{iI} \end{aligned} \tag{6.24}$$

Using (6.24) into the weak form for MPM formulation and using arbitrariness of δu_{iI} and $\delta u_{iI}|_{\partial\Omega_u(t)} = 0$ results in individual nodal momentum equation described below

$$\dot{p}_{iI} = f^{external}_{iI} - f^{internal}_{iI} \quad \forall x_I \notin \Gamma_u \tag{6.25}$$

where

$$\dot{p}_{iI} = m_{IJ} \dot{u}_{iJ}, \quad i \in \mathcal{C} = \{x, y, z\} \tag{6.26}$$

$$m_{IJ} = \sum_{p=1}^{n_p} m_p N_{Ip} N_{Jp} \tag{6.27}$$

is the consistent mass matrix of the background grid. Moreover,

$$f_{iI}^{internal} = \sum_{p=1}^{n_p} N_{Ip,j}\sigma_{ijp}\frac{m_p}{\rho_p} \tag{6.28}$$

and

$$f_{iI}^{external} = \sum_{p=1}^{n_p} m_p N_{Ip} b_{ip} + \sum_{p=1}^{n_p} N_{Ip}\bar{t}_{ip} h^{-1}\frac{m_p}{\rho_p} \tag{6.29}$$

are the internal and the external forces respectively. Once the accelerations from (6.25) are determined , velocity and positions of the particles are updated by any of the time integration schemes. The equation (6.25) represents spatial discretization and this equation will be the starting point of Chapter 8 and 9 where we will discuss time integration schemes.

In the Eulerian phase, the grid is simply remapped to its original configuration, while particles remain in their updated positions. The main reason behind this step is to avoid excessive mesh distortion since particles moves with grid in the Lagrangian phase. The background grid is just used to evaluate shape functions and its gradients. Reconstruction of the solution on the remapped grid is necessary to advance them in next computational step. This is achieved by interpolating physical variables from particles to nodes.

7

Challenges for Achieving High Performance for the Material Point Method

7.1. Introduction

This chapter will investigate challenges of achieving high performance in the Material Point Method (MPM) introduced in Chapter 6 by examining insights into data traversal, data placement and interaction between elements, vertices/ nodes and particles. In MPM, particle-grid interaction is core component where data placement of elements, vertices and particles in memory and their interleaved unstructured memory access pattern is a major obstruction to solve more complexed and massive problems efficiently.

Data traversal and data placement are two important pillars determining computational performance for the Material Point Method even on small scale servers. This chapter first reviews memory access pattern and identifies potential challenge in achieving high performance for particle-grid interaction in the MPM context and then proposes Space Filling Curve (SFC) as a technique to traverse element of underlying grid and vertices array.

7.1.1. Understanding Memory Access Pattern in Particle to Grid Interaction

This section will investigate the impact of memory access patterns in the particle-grid interaction in MPM on the computational performance. The MPM computational cycle is illustrated in Figure 7.1.

There are two major data structures in the particle-grid interpolation kernel. First is the mesh array and second is the particle array. Figure 7.2 shows 3 particles residing in each element and its unordered connectivity. The original problem is traversing the elemental array and accessing non-strided particle data. Since particle data is not aligned with the travelling order of the element array, huge cache miss penalties will lead to performance degradation, redundant engagement of processing cores and increase in traffic to main memory (DRAM).

In the MPM formulation particles discretize the continuum, and the background mesh is just used for solving the governing equations, but nonetheless, the physical variables between the nodes and the particles have to be exchanged to initialize for the next time step. This complicated interaction makes it quite hard to utilise the underlying hardware resources efficiently since mesh data and particle data may be located in entirely different parts of memory leading to an inefficient use of the memory subsystems. In MPM, the particles indirectly access or update the nodal data of the host element and therefore the data layout of the nodal array also implicitly plays an important role.

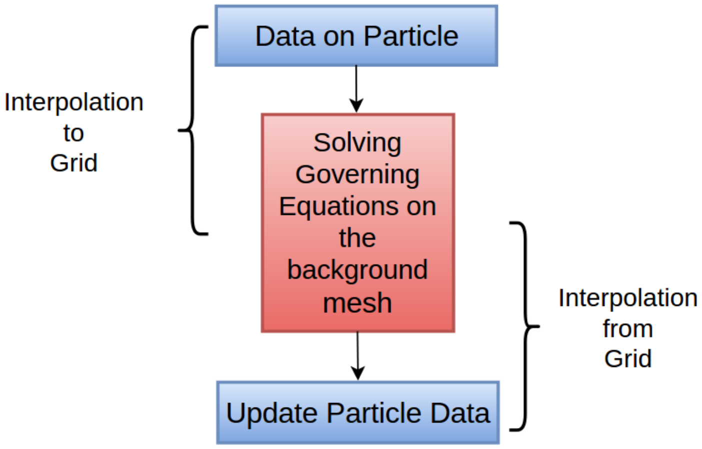


Figure 7.1: Concept of MPM simulation

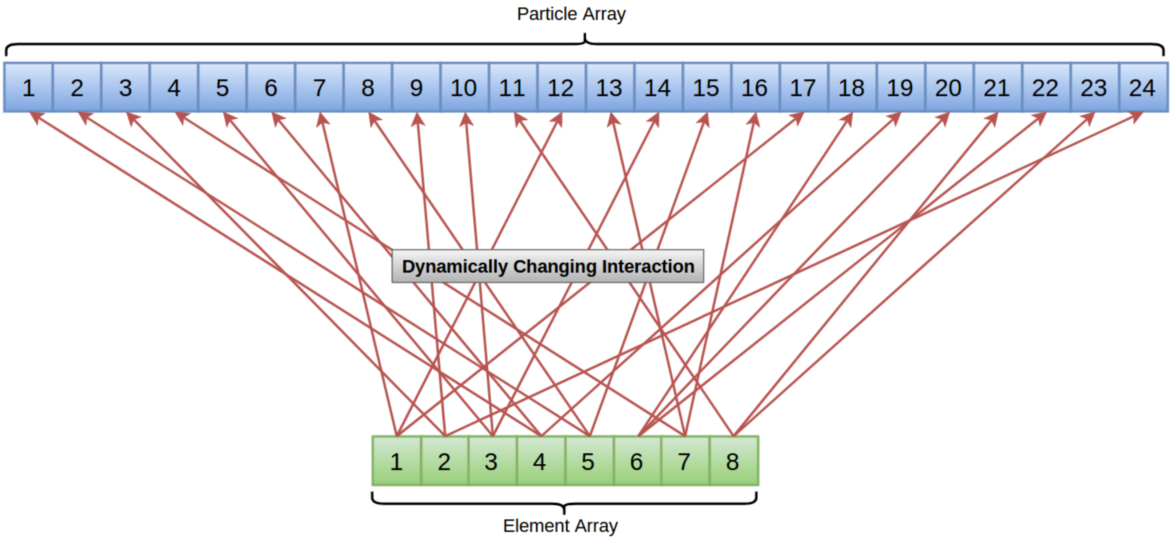


Figure 7.2: Particle-grid connectivity

Figure 7.3 shows the intricate connectivity between elements, vertices and particles. Consider a for loop travelling through the elemental array, accessing data of residing particles and updating nodal data and vice versa. In typical scenarios like this, it is important that the particle data placement and the vertex array numbering be aligned to order of the elemental array as shown in Figure 7.4 to fully exploit memory hierarchy efficiently. The idea and its variant will be discussed briefly in the next section.

7.1.2. Tackling Particle to Grid Interaction Efficiently in MPM Simulation

In MPM, particles keep moving throughout the grid dynamically changing the linked data structure and therefore it becomes impossible to maintain a single data layout for particles in memory.

Each computational cycle involves implicit interaction between arrays of particle and vertices invoking lots of random access. The optimal solution to efficiently handle mesh data

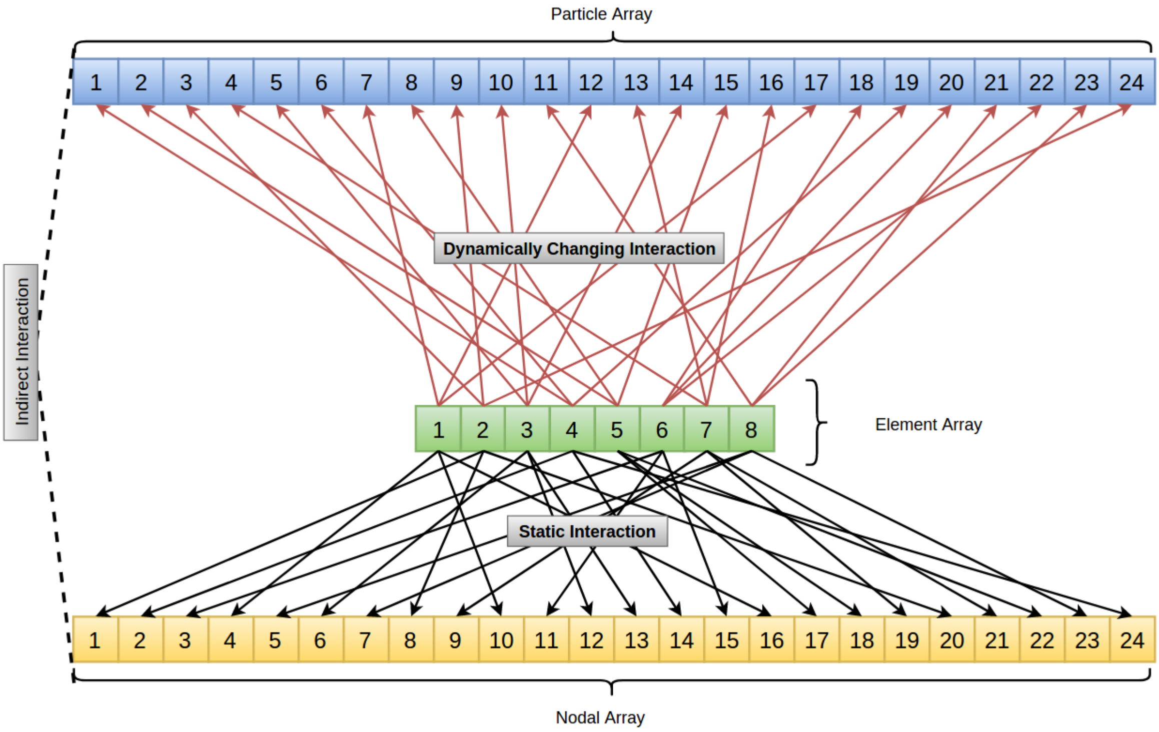


Figure 7.3: Particles, nodes and elements interaction in memory

layout is to reorder elements and vertices to preserve temporal and spatial locality and sorting of particles to align particle data with the elemental array indirectly aligning with vertex array. However this can be computationally intensive due to involvement of expensive sort algorithms but since sorting is not required in every time step it can also be performed every few time steps to align the data.

7.1.3. The Hidden Potential Challenge

There is another perspective which should be given enough attention especially in the context of the Material Point Method. It is clear that particles keep moving within the underlying grid perhaps in every time step, dynamically changing the linked data structure and therefore it is difficult to maintain a single layout of particle data structure to enhance cache efficiency. On the other hand it becomes clear that only those parts of static background grid which are occupied by particles are used actively leaving the other parts inactive . In other words a set of elements gets activated and deactivated in a particular time step depending upon the presence or absence of particles inside the elements as shown in Figure 7.5.

