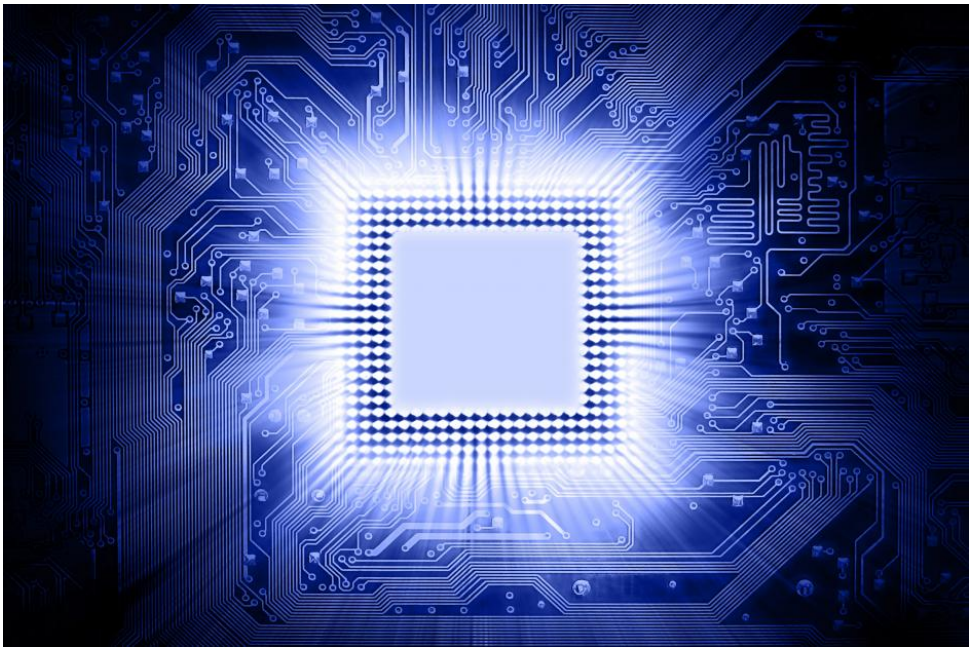
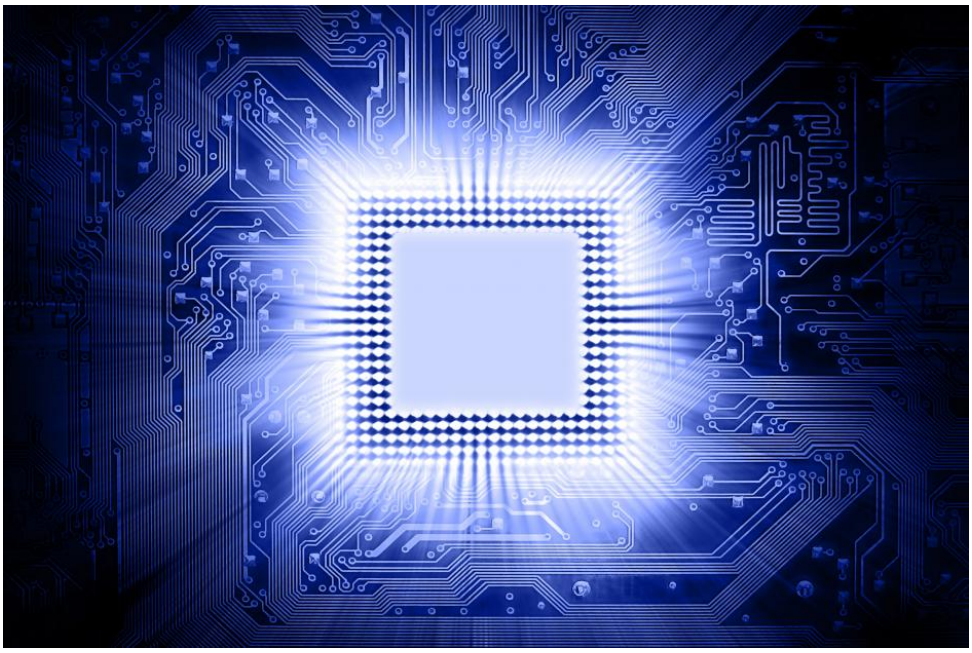
Accelerating the Material Point Method on Emerging Computing Architectures

Literature Study



Accelerating the Material Point Method on Emerging Computing Architectures

Literature Study



Sagar Dolas
4593065

Dr. Matthias Moeller, Technical University of Delft
Dr. Vahid Galavi, Unit Geo-Engineering,Deltares

Contents

Introduction

Simulating fluids and solids undergoing large deformation remains an important and challenging problems in computational Geomechanics. The conventional technique such as finite element method is not sufficiently robust in case of such problems when formulated in the Lagrangian description of motion.

The *material point method* is a sophisticated meshfree numerical technique in which the continuum is represented by *Lagrangian points* called *material points* or *particles*. Large deformation is modeled by particles moving through underlying Eulerian fixed mesh in which particles carry all physical properties of the continuum, whereas the *Eulerian mesh* and its Gauss points carry no permanent information.