Figure 7.6 shows the corresponding memory layout of the mesh. The hidden challenge is to order the set of active elements as close as possible to each other in memory so as to mimic optimal interaction as described by Figure 7.4. This approach involves generating mesh layouts dynamically within time steps also shown illustratively in Figure 7.7

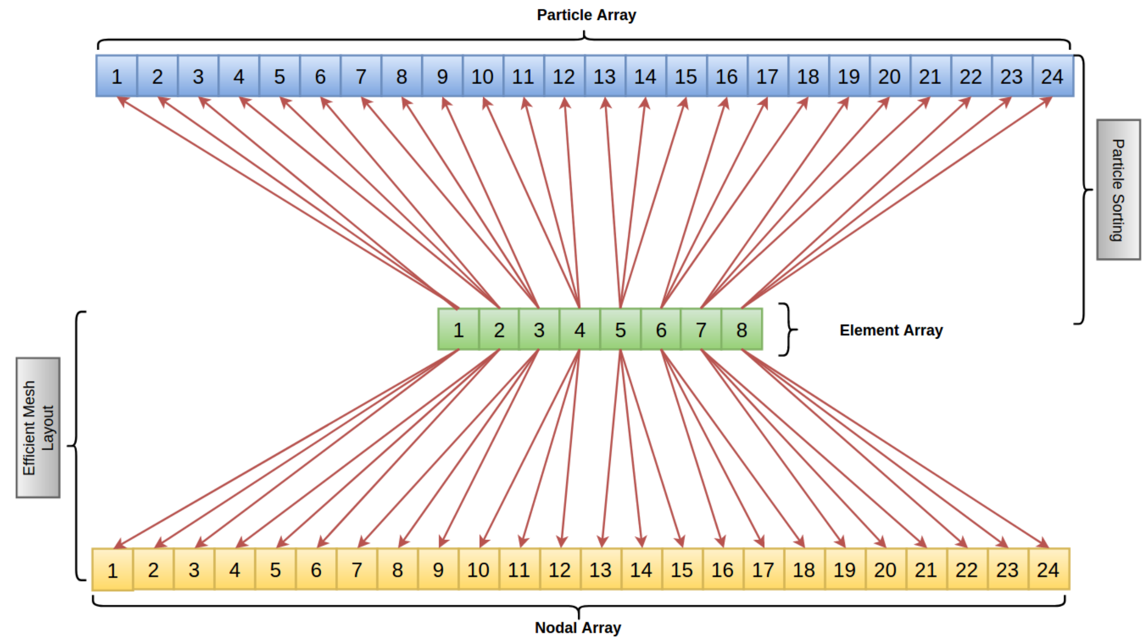


Figure 7.4: Optimal interaction between particles, elements and vertices

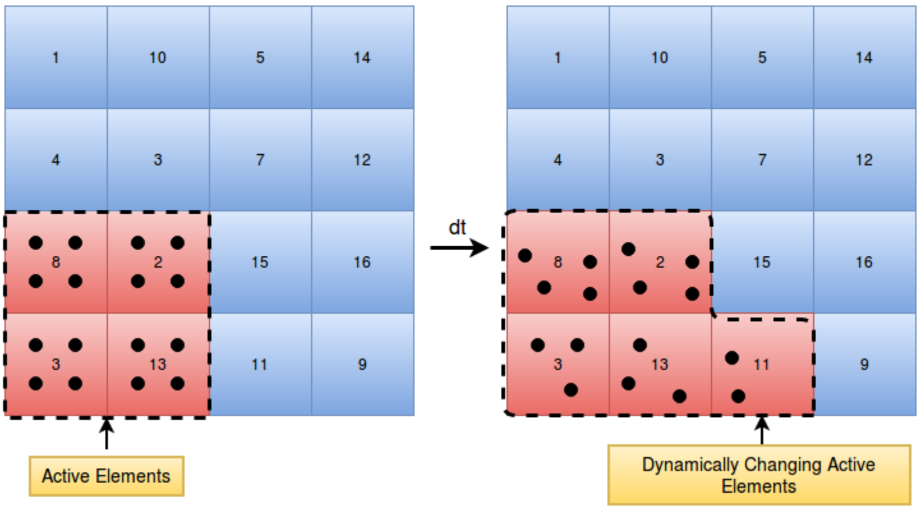


Figure 7.5: Set of dynamically changing active element and their natural ordering

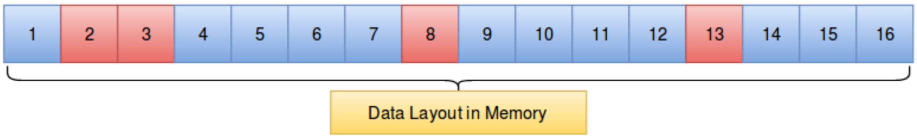


Figure 7.6: Data layout of active elements in memory with natural ordering

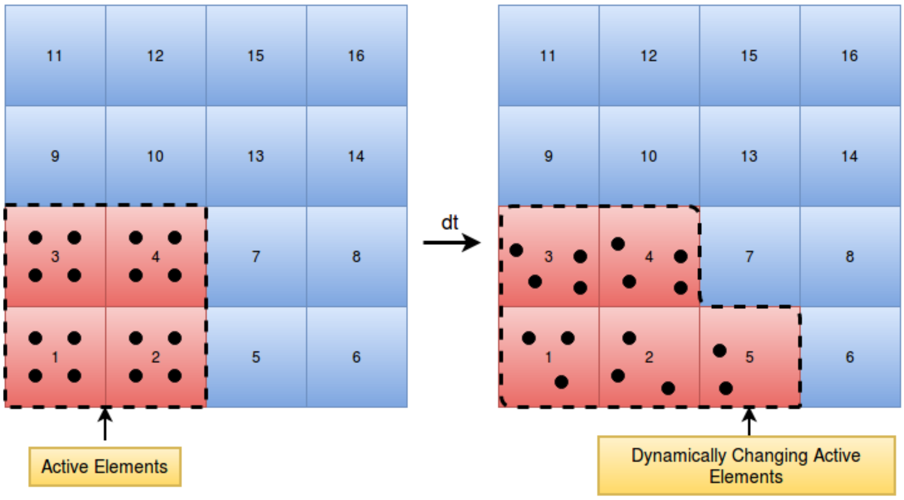


Figure 7.7: Set of active elements with SFC ordering

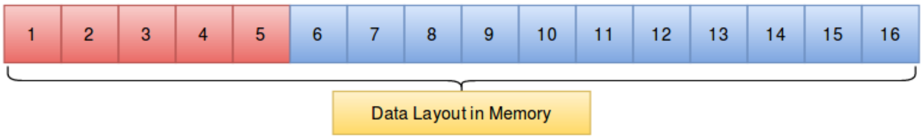


Figure 7.8: Data Layout of active elements in memory with SFC Ordering

7.2. Focus of Study

Inspired by the ability of Space Filling Curves to reorder data to preserve spatial and temporal locality and also minimize cache misses as seen in earlier chapters, this section concludes that the use of Morton-order SFC to reorder elements and vertices and analyzes the performance impact in the Material Point Method context.

In the coming chapters on analysis of Explicit and Implicit MPM solvers, performance impact of reordering the elements and vertices of the background mesh will be documented and conclusions will be drawn to assess qualitative impact of cache oblivious data layouts on the Material Point Method, where particle sorting and mesh layout both play equally important role

8

Performance Analysis of Explicit MPM 3D Simulation

8.1. Introduction

This chapter explains the explicit Material Point Method algorithm, its computational efficiency, limits and discusses the performance analysis of mesh reordering on the test case. The explicit Material Point Method is effective and efficient in accurately capturing high frequencies for the transient development of simulations such as impact or blast. In such scenarios the explicit Material Point Method offers computational efficiency because of two reasons: first, the time-step size matches with the time-step size required by the stability criteria and second, it involves an explicit update of particles moving within the grid at little computational cost.

This chapter first introduces the explicit time integration scheme, discusses a brief overview of stability criteria for the explicit Material Point Method to advance in time and then provide in depth explanation of explicit MPM algorithm in detail.

8.1.1. Explicit Time Integration scheme and Stability Criteria

This section will give a brief introduction to the leapfrog time integration scheme and stability criteria for linear elastic material. In the explicit scheme, the position is updated at full-time step, but velocity is advanced only half a time in such a way to interleave update of position and velocity. In explicit MPM, both particles and nodal properties are updated using the leapfrog time integration scheme. The detailed mathematical description is available in the book The Material Point Method [16] and only essential mathematical ideas are presented here. Taking nodal equation of motion at n^{th} time step :

$$m_{iI}\ddot{u}_{iI}^{n} = f_{iI}^{n} \tag{8.1}$$

and knowing m_{iI} and f_{iI}^{n} at t^{n}, the velocity $\dot{u}_{iI}^{n+1}$ and position u_{iI}^{n+1} have to be calculated at t^{n+1}. This can be done in the following way. The nodal acceleration is $\ddot{u}_{iI}^{n}$ calculated from (9.1) and

$$\dot{u}_{iI}^{n+\frac{1}{2}} = \dot{u}_{iI}^{n-\frac{1}{2}} + \Delta t^{n}\ddot{u}_{iI}^{n} \tag{8.2}$$

$$u_{iI}^{n+1} = u_{iI}^{n} + \Delta t^{n+\frac{1}{2}}\dot{u}_{iI}^{n+\frac{1}{2}} \tag{8.3}$$

Here, the nodal velocity $\dot{u}_{iI}^{n+\frac{1}{2}}$ at time level $t^{n+\frac{1}{2}}$ is updated using (8.2) and the position $u_{iI}^{n+1}timelevel$ at t^{n+1} is updated using (8.3). Finally t^{n+1} and n are also updated as shown below which completes the cycle.

$$t^{n+1} = t^n + \Delta t^{n+\frac{1}{2}} \tag{8.4}$$

$$n = n + 1 \tag{8.5}$$

The time integration scheme described above is conditionally stable in a sense that, time step size Δt must be smaller that critical time step size Δt_c which is given in [16] as

$$\Delta t_c = min|_{elements} \frac{l^e}{c} \tag{8.6}$$

where l^e is the characteristic length of the element in the mesh and c is the adiabatic speed of sound formulated for linear elastic material in [16] as

$$c = \sqrt{\frac{E(1-\nu)}{(1+\nu)(1-2\nu)\rho}} \tag{8.7}$$

In explicit formulations the disturbance should not travel more than the characteristic length of element within a single time step and, therefore, the time step size has to be very small which makes this scheme computationally expensive for problems where high-frequency analysis is not required.

8.1.2. Explicit MPM Algorithm

This section describes important steps in the explicit MPM algorithm and their impact on performance. Figure 8.1 describes the *Update Stress Last* (USL) variant of explicit MPM algorithm in which stress on the particle is updated at the end of the each time step.

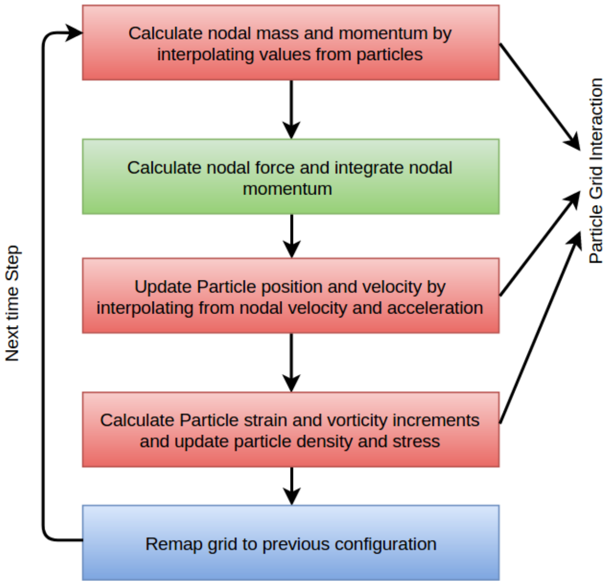

Calculate nodal mass and momentum by interpolating values from particles
Calculate nodal force and integrate nodal momentum
Update Particle position and velocity by interpolating from nodal velocity and acceleration
Calculate Particle strain and vorticity increments and update particle density and stress
Remap grid to previous configuration
Next time Step
Particle Grid Interaction

Figure 8.1: Explicit MPM Algorithm

The explicit MPM algorithm consists of the following steps :

$$
\begin{aligned}
m_I^n &= \sum_{p=1}^{n_p} m_p N_{Ip}^n \\
p_{iI}^{n-\frac{1}{2}} &= \sum_{p=1}^{np} m_p v_{ip}^{n-\frac{1}{2}} N_{Ip}^n
\end{aligned}
\tag{8.8}
$$

The grid nodal mass and momentum are calculated by interpolating from particle mass and momentum as shown in (8.8). The nodal internal force $f_{iI}^{int,n}$, the external force $f_{iI}^{ext,n}$ and the total force f_{iI}^n are calculated afterwards :

$$
\begin{aligned}
f_{iI}^{int,n} &= \sum_{p=1}^{n_p} N_{Ip,j}^n \sigma_{ijp} \frac{m_p}{\rho_p} \\
f_{iI}^{ext,n} &= \sum_{p=1}^{n_p} m_p N_{Ip}^n b_{ip}^n + \sum_{p=1}^{n_p} N_{Ip}^n \bar{t}_{ip}^n h^{-1} \frac{m_p}{\rho_p} \\
f_{iI}^n &= f_{iI}^{ext,n} - f_{iI}^{int,n}
\end{aligned}
\tag{8.9}
$$

In the next step, nodal momentum is integrated in time as follows :

$$
p_{iI}^{n+\frac{1}{2}} = p_{iI}^{n-\frac{1}{2}} + f_{iI}^n \Delta t^n
\tag{8.10}
$$

assuming uniform Δt for all time steps. The particle velocity and position is updated by :

$$
\begin{aligned}
v_{ip}^{n+\frac{1}{2}} &= v_{ip}^{n-\frac{1}{2}} + \sum_{I=1}^{n_e} \frac{f_{iI}^n N_{Ip}^n}{m_I^n} \Delta t^n \\
x_{ip}^{n+1} &= x_{ip}^n + \sum_{I=1}^{n_e} \frac{p_{iI}^{n+\frac{1}{2}} N_{Ip}^n}{m_I^n} \Delta t^n
\end{aligned}
\tag{8.11}
$$

where n_e is the number of nodes for a particular element. The final step consists in calculating the strain and updating the particle density as follows ;

1. The grid nodal velocity is updated using

$$
v_{iI}^{n+\frac{1}{2}} = \frac{p_{iI}^{n+\frac{1}{2}}}{m_I^n}
\tag{8.12}
$$

2. The particle strain increment $\Delta\epsilon_{ijp}^{n+\frac{1}{2}}$ and vorticity increment $\Delta\Omega_{ijp}^{n+\frac{1}{2}}$ are updated using :

$$
\begin{aligned}
\Delta\epsilon_{ijp}^{n+\frac{1}{2}} &= \frac{1}{2}(N_{Ip,j}^n v_{iI}^{n+\frac{1}{2}} + N_{ip,i}^n v_{jI}^{n+\frac{1}{2}})\Delta t^{n+\frac{1}{2}} \\
\Delta\Omega_{ijp}^{n+\frac{1}{2}} &= \frac{1}{2}(N_{Ip,j}^n v_{iI}^{n+\frac{1}{2}} + N_{ip,i}^n v_{jI}^{n+\frac{1}{2}})\Delta t^{n+\frac{1}{2}}
\end{aligned}
\tag{8.13}
$$

3. The particle density is updated as follows : =

$$
\rho_p^{n+1} = \frac{\rho_p^n}{(1 + \epsilon_{ijp}^{n+\frac{1}{2}})}
\tag{8.14}
$$

The last step is to remap the deformed grid to its original configuration. The major computational work in the above algorithm is caused by the particle-grid interaction regarding the evaluation of shape functions and its derivatives. The mathematical formulation involves a lot of unstructured indirect random access to the underlying memory due to an implicit particle-node interaction. To properly utilise underlying memory subsystem, it is crucial to have data layout which encourages coalesced access pattern to main memory (DRAM).

8.1.3. Introduction to the Explicit MPM Solver

This section gives an introduction to the explicit MPM 3D solver Anura3D[1] which is a software tool for MPM analysis developed by the Deltares [2] . This software has a 3D and a 2D implementation of the Material Point Method and is used for simulating the physics involved with soil-water structure interaction and large deformation problems in computational Geomechanics. The process to perform a numerical simulation consists of three parts, and the schematic diagram of the procedure is shown in Figure 8.2.

1. Creation of input data (pre-processing with GID software).
2. Calculation with Anura3D software.
3. Visualization of the results with ParaView.

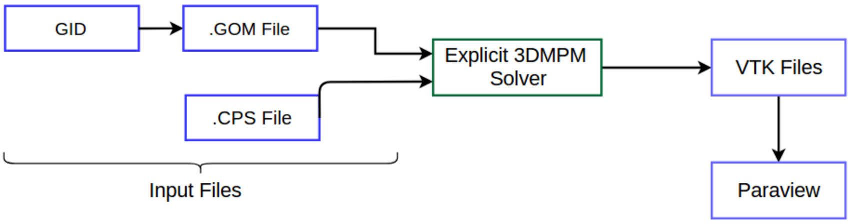


Figure 8.2: Schematic Diagram of Numerical Simulation with Anura3D

Anura3D is written in FORTRAN and supports three phase flows (solid + liquid + gas). The main limitations of Anura3D in simulating large scale problems is its poor parallel performance and the poor single-core performance due to excessive indirect addressing overhead by unstructured data access. This chapter explores a technique which lowers unstructured random access by reordering or generating cache oblivious mesh data layout of the underlying grid.

[1] http://www.mpm-dredge.eu/

[2] https://www.deltares.nl/nl/

8.2. Performance Analysis on Dam Collapse Simulation

This section discusses the impact of mesh reordering with Space Filling Curve on the performance of Anura3D for a test case. It explains the test case geometry, simulation details and also presents the data traversal visualisation of the underlying grid and relevant results .

8.2.1. Test Case Geometry, Boundary Condition and Simulation Parameters

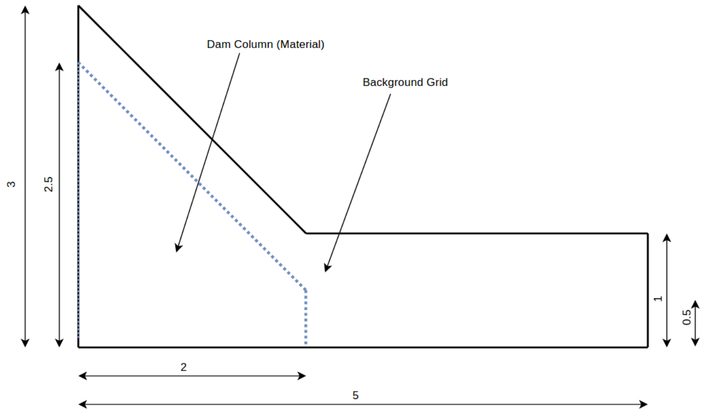


Figure 8.3: Illustrative Dam Collapse Geometry with dimensions

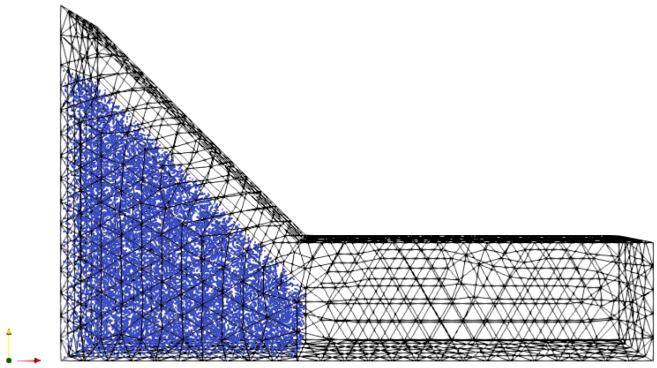


Figure 8.4: Material Domain discretization by particles and finite element mesh of background grid

Test Case Geometry

Figures 8.3 and 8.4 explain the geometry of the computational domain with the underlying grid and the material domain discretization with particles. Figure 8.3 shows the non-dimensional XY projection of the three-dimensional geometry with the depth of 1 unit. The mesh properties are shown in Table 8.1

Table 8.1: Mesh Parameters

Number of Elements	8583
Number of Nodes	13146
Number of Soil Particles / Element	4
Number of Elements with Particles	3600

Boundary Condition

Displacement boundary condition are applied on all faces forcing the normal displacements of all nodes on the respective face to be zero :

$$\mathbf{u}_{node}^{normal} = 0, \forall \ nodes \in \partial\Omega \tag{8.15}$$

Simulation Parameters and Material Model Description

The relevant simulation parameters and material model description are provided in Tables 8.2 and 8.3.

Table 8.2: Simulation Parameters

Simulation category	Single phase (soil)
Material Model	Mohr Columb
Time step size	0.01
Courant Number	0.8
Number of Time steps	500

Table 8.3: Material Model Description

Material Type	1-phase solid
Initial Porosity	0.3
Density solid (kg/m3)	2650
K0 - value	0.4
Material Model	Mohr Coulomb
Young's Modulus	1000
Poisson Ratio	0.2
Friction angle	30

Table 8.4: Load Condition

Gravity(z direction)	-9.8 m/s2

8.2.2. Methodology

Observing Figure 8.5 and comparing it with Figure 8.2, the Space Filling Curve toolbox is added as preprocessing tool to reorder the mesh file. The necessary steps taken to reorder the mesh are as follows :

1. ANURA3D uses ten noded tetrahedral elements for which reordering is even more challenging and expensive since only the first four nodes are used for the evaluation of the deformation matrix. The remaining nodes are used for particle tracking, but all ten nodes have to be stored. The ordering of elements and nodes/ vertices is critical in the sense that, determinant of deformation matrix becomes negative if evaluated otherwise. In this case, only the first four nodes are used for calculating the SFC index and the problem described above is solved by constructing the map between old and new vertices which makes it possible to maintain the ordering of nodes in particular fashion in the new geometric file.

2. Fixities are used in input geometric file (.GOM) to fix displacements of nodes at the boundary of the background grid. Since node numbering changes during mesh reordering, accordingly, nodal fixities also to be reordered as well which increases the cost of mesh reordering. Once elements are sorted and the new vertex numbering array is built the old fixities are mapped to new ones and sorted accordingly.

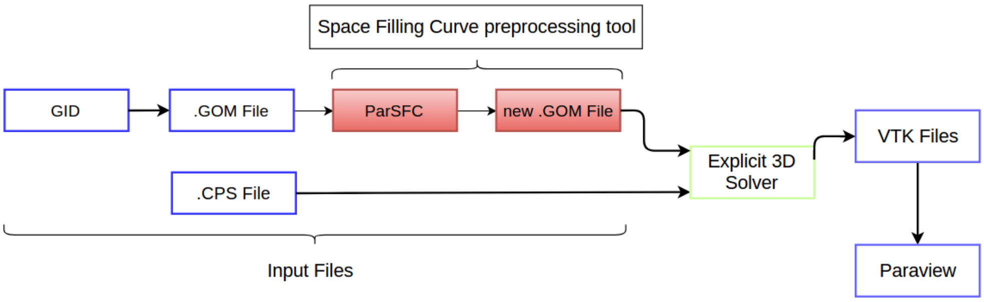


Figure 8.5: Space Filling curve preprocessing tool inserted in schematic diagram of ANURA3D

Machine used for Simulation

HP Elitebook 2570p is used which supports dual core Intel(R) i5-3360M processor. Since Anura3D does not efficiently support parallel computing, the analysis is carried out on a single core of an HP Elitebook having 32KB of private L1 Cache, 256 KB of private L2 cache and 3MB of shared L3 cache and supporting 16 GB of DRAM.

8.2.3. Visualization

Data Traversal Simulation

The simulation of data traversal of the underlying mesh with natural ordering is shown in Figures 8.6 to 8.11 and with SFC reordering is shown in Figures 9.15 to 9.20. Traversing the elements plays a crucial role in the explicit MPM method, and it largely determines the performance. The main idea behind the simulation of the element traversal is to grasp the concrete reason behind the unstructured data access visually. Natural ordering travels through the elements in random order.

Although the order in which elements are travelled is not very clear in these snapshots, but bird eye view places elements traversal with Morton-order SFC far better than without it. The blocking pattern of SFC is visible and this kind of layout encourages cache efficient computing. The simulation videos are available online at SimVideos [3] .

Dam Collapse Simulation

Dam collapse simulation is shown in Figures 8.18 to 8.23. It displays the deformation of the material continuum in blue under the natural load of gravity. It is intuitive from the visualisations that the column of material deforms as expected and occupies ≈30-35 % of the background grid at all time steps. It is evident from visualisations that particles quickly change their positions, and therefore, their connectivity changes in the underlying memory giving birth to more unstructured data access.