The material point method proves to be an efficient and promising numerical technique for simulating large deformation problems in Geotechnical Engineering but poses huge computational challenges for simulations involving a large number of *particles* and degrees of freedom. The Clever and efficient use of data structures and improved data access patterns will help achieve efficient memory management and overall reduction in computational time, which is the main goal of this research work.

Deltares (a Independent Dutch research and consulting organization) is working on simulation techniques for driving of monopiles, steel tubes of diameter up to 11 meters and length of more than 60 meters into the seafloor for construction of offshore wind farms. The currently implemented MPM code is capable of simulating complex soil behaviors, but suffers from huge penalties in terms of data access pattern, data alignment and inefficient use of *threads* / *processes* in shared memory computing environments for almost all computational working loops.

This master's project jointly done with *Deltares* focuses on accelerating currently implemented two phase explicit 3D unstructured finite element MPM code on emerging computing architectures. Achieving scalability and parallel efficiency on ccNUMA machines will be the prime goal of the current work.

1

The Material Point Method

This chapter introduces the material point method as a numerical method to solve problems involving large deformations. Firstly, a brief overview of the material point method is provided. In the following sections, the basic algorithmic structure of the material point method is explained in detail after which each step in MPM is discussed. Finally, this chapter concludes with challenges encountered in parallelization of the material point method and overall overview of the literature survey.

1.1. Introduction

Numerical simulations of large deformation problems, such as the impact of submarine landslides on pipelines, driving monopiles into the ground are technically the challenging problems in computational science. Although widely used traditional finite element method has been successfully solving majority of problems of scientific and academic interest, they are not suitable for problems involving large deformation since the Lagrangian mesh moves and distorts with the material causing mesh distortions and mesh tangling.

In Eulerian formulations the governing equations are solved on the fixed grids, and the material moves through the mesh, which helps Eulerian description to deal with highly distorted motion. There are also some mixed methods which combine the advantages of these two descriptions and avoid their disadvantages such as Arbitrary Lagrangian-Eulerian method. The Material Point Method was born as a simple ALE method but with focus on problems in solid mechanics.

1.2. MPM Functioning

The material point method is used to solve the partial differential equations that describes deformation process of continuum bodies. A continuum is represented as a set of material points or particles which store all physical properties of the material such as mass, velocity and stresses. The solution of the partial differential

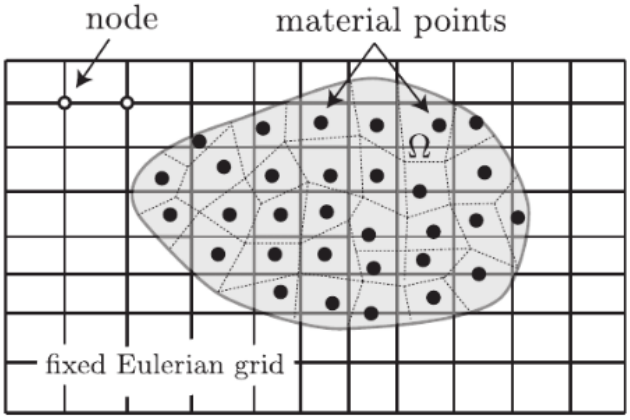


Figure 1.1: Material Points laid over fixed Eulerian Grid

equations is obtained at the material points. The material points move through the mesh over time, representing the deforming body. On the background grid, the equations of conservation of momentum is solved in the same way as with the finite element method. The weak form of the conservation of linear momentum is derived by multiplication with a test function and integration over the domain. The solution of this weak form is then approximated by a linear combination of the basis functions, where each basis function is associated with a degree of freedom. The Figure 1.1 illustrates the concept of Material points laid over the fixed Eulerian grid [5].

1.3. Review of MPM algorithm

Each incremental step in the MPM consist of mapping particle attributes to nearby nodes of the background mesh to allow governing equations to be solved. On the background grid, the equation of motion is solved in updated Lagrangian frame. After the Lagrangian calculation, the velocities and accelerations are mapped from element nodes to the particles, and the particle position and field variables are then updated for the next step. Because the background mesh is fixed in space, mesh distortions is avoided. The Figure 1.2 pictorially explains the basic steps of the Material Point Method.

The standard MPM algorithm is described in [3] briefly as follows. In the following equations, subscript i denotes the value of the grid node i, subscript p denotes the value of material point p, and superscripts n and $n+1$ denote the value at time step n and $n+1$, respectively. $S_{ip} = N_i(X_p)$ is the shape function of node i evaluated at particle p, and $G_{ip} = \nabla N_i(X_p)$ is the gradient of the shape function of the node i evaluated at particle p. MPM algorithm can be summarized as follows.