[3] https://github.com/computingdolas/Simulation_videos/tree/master

Data Traversal in Background Mesh with Natural Ordering

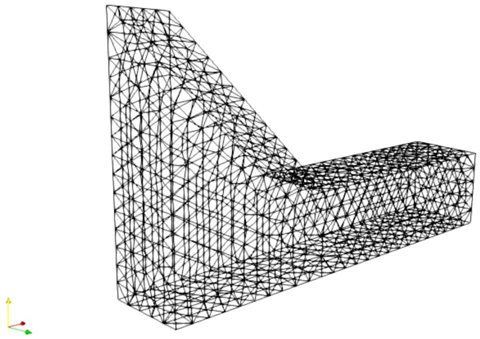


Figure 8.6: 0 Elements traveled

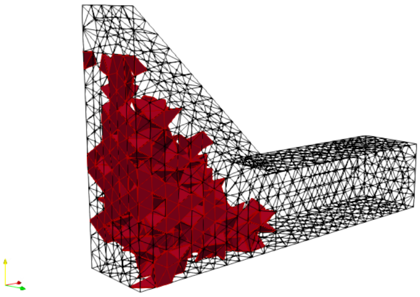


Figure 8.7: 1500 Elements traveled

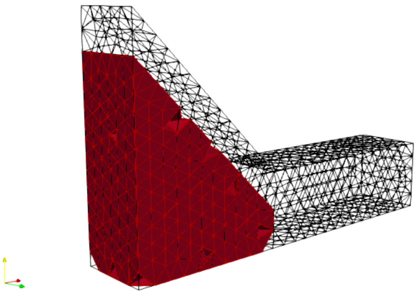


Figure 8.8: 3500 Elements traveled

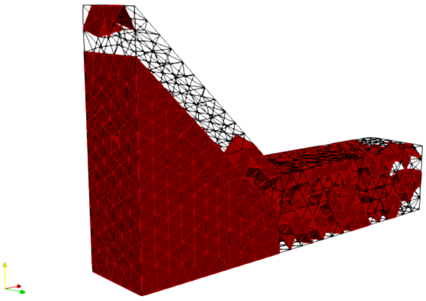


Figure 8.9: 5500 Elements traveled

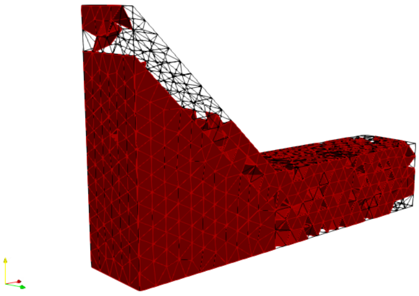


Figure 8.10: 7000 Elements traveled

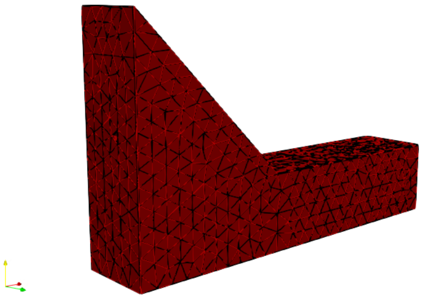


Figure 8.11: 8500 Elements traveled

Data Traversal in Background Grid with SFC Ordering

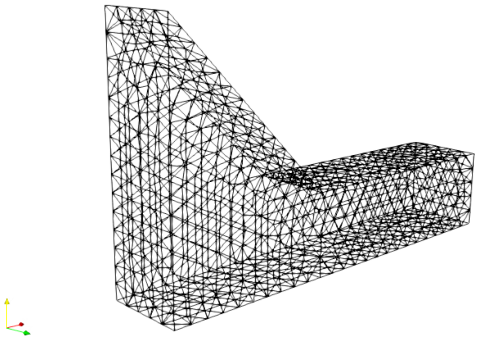


Figure 8.12: 0 Elements traveled

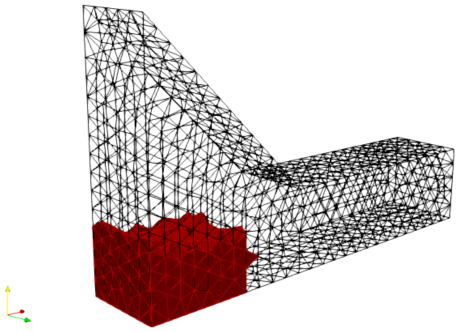


Figure 8.13: 1500 Elements traveled

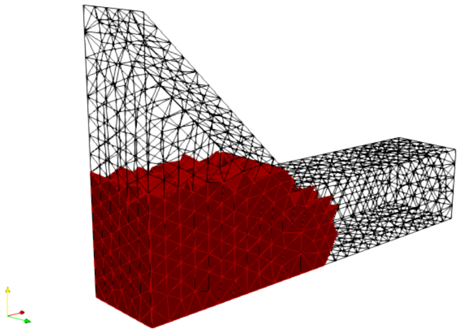


Figure 8.14: 3500 Elements traveled

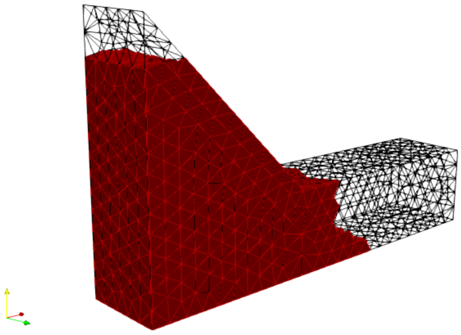


Figure 8.15: 5500 Elements traveled

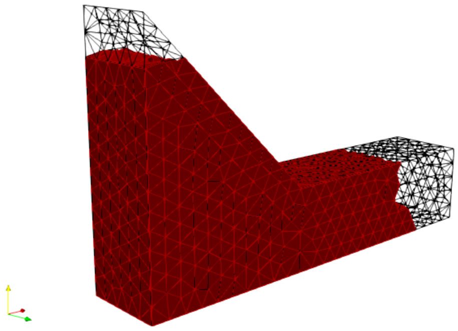


Figure 8.16: 7000 Elements traveled

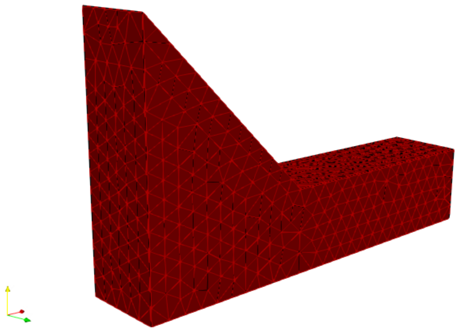


Figure 8.17: 8500 Elements traveled

Data Collapse simulation for 250 time steps

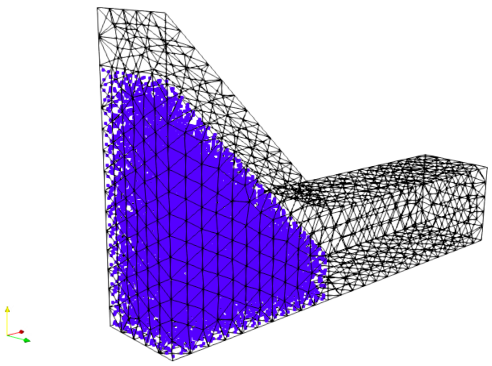


Figure 8.18: 0th Time step

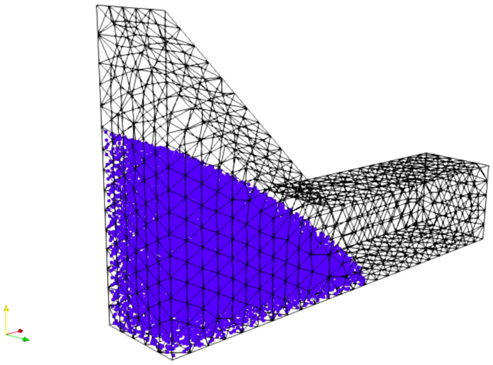


Figure 8.19: 50th Time step

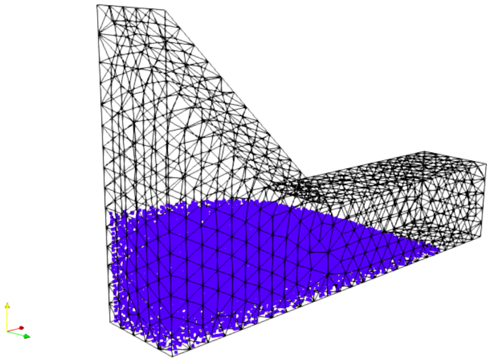


Figure 8.20: 100th Time step

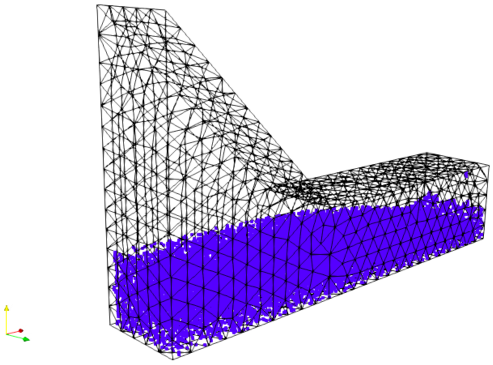


Figure 8.21: 150th Time step

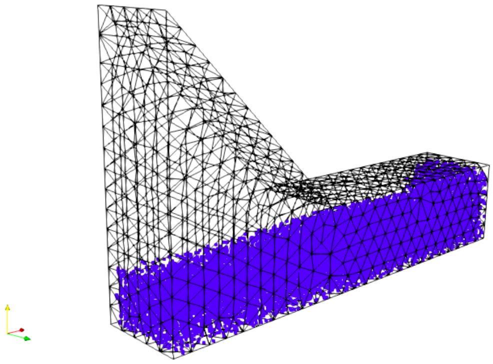


Figure 8.22: 200th Time step

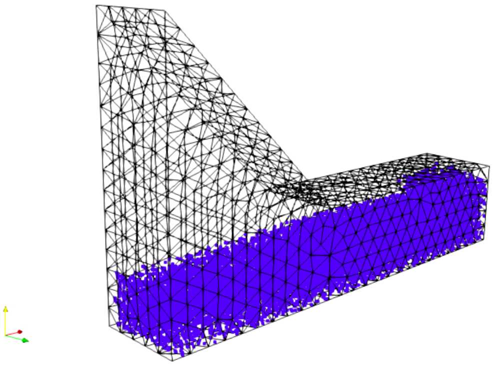


Figure 8.23: 250th Time step

8.2.4. Results and Discussion

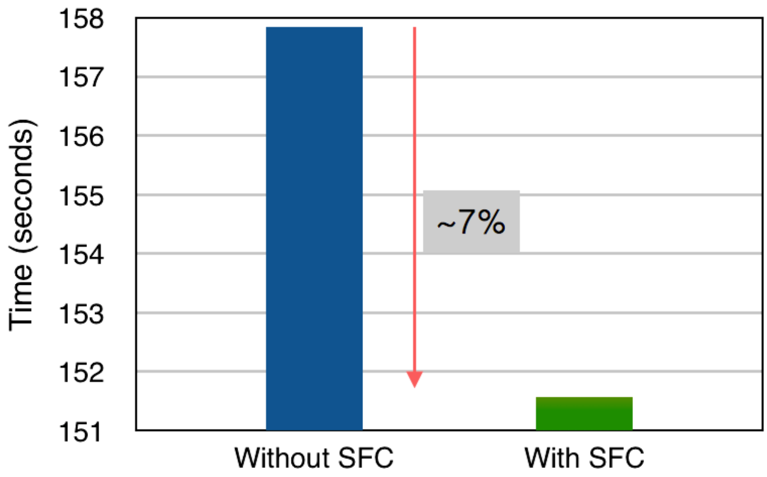


Figure 8.24: CPU time with and without SFC

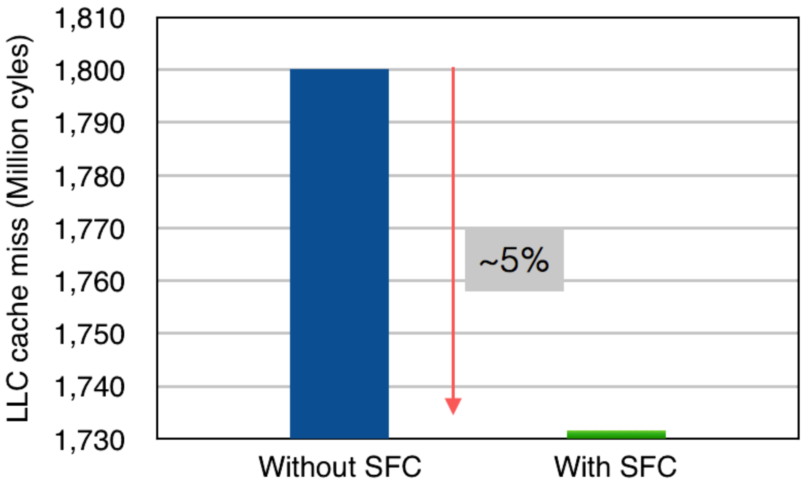


Figure 8.25: LLC Miss with and without SFC

The reduction in CPU time and Last Level Cache misses are shown in Figures 8.24 and 8.25 respectively. There is a significant decrease in computational time and Last Level Cache misses hinting at the reduction in unstructured random access when SFC reordering is adopted. Extrapolating this performance increment for simulations which can run for a week, SFC reordering can lead to ≈12 hours of reduction in computational time.

On the other hand, improvements in the computational performance cannot be considered to be groundbreaking or remarkable. The reason is that SFC reordering was used as a preprocessor only whereby ≈30-35 % of the reordered elements are active per time step. The main point here is that mesh data reordering has the significant impact on computational performance but dynamically generating cache efficient mesh data layouts of active elements every few time steps and performing particle sorting according to these reordered elements might improve performance remarkably.

8.3. Conclusion

This chapter started with a brief introduction to the explicit Material Point Method formulation, its stability criteria followed by a compact presentation of the explicit MPM 3D solver, Anura3D and additions made to it to incorporate mesh reordering mechanism. Visualization of data traversal with and without Morton-order SFC ordering gave the overall picture of comparison of chaotic, random access data traversal with blocked or cache oblivious data access pattern. That visualisation helped to understand the reason behind the birth of a vast number of unstructured memory accesses and a plausible solution to improvise it. The simulation of dam collapse showed deformation of the material continuum or material particles under a load of gravity, within the background grid leading to frequent changes in the set of active elements.

Performance analysis showed that reordering the underlying grid leads to an improved particle-grid interaction due to greater utilisation of the cache hierarchy. The conclusion is that the reordering of active elements is more important because active elements frequently change each time step changing connectivity and diminishing the effect of static mesh reordering. Since SFC generation is not expensive, it could be beneficial to dynamically generate SFC ordering on the fly for the array of active elements and also carry out particle sorting to maintain interaction close to optimal interleaved memory access interaction.

9

Analysis of Implicit MPM 3D Simulation

9.1. Introduction

This chapter explains the concept of implicit MPM algorithm in detail and discusses some code blocks of open source MPM code Kratos [4] which was used here for numerical experiments.

9.1.1. Newmark-Beta Implicit Time Integration Scheme

The Newmark-beta scheme is a time integration scheme for differential equations, and it is widely used for dynamical advancement of structures and solids.

The implicit time integration scheme used here is a unconditionally stable [16] and allows for larger time steps as opposed to explicit time integration schemes. The Newmark time integration scheme is a possible choice for implicit time stepping and it updates the velocity $\dot{u}_{iI}^{n+1}$ and the position u_{iI}^{n+1} at t^{n+1} as follows.

$$\dot{u}_{iI}^{n+1} = \dot{u}_{iI}^{n} + \Delta t[(1-\gamma)\ddot{u}_{iI}^{n} + \gamma\ddot{u}_{iI}^{n+1}] \tag{9.1}$$

$$u_{iI}^{n+1} = u_{iI}^{n} + \Delta t\dot{u}_{iI}^{n} + \frac{\Delta t^2}{2}[(1-2\beta)\ddot{u}_{iI}^{n} + 2\beta\ddot{u}_{iI}^{n+1}] \tag{9.2}$$

where $\Delta t = t^{n+1} - t^n$.

9.1.2. Newton's Method to Solve NonLinear System

Following Chapter 3 of [16] (9.1) can be manipulated to formulate acceleration $\ddot{u}_{iI}^{n+1}$ as following :

$$\begin{aligned} \ddot{u}_{iI}^{n+1} &= \frac{1}{\beta\Delta t^2}(u_{iI}^{n+1} - \tilde{u}_{iI}^{n}) \\ \tilde{u}_{iI}^{n} &= u_{iI}^{n} + \Delta t\dot{u}_{iI}^{n} + \frac{\Delta t^2}{2}(1-2\beta)\ddot{u}_{iI}^{n} \end{aligned} \tag{9.3}$$

Using (9.3) in the nodal equation of motion $m_{IJ}\ddot{u}_{iI}^{n+1} = f_{iI}^{ext,n+1} - f_{iI}^{int,n+1}$ at time step t^{n+1} explained in Chapter 6 results in the nonlinear problem :

$$\begin{aligned} \frac{1}{\beta\Delta t^2}\mathbf{M}(\mathbf{c}^{n+1} - \tilde{\mathbf{c}}^{n}) &= \mathbf{f}^{ext,n+1} - \mathbf{f}^{int,n+1}(\mathbf{c}^{n+1}) \\ \mathbf{R}^{n+1}(\mathbf{c}^{n+1}) = \frac{1}{\beta\Delta t^2}\mathbf{M}(\mathbf{c}^{n+1} - \tilde{\mathbf{c}}^{n}) &- \mathbf{f}^{ext,n+1} + \mathbf{f}^{int,n+1}(c^{n+1}) = 0 \end{aligned} \tag{9.4}$$

where $\mathbf{M}$ is the consistent mass matrix and $\mathbf{c}^{n+1}$ is vector of displacements or degrees of freedom. Newton's method linearises nonlinear problem $\mathbf{R}^{n+1}(\mathbf{c}^{n+1}) = 0$ by starting with initial

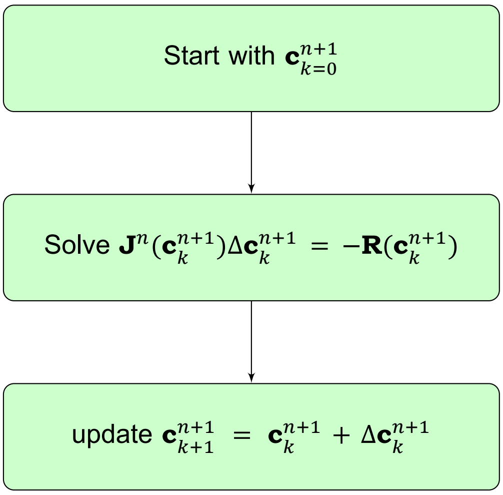

Start with $\mathbf{c}_{k=0}^{n+1}$
Solve $\mathbf{J}^n(\mathbf{c}_k^{n+1})\Delta\mathbf{c}_k^{n+1} = -\mathbf{R}(\mathbf{c}_k^{n+1})$
update $\mathbf{c}_{k+1}^{n+1} = \mathbf{c}_k^{n+1} + \Delta\mathbf{c}_k^{n+1}$

Figure 9.1: Newton's Iteration

guess $\mathbf{c}_k^{n+1}$ with $k = 0$ and updating $\mathbf{c}_k^{n+1}$ until $\mathbf{R}^{n+1}(\mathbf{c}_k^{n+1})$ is less than some specified tolerance. The single step of Newton's method is explained in Figure 9.1 where $\mathbf{J}^n$ is the Jacobian matrix assembled at n^{th} time step and is described latter.

The Newton's method for MPM context will be discussed in the following steps. Since initial guess c_k^{n+1} will not satisfy the original (9.4) so the resultant equation with $\mathbf{c}_k^{n+1}$ becomes

$$\mathbf{R}(c_k^{n+1}) = \frac{1}{\beta\Delta t^2}\mathbf{M}(c_k^{n+1} - \bar{c}^n) - \mathbf{f}^{ext,n+1} + \mathbf{f}^{int,n+1}(c_k^{n+1}) \neq 0 \tag{9.5}$$

and c_{k+1}^{n+1} is updated as follows :

$$\mathbf{c}_{k+1}^{n+1} = \mathbf{c}_k^{n+1} + \Delta\mathbf{c}_k^{n+1} \tag{9.6}$$

To understand the calculation of $\Delta\mathbf{c}_k^{n+1}$, the second part of (9.4) can be expanded with Taylor series to approximate $\mathbf{c}_{k+1}^{n+1}$ as follows :

$$\mathbf{R}(\mathbf{c}_{k+1}^{n+1}) = \mathbf{R}(\mathbf{c}_k^{n+1}) + \frac{\partial\mathbf{R}(\mathbf{c}_k^{n+1})}{\partial\mathbf{c}}\Delta\mathbf{c}_k^{n+1} + \mathcal{O}(\Delta\mathbf{c}_k^{n+1}) = 0 \tag{9.7}$$

Evaluating $\frac{\partial\mathbf{R}(c_k^{n+1})}{\partial c}$ from (9.5) results in the following form

$$\frac{\partial\mathbf{R}(c_k^{n+1})}{\partial c} = \frac{1}{\beta\Delta t^2}\mathbf{M} + \frac{\partial f^{int,n+1}(\mathbf{c}_k^{n+1})}{\partial\mathbf{c}} \tag{9.8}$$

the term $\frac{\partial f^{int,n+1}(\mathbf{c}_k^{n+1})}{\partial\mathbf{c}}$ can be rewritten as $\mathbf{K}^{int}$. Combining (9.6), (9.7) and (9.8) results in

$$\mathbf{J}^{eff}\Delta\mathbf{c}_k^{n+1} = -R(\mathbf{c}_k^{n+1}) \tag{9.9}$$

where

$$\mathbf{J}^{eff} = \frac{1}{\beta\Delta t^2}\mathbf{M} + \mathbf{K}^{int} \tag{9.10}$$

$$\mathbf{K}^{int} = \mathbf{K}^{material} + \mathbf{K}^{geometry} \tag{9.11}$$

Expression for the material stiffness matrix $\mathbf{K}^{material}$ and geometric tangential stiffness matrix $\mathbf{K}^{geometric}$ as follows [8]

$$\mathbf{K}_{IJ}^{material} = \sum_p V_p\mathbf{B}_{Ip}^T\mathbf{D}_p\mathbf{B}_{Jp} \tag{9.12}$$

$$\mathbf{K}_{IJ}^{geometric} = H_{IJ}\mathbf{I}_{3X3} \tag{9.13}$$

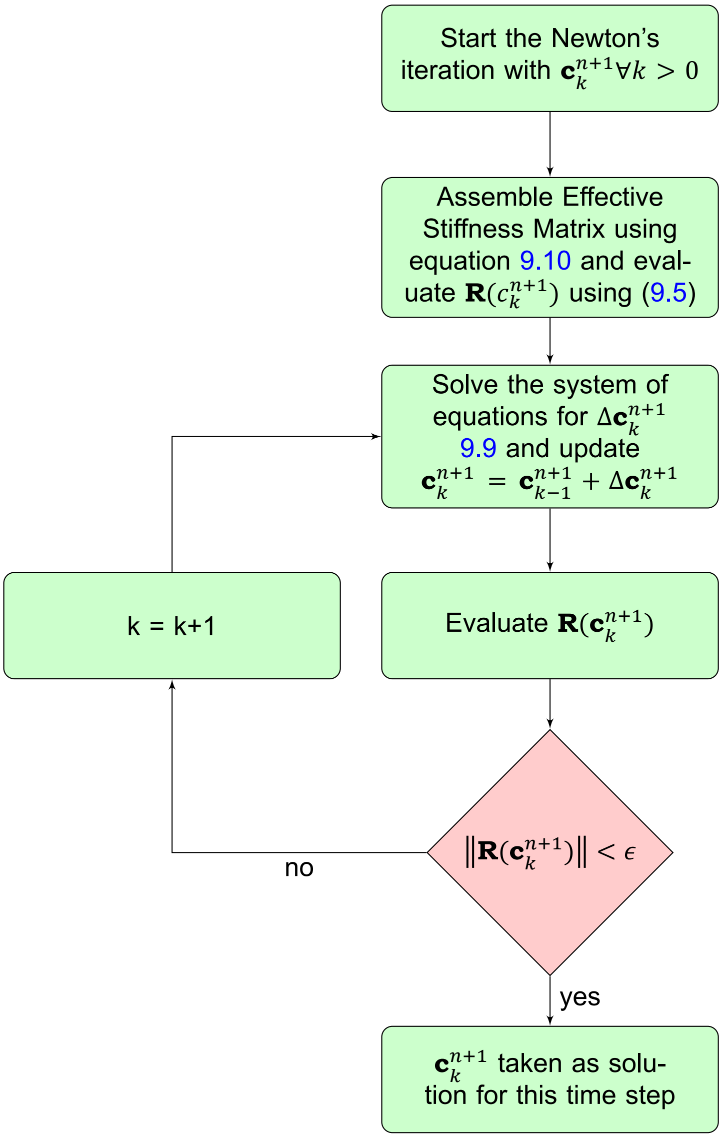

Start the Newton's iteration with $\mathbf{c}_k^{n+1} \forall k > 0$
Assemble Effective Stiffness Matrix using equation 9.10 and evaluate $\mathbf{R}(c_k^{n+1})$ using (9.5)
Solve the system of equations for $\Delta\mathbf{c}_k^{n+1}$ 9.9 and update $\mathbf{c}_k^{n+1} = \mathbf{c}_{k-1}^{n+1} + \Delta\mathbf{c}_k^{n+1}$
k = k+1
Evaluate $\mathbf{R}(\mathbf{c}_k^{n+1})$
$\|\mathbf{R}(\mathbf{c}_k^{n+1})\| < \epsilon$
no
yes
$\mathbf{c}_k^{n+1}$ taken as solution for this time step

Figure 9.2: Newton's iteration in Implicit MPM algorithm

where V_p represents the volume of particle p, $\mathbf{D}_p$ denotes the elastic plastic tensor in Voigt notation, $\mathbf{B}_{Ip}$ denote gradient of the I^{th} shape function at particle p and

$$H_{IJ} = \sum_p (\frac{\partial N_I}{\partial x})^T \sigma_p \frac{\partial N_J}{\partial x} \tag{9.14}$$

is a scalar quantity.

System (9.9) is solved for $\Delta\mathbf{c}_k^{n+1}$ using either direct or iterative methods and then $\mathbf{c}_{k+1}^{n+1}$ is updated using (9.6) and convergence criteria is evaluated. The above mentioned steps are repeated until $\|R(\mathbf{c}_{k+1}^{n+1})\| < \epsilon$ and the final solution $\mathbf{c}_k^{n+1}$ is used to update velocity $\dot{u}_{iI}^{n+1}$ and position u_{iI}^{n+1} at t^{n+1} by using equations (9.1) and (9.2). The overall solution procedure for Newton's method is illustrated in Figure 9.2.

9.1.3. Focus of Study in Implicit MPM Algorithm

The four major steps in the Implicit MPM algorithm are illustrated in Figure 9.3. Performance critical parts are particle-grid interaction and Newton's method to advance nodal properties

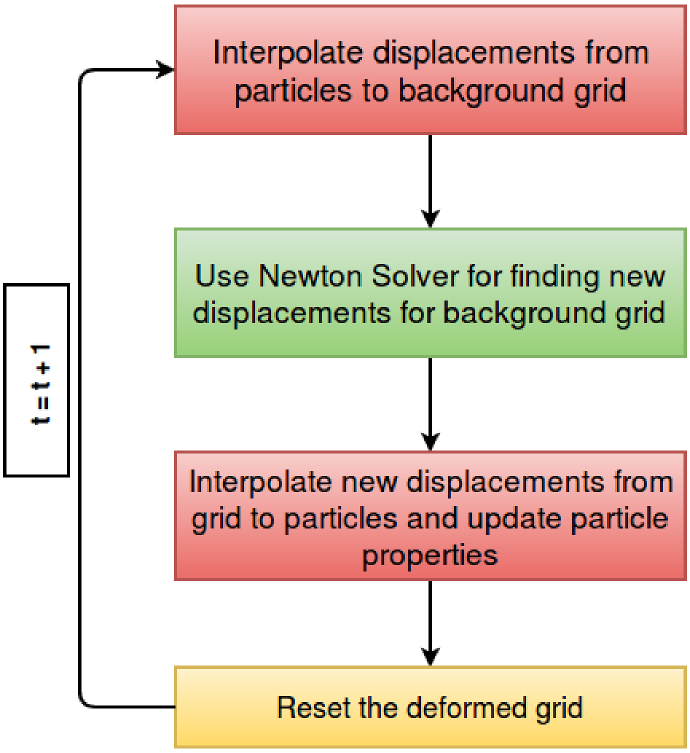

Interpolate displacements from particles to background grid
Use Newton Solver for finding new displacements for background grid
Interpolate new displacements from grid to particles and update particle properties
Reset the deformed grid
t = t + 1

Figure 9.3: Steps in Implicit MPM algorithm

in time which makes it computationally more expensive than the explicit MPM algorithm. The nonlinear solver in each iteration involves solving linear system of the form $Ax = b$ whereby the sparsity pattern of matrix A has a significant impact on overall performance. In this chapter, we want to understand the performance impact of SFC reordering of the elements and vertices of the background grid on particle-grid interaction and the nonlinear solver. Since Morton-order SFC reordering has the positive impact on the structure of sparse symmetric FE matrix which has been studied in Chapter 5, this analysis will help us to understand the unifying impact of SFC ordering on improving cache utilisation for both particle-grid interaction and linear solver. The reason mentioned above makes this analysis different from explicit MPM from the perspective of computational performance.

9.1.4. Implicit MPM Solver

This section gives an introduction to the implicit MPM 3D solver. Kratos [1] is a software framework for multi-physics finite element analysis developed by the International Center for Numerical Methods in Engineering (CIMNE) [2]. Kratos is an open-source object oriented framework for building finite element based applications. The Python language is used as an interface to the core functionalities which are written in C++ which significantly improves flexibility for both user and developer ends. The Particle Mechanics application in Kratos is dedicated to implicit Material Point Method development and analysis. The structure of the MPM application embedded within Kratos is shown in Figure 9.4.

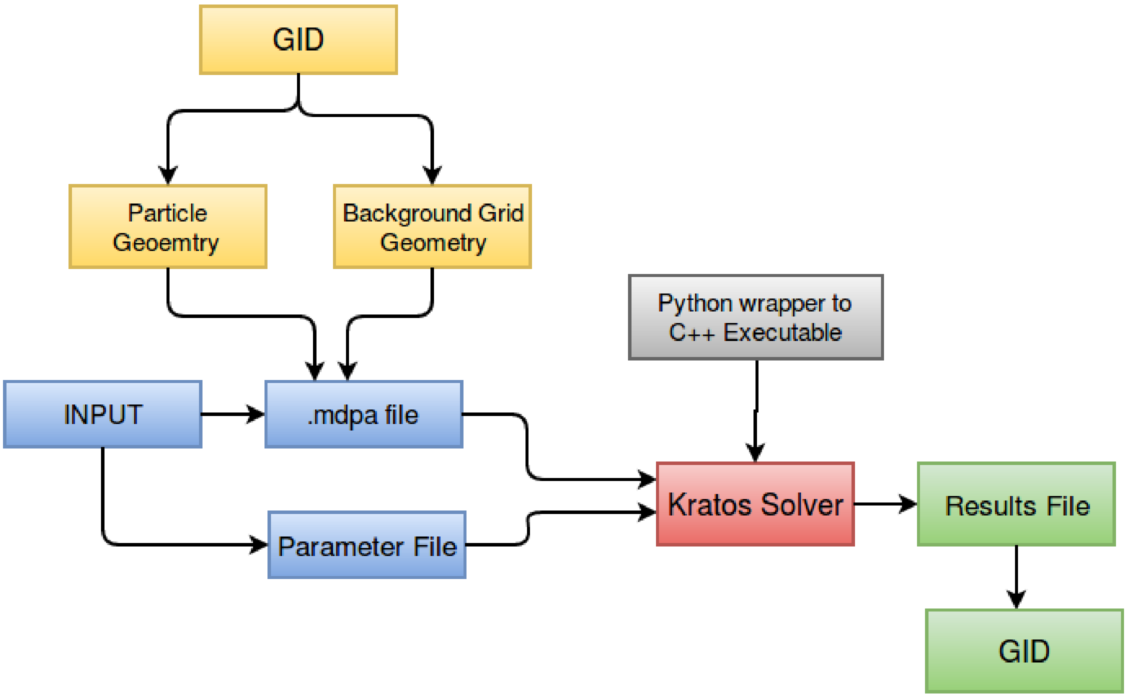


Figure 9.4: Implicit MPM solver

The main phase of simulation with Kratos is summarized as follows and then explained in detail below.

1. Creation of input data (pre-processing with GID [3]).
2. Calculation with Kratos Solver.
3. Output files and visualization with GID (post-processing).

Creation of Input Data

The most recent version of the MPM application in Kratos supports input files from the GID pre-processing tool. Background geometry and material geometry are constructed separately in GID and then meshed. The GID pre-processor generates .mdpa files which along with the parameter file which act as an input to the core Kratos MPM Solver. The parameter file is a Python file containing details about simulation parameters and essential information required by the solver.

[1] https://github.com/KratosMultiphysics/Kratos/wiki/Kratos-For-Dummies
[2] http://www.cimne.com/
[3] https://www.gidhome.com/

Next, objects to material geometry and background geometry are initialized with information of their respective grids. Object to the set of Material points is created using Gauss points from the Finite Element mesh of the material grid object. The object to material points is forced to shared nodes with object of the background mesh and material grid object is deleted.

9.2. Performance Analysis on Dam Collapse Simulation

This section discusses the performance impact of Morton-order SFC reordering of the background grid on the implicit MPM scheme with test case simulation. This section describes the methodology, test case geometry, simulation details and also presents data traversal visualisation of the background grid and relevant results to assess the ability of mesh data reordering on computational performance of a dam collapse simulation.

9.2.1. Test Case Geometry, Boundary Condition and Simulation Parameters

This section describes test case geometry with dimensions, discretization of background grid, particle discretization of the material, simulation parameters and material model.

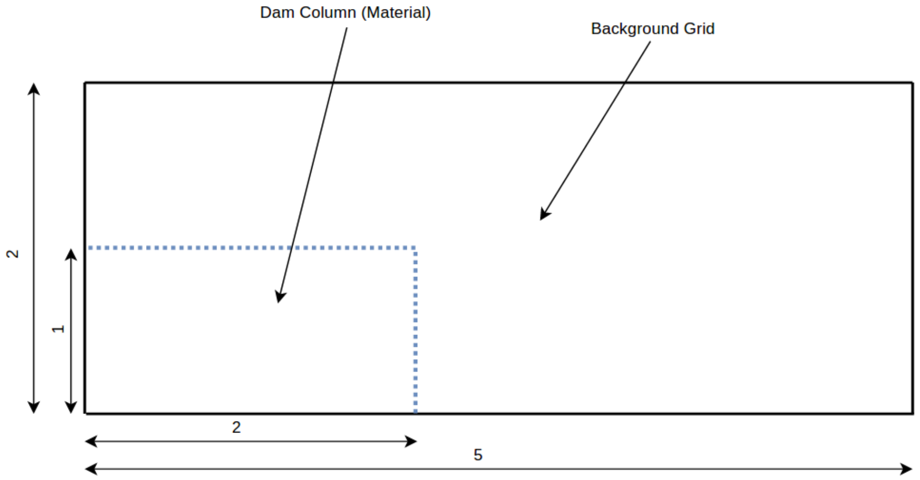
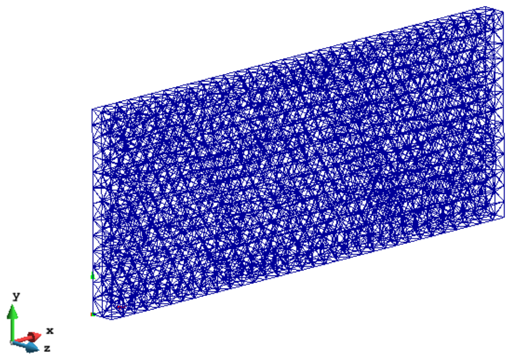


Figure 9.5: Illustrative projection of dam collapse geometry with dimensions



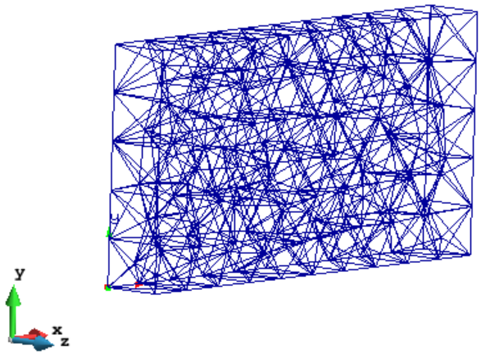


Figure 9.6: Background Grid Discretization

Figure 9.7: Material continuum Discretization

Test Case Geometry

Figure 9.5 explains a two dimensional XY projection of the three dimensional geometry with depth of 0.25 unit. Figures 9.6 and 9.7 show individual finite element discretizations of the background grid and the material continuum. Tables 9.1 and 9.2 present details of background mesh and the material mesh respectively.

Table 9.1: Mesh Properties of the background mesh

Number of Tetrahedra Elements	6755
Number of Nodes	1756

Table 9.2: Material discretization

Number of Tetrahedra Elements	673
Number of Nodes	200
Number of Particles per element	4

Boundary Condition

A displacement boundary condition is applied on all faces by forcing the normal displacements at all nodes on the boundary to be zero.

$$\mathbf{u}_{node}^{normal} = 0, \forall nodes \in \partial\Omega \tag{9.15}$$

Table 9.3: Load condition

Gravity(z direction)	-9.8 m/s2

Simulation Parameter and Material Model

The simulation parameters and material model description are shown in Tables 9.4 and 9.5. The direct solver ”Super LU” is used to solve the linear systems of equation within Newton’s method. A central difference explicit integration scheme is used to advance particle properties in time and space.

9.2.2. Methodology

This section briefly describes important steps taken to simulate a dam collapse with the implicit MPM solver. Figure 9.8 represents bird’s eye view of the changes made to the existing code design for analysis. It is clear that mesh reordering with Morton-order SFC is performed on the underlying grid and then supplied as an input to the solver for numerical calculations. The machine used for simulation here for performance analysis is same as used in Chapter 8.

Tools Used for Simulation

Perf, a linux tool described earlier in chapter 3 is used for system monitoring and measurements. It analyses hardware counters supported by Intel architectures to calculate various performance indexes which can be used for analysis. This chapter makes use of cache-misses, a hardware event supported by Intel architectures and also by Perf to observe utilization of cache hierarchy with and without use of mesh reordering.

Table 9.4: Simulation parameters

Simulation Type	3D Implicit MPM
Solver Type	Dynamic
Solution Method	Newton Raphson
Time Step size	0.001
Number of Time Steps	1000
Explicit Integration Scheme	Central Differences
Convergence Criteria	Residual Based
Residual absolute tolerance	10^{-9}
Linear Equation Solver	Super LU

Table 9.5: Material Properties

Material Type	1-phase solid
Constitutive Law	HyperelasticplasticJ2
Density	1000 Kg/m3
Young’s Modulus	1000000
Poisson’s Ratio	0.25
Yield Stress	700
Isotropic Hardening Modulus	1000

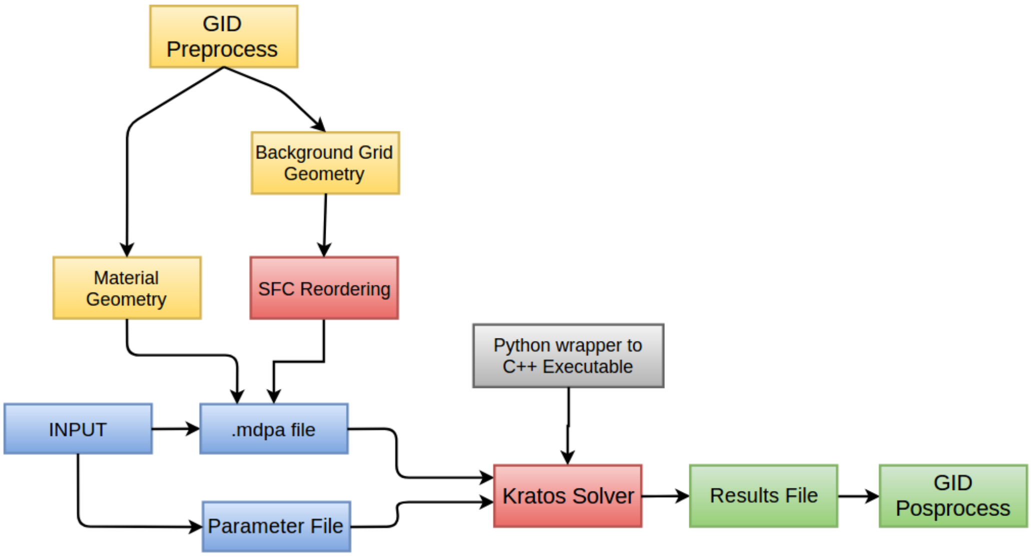


Figure 9.8: Space Filling curve preprocessing tool inserted in schematic diagram of Kratos Simulation Pipeline

Background Mesh Data Traversal with Natural Ordering

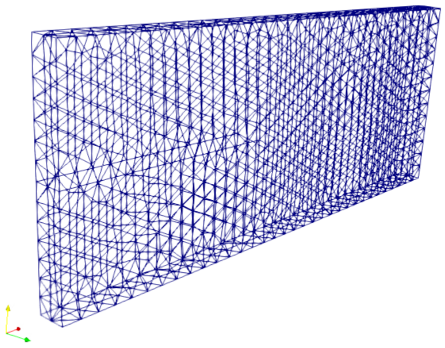


Figure 9.9: 0 Elements Travelled

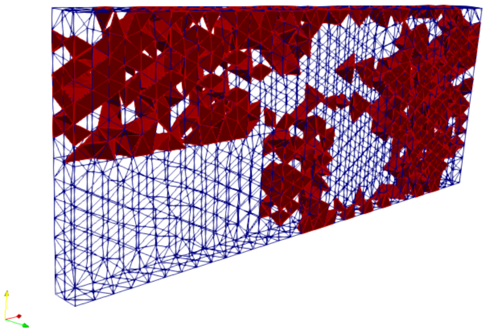


Figure 9.10: 1300 Elements Travelled

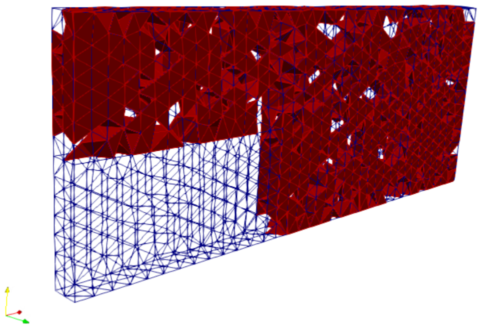


Figure 9.11: 2600 Elements Travelled

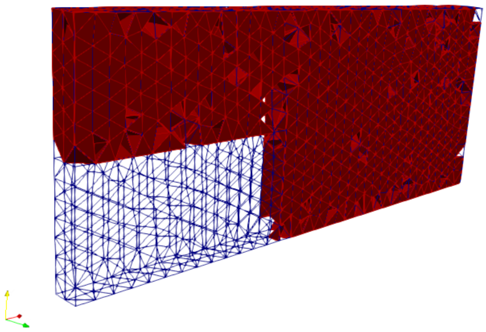


Figure 9.12: 3900 Elements Travelled

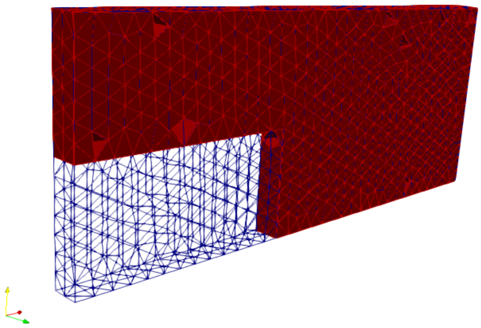


Figure 9.13: 5300 Elements Travelled

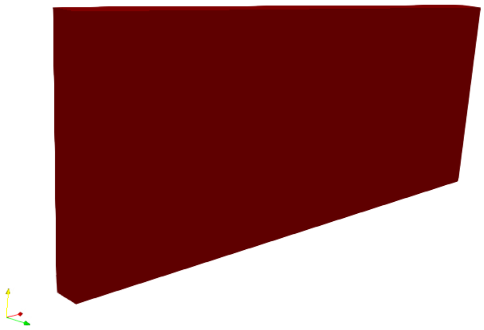


Figure 9.14: 6755 Elements Travelled

Background Mesh Data Traversal with Space Filling Curve

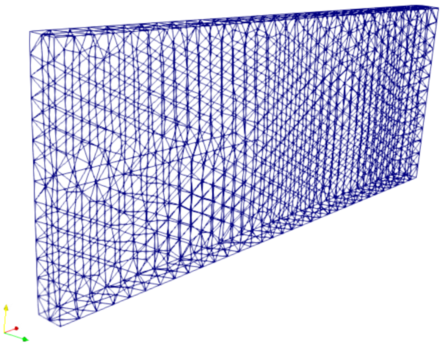


Figure 9.15: 0 Elements Travelled

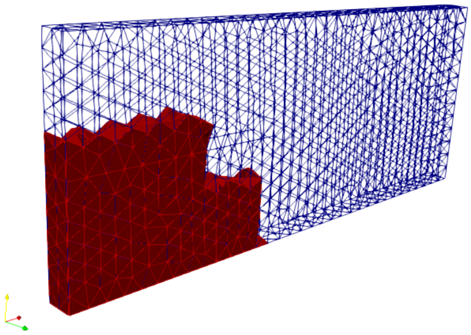


Figure 9.16: 1300 Elements Travelled

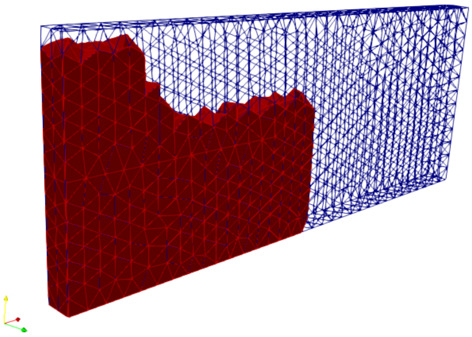


Figure 9.17: 2600 Elements Travelled

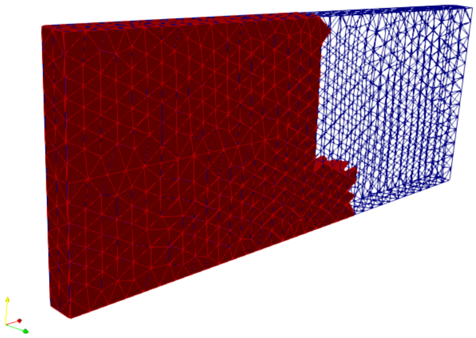


Figure 9.18: 3900 Elements Travelled

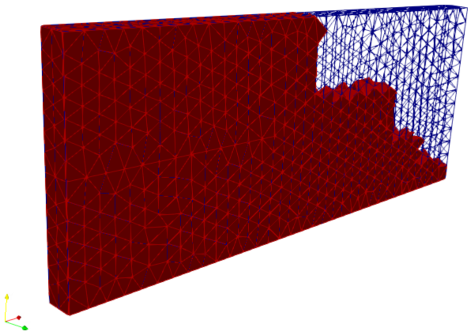


Figure 9.19: 5300 Elements Travelled

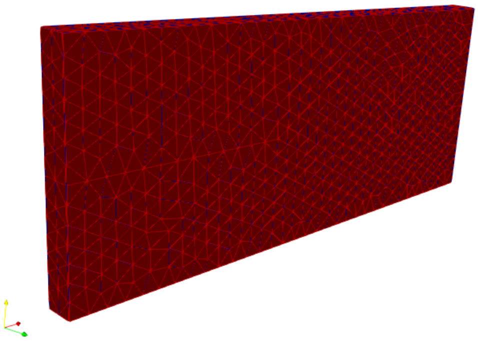


Figure 9.20: 6755 Elements Travelled

Simulation of Dam Collapse for 750 time steps

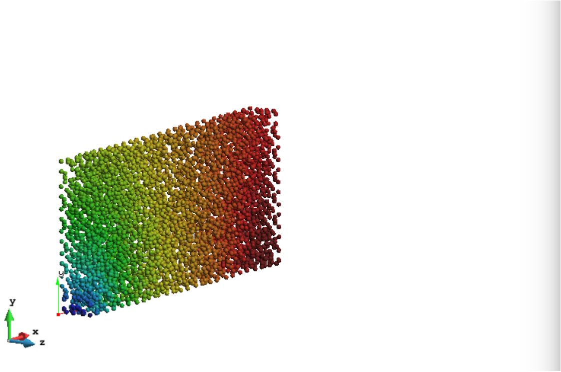


Figure 9.21: Time Step = 0

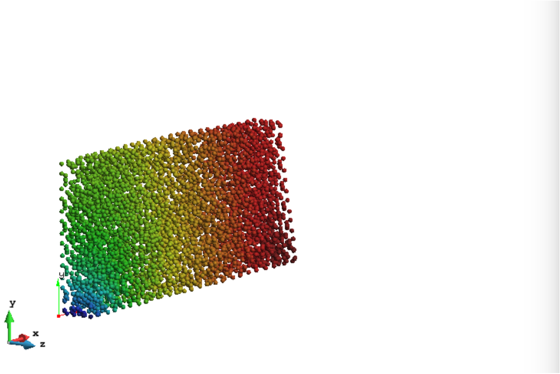


Figure 9.22: Time Step = 150

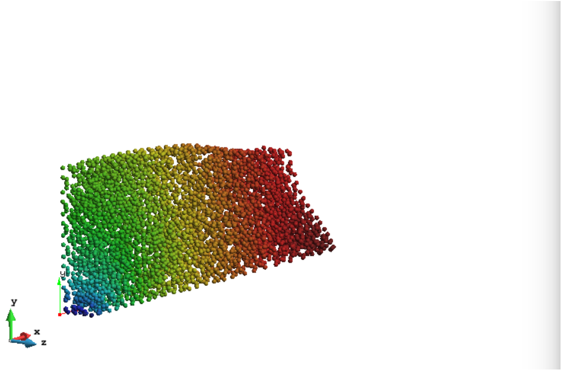


Figure 9.23: Time Step = 300

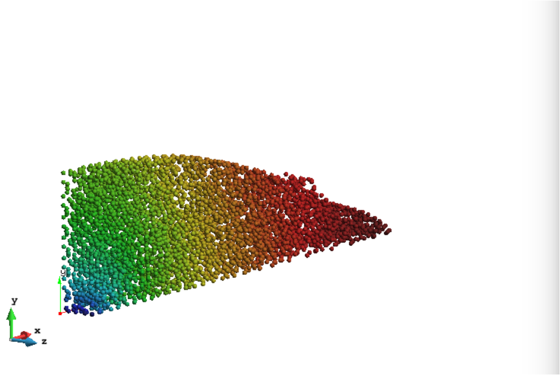


Figure 9.24: Time Step = 450

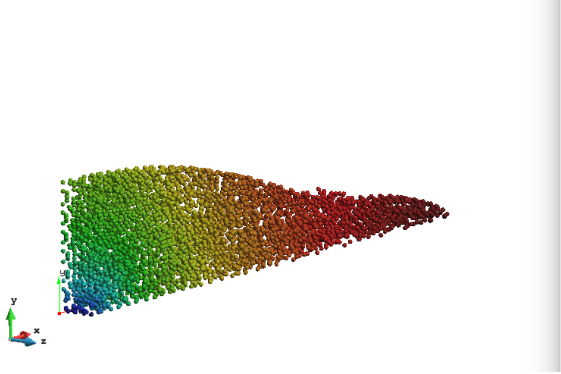


Figure 9.25: Time Step = 600

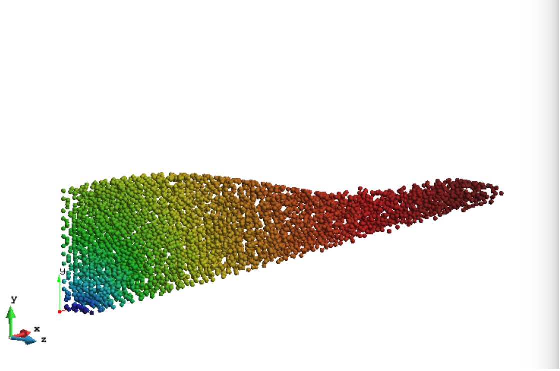


Figure 9.26: Time Step = 750

9.2.3. Results and Discussions

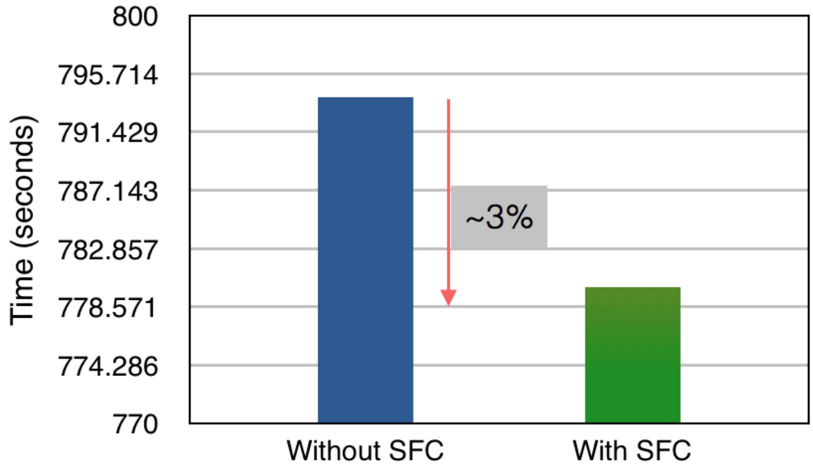


Figure 9.27: Total CPU for 1000 time steps

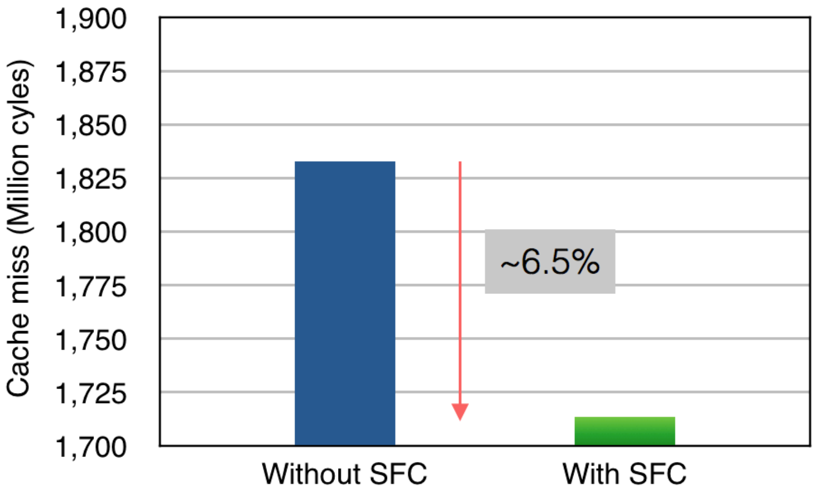


Figure 9.28: Cache Miss in million cycles

Data Traversal Visualization and Dam Collapse Simulation

Data traversal visualization of the underlying grid with natural ordering is shown in Figures 9.9 to 9.14 and SFC ordering is shown in Figures 9.15 to 9.20 respectively. The appearance and distribution of colours demonstrate memory access pattern while traversing the underlying grid.

Simulation of dam collapse is shown in Figures 9.21 to 9.26 under the natural load of gravity acting in the negative z direction. The material flows as expected and occupies only ≈20-25 % of the background grid at all time steps.

Computational Performance

The improvement in computational time and cache utilisation is shown in Figures 9.27 and 9.28. There is a reduction in CPU time by ≈3 %, and reduction in cache misses by ≈6.5 %.

The reason for the small improvement in computational performance is the occupancy of only $(1/5)^{th}$ of all elements of the background grid making it difficult for static mesh reordering to show its full potential.

9.3. Conclusion

This chapter started with the mathematical description of the components of the implicit MPM algorithm followed by description of the test case. The chapter explained the potential importance of reordering of the underlying grid on particle-grid interaction and nonlinear solver. Data traversal visualisation was presented with an emphasis on the source of bad memory access pattern. The chapter also presented dam collapse simulation by implicit MPM technique.

The focus of this chapter was to understand the impact of SFC reordering for the underlying grid on cache utilisation especially when the nonlinear system is solved in addition to the particle-grid interaction kernel. It is concluded that ordering the underlying grid with SFC in practice leads to improvement in cache reuse although the improvement is far from substantial in this case. Much better results in terms of computational time and cache misses, were expected. The possible reason for low performance can be because :

The movement of particles within the background grid matters a lot and affects the memory access pattern since it dynamically changes connectivity in the underlying memory inhibiting coalesced memory access pattern and changing matrix structure associated with Newton's method every time step. The crucial point is that since material continuum does not occupy the entire background grid, static ordering of the background grid results in the formation of clusters of active elements scattered far away in memory. This complex situation makes it difficult to utilise the caches efficiently, and on the other hand, particles are not sorted in memory as they change their spatial position thereby incurring a lot of cache miss and hence increased time. The another possible reason for non-significant improvement can be small problem size whose memory footprint is not much greater than Last Level Cache. Therefore cache oblivious ordering does not help in reducing cache misses significantly. The constraint of single-core computing leaves us with no choice but qualitatively understand effect of SFC ordering.

10

Summary, Conclusions and Future Work

10.1. Summary

This thesis started with the theoretical understanding of single and multi-core processor configuration in chapter 2. We discussed trends in the development of modern micro-architectures and examined challenges for achieving high performance on single and multicore machines. The reason for domination of multi-core architecture in the hardware industry is presented along with reviewing of additional problems involved on the programmer's side. The second chapter concluded with highlighting the primary idea of ensuring data locality and thread pinning on Non-uniform Memory Access (NUMA) machines to achieve optimal performance.

Chapter 3 presented an experimental investigation of a modified stream-benchmarks on NUMA machine. The aim was to understand the performance bottlenecks and develop strategies to reduce the overall computational time and the energy consumption. In this chapter, we investigated the performance impact of different data traversal schemes on bandwidth-bound and compute-bound kernels. The central message was that the data traversal influences the code performance to a large extent and should be designed to utilise the underlying hardware efficiently.

In chapter 4 we introduced the concept of Space Filling Curves (SFC). This chapter first provided the mathematical basis and explained the construction of Morton-order SFC for arbitrary two and three-dimensional meshes. The unique part of this master's thesis was the visualisations of data traversal on the arbitrary grids. The visualisations intuitively explained irregular and cache oblivious data access pattern. These types of visualisations helped us to understand the reason behind the inefficient utilisation of the cache hierarchy and ways to improve it. The takeaway message from this chapter is that the SFC is a cheap and efficient mesh reordering approach to improve cache utilisation for numerical methods relying on grid objects.

Chapter 5 provided an analysis of the impact of Morton-order SFC reordering on the performance of Finite Element solver for Helmholtz equation. This chapter contained a performance analysis of matrix assembly, sparse matrix-vector multiplication and LU factorization and discussed the sparsity patterns of the assembled global stiffness matrix resulting from different reordering techniques. The analysis in this chapter proceeded with concerning data locality and data access pattern and was categorised into: CPU time and energy efficiency, cache utilisation and efficient use of memory subsystems. The main learning point in this chapter was that although many similar algorithms exist for cache-aware mesh reordering, the SFC approach offers the unique combination of inexpensive construction and robust functionality close to optimal cache-oblivious layout.

The mathematical background of the Material Point Method (MPM) and a short introduction to the complex memory access pattern in particle-grid interaction were described in Chapters 6 and 7. Chapter 6 covered the basics of solid mechanics and its connection to the central equation in the MPM. It explained the Lagrangian and the Eulerian steps in MPM computational cycle. Chapter 7 went deep into the ideological understanding of the complex memory access pattern on the performance of the MPM code. The central message was that a naive MPM implementation can be vulnerable to inefficient utilisation of the cache hierarchy and therefore SFC reordering of THE background grid and sorting of particles every few computational cycles is required to improvise it.

This master's thesis finally ended with the performance analysis of an explicit and an implicit MPM solver in Chapters 8 and 9 respectively. The central part of the analysis was to measure the performance impact of reordering elements of the underlying grid on cache utilisation. The SFC reordering was included as a preprocessing step and results were analysed concerning computational time and number of cache misses. In both chapters, the results showed that although a cache oblivious layout of the background grid did not exceptionally improve computational time, the dynamic generation of SFC reordering and particle sorting could improve performance to much greater extent.

Altogether it can be concluded that the data access pattern affects performance significantly. There is a need to understand and identify bottlenecks in these data access patterns and implement algorithms with minimal overhead to improve effective utilisation of memory subsystems. Since energy efficiency is also a concern, in this master's thesis computational time and energy consumption both have been given equal attention and analysed at different instances.

10.2. Conclusions

Six open-ended research objectives were posed in section 1.3 of Chapter 1, and we will conclude by mentioning briefly about each of them with reference to the particular Chapters.

Chapters 2 and 3 were focussed on research objectives 1-3. Data movement between the main memory and the CPU on modern NUMA machine was investigated to identify generic performance bottlenecks concerning floating point performance and memory bandwidth. The strategies adopted there could be used in wide variety of numerical algorithm for high-performance computing. Chapter 4 was intended to complete objective 5 to choose and understand advanced data traversal scheme. SFC as a reordering approach was chosen and it was learnt from this chapter that, SFC reordering is a grid based based cheap and robust alternative to existing state of art reordering algorithms. In this chapter, we understood the concrete reason of improvement in cache utilisation by visualising grid traversal by SFCs.

Chapters 5, 6 and 7 helped in accomplishing objective 4. The complex memory access pattern in the advanced numerical methods were learnt in these chapters. The performance analysis of Finite Element solver with SFC ordering approach sets an expectation which is beneficial to assess its feasibility. Finally Chapters 8 and 9 explores objective 6. In this chapters, we look briefly into the structure of two MPM softwares and analyse performance impact of SFC reordering. We conclude that both particle-grid interaction and Newton's method to solve the nonlinear implicit problem can benefit from the blocking pattern of SFC reordering approach. This blocking pattern of SFC ordering helped improving cache utilisation significantly, but a lot can be achieved by improving parallel performance of both Anura3D and Kratos-Particle-Mechanics part to enable simulation of interesting and bigger problems on small scale servers efficiently.

10.3. Future Work

In this section, we highlight two key directions for the future work. The first direction is the acceleration of the particle-grid interaction in explicit MPM on NUMA machine. In MPM not all elements of the background are active at all times, so reordering only the active elements every few computational cycles can make a huge difference. Although SFC construction supports parallel behaviour, the future work should assess the feasibility of dynamic construction of mesh reordering and particle sorting every few computational cycles in Anura3D.

The second direction can be the implementation of explicit MPM algorithm along with grid reordering and particle sorting on Graphical Processing Units (GPUs). It might be interesting to observe the performance impact of Space Filling Curve reordering on the computational performance of particle-grid interaction since reordering will support optimal use of shared memory and encourage coalesced memory access to global memory banks in GPUs.

Bibliography

[1] M. J. Aftosmis, M. J. Berger, and S. M. Murman. Applications of space filling curves to cartesian methods for cfd. *NAS Technical Report*, 2004.

[2] Frédéric Alauzet and Adrien Loseille. On the use of space filling curves for parallel anisotropic mesh adaptation. *Proceedings of the 18th International Meshing Roundtable*, pages 337–357, 2009. doi: 10.1007/978-3-642-04319-2_20.

[3] P. M. Campbell, J. E. Flaherty K. D. Devine, L. G. Gervasio, and J. D. Teresco†. Dynamic load balancing using space filling curves. *A Technical Report*, 2003.

[4] Pooyan Dadvand, Riccardo Rossi, and Eugenio Onate. An object oriented environment for developing finite element codes for multi-disciplinary applications. *Archives of Computational Methods in Engineering*, 17:253–297, 2010. doi: 10.1007/s11831-010-9045-2.

[5] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. *International symposium on Low power electronics and design*, pages 189–194, 2010. doi: 10.1145/1840845.1840883.

[6] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. 2012.

[7] Shankar Sastry et al. Mesh vertex and element reordering techniques for improved cache utilization in parallel mesh warping algorithms. *Engineers with Computers*, 30:535–547, 2014. doi: 10.1007/s00366-014-0363-0.

[8] Z. Chen et al. A frictional contact algorithm for implicit material point method. *Computer Methods in Applied Mechanics and Engineering*, 321:124–144, 2017. doi: 10.1016/j.cma.2017.04.006.

[9] Daniel F. Harlacher, Harald Klimach, Sabine Roller, Christian Siebert, and Felix Wolf. Dynamic load balancing for unstructured meshes with space filling curves. *International Parallel and Distributed Processing Symposium Workshop*, 2012.

[10] Rolf Niedermeier, Klaus Reinhardt, and Peter Sanders. Towards optimal locality in mesh indexings. *Discrete Applied Mathematics*, 117:211–237, 2002. doi: 10.1016/S0166-218X(00)00326-7.

[11] Leonid Oliker, Xiaoye Li, Parry Husbands, , and Rupak Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *Society for Industrial and Applied Mathematics*, 44:373–393, 2002. doi: doi.org/10.1137/S00361445003820.

[12] Nico Reissman, Jan Christian Meyer, and Magnus Jahre. A study of energy and locality effects using space filling curves. *Parallel and Distributed Processing Symposium Workshop*, 2014. doi: 10.1109/IPDPSW.2014.93.

[13] Hans Sagan. Space filling curves. 1994.

[14] Deborah Sulsky, Shia-Jian Zhou, and H. L. Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications*, 87:236–252, 1995. doi: 10.1016/0010-4655(94)00170-7.

[15] Huy T. Vo, Claudio T. Silva, Luiz F. Scheidegger, and Valerio Pascucc. Simple and efficient mesh layout with space filling curves. *Journal of Graphic tool*, 16:1–15, 2014. doi: 10.1080/2151237X.2012.641828.

[16] Xiong Zhang, Zhen Chen, and Yan Liu. *The Material Point Method*. 2017.