1. Map the particle variables to the grid nodes to establish their momentum

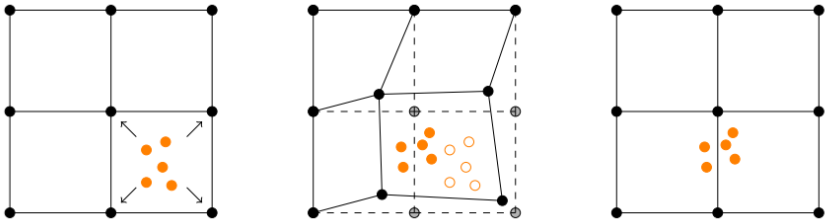


Figure 1.2: Basic steps in MPM Algorithm

equations. The mass of grid node i can be obtained by mapping the mass of those particles located in the cells connected to the grid node i, namely

$$m_i^n = \sum_p m_p S_{ip}^n \tag{1.1}$$

The momentum of the grid node i can be obtained in a similar fashion as

$$p_i^n = \sum_p m_p v_p^n S_{ip}^n \tag{1.2}$$

The force on grid node i can be calculated as

$$f_i^n = (f_i^{int})^n + (f_i^{ext})^n \tag{1.3}$$

where f_i^{int} and f_i^{ext} can be calculated differently.

2. Update the momentum of grid node i using explicit time integration scheme.

$$p_i^{n+1} = p_i^n + f_i^n \Delta t \tag{1.4}$$

both p_i^{n+1} and p_i^n are set to zero on the fixed boundary.

3. Map the nodal results back to particle p to update its position x_p^{n+1} and velocity v_p^{n+1}

$$x_p^{n+1} = x_p^n + \bar{v}_p^{n+1} \Delta t \tag{1.5}$$

$$v_p^{n+1} = v_p^n + a_p^n \Delta t \tag{1.6}$$

where

$$\bar{v}_p^{n+1} = \sum \frac{p_i^{n+1}}{m_i^n} S_{ip}^k \tag{1.7}$$

$$a_p^k = \sum \frac{f_i^n}{m_i^n} S_{ip}^n \tag{1.8}$$

4. Update the particle stress using a constitutive model.

1

1.4. Parallelization of the Material Point Method

Parallelization of the Material Point Method [2] poses huge computational challenge for simulations involving millions of degrees of freedom and thousands of elements. This work takes necessary steps to accelerate MPM on computing architectures.

Historically Parker[7] developed a parallel MPM code using MPI. A parallel computation with 16 millions particles was executed by Parker and co-workers in Massively Parallel Processing (MPP) machines. MPI is an application program interface to design parallel code, which supports distributed memory computing such as nodes and cluster. In a parallel MPM developed by Parker (2006), the computational domain was decomposed into many patches.The information about neighboring patched was exchanged via explicit send and receive operations.

In shared memory machines, parallelization for MPM [1] could be achieved using OpenMP. OpenMP has significant advantage of allowing program to be incrementally parallelized. OpenMP [4] is suitable for shared memory machines such as multi-core computers, symmetric multiprocessing machines and ccNUMA nodes. Although shared memory machines can never compete with distributed memory machines for large scale simulations, the high performance computational environment can be setup by arranging multiple of these SMPs easily and cheaply. Therefore in this work, we will be focusing on adopting some techniques which make MPM code faster and efficient for SMP architecture.

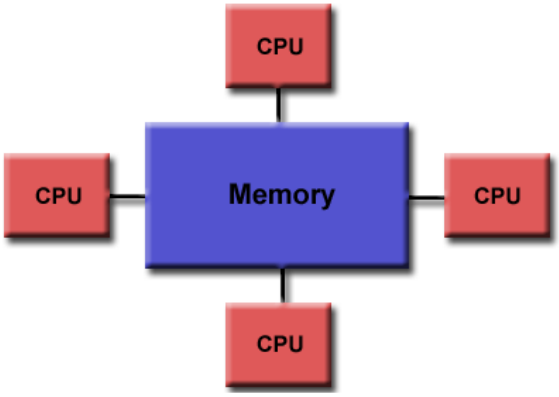


Figure 1.3: Shared memory computing infrastructure

The Figure 1.3 depicts typical shared memory computing environment in which multiple cores share the same memory.

One of the core computational kernels for MPM code is particle to grid interpolation process. The interpolation process requires interpolating nodal data from particles in unstructured grid and vice-versa. These unstructured grid poses significant challenges in terms of computational efficiency due to relative complex nature of unstructured data. This is because, the particles are distributed randomly throughout the mesh, and interpolating the nodal data from particles require random access to an unstructured grid data structure. This random access poses a

significant performance challenge due to inefficient use of caches of modern computing architectures. Managing cache is critical to achieving high performance on cache based machines. A common technique is to partition or block data into pieces that can fit comfortably in as high a level of the cache as possible and is the main theme of the current work.

1.5. Parallelization Strategies for MPM on ccNUMA systems

NUMA or Non-uniform memory access is a shared memory architecture used in today's multiprocessing systems. Each CPU is assigned its own local memory and can access memory from other CPUs in the system. Local memory access provides a low latency - high bandwidth performance, while memory access owned by other CPUs has higher latency and lower bandwidth performance.

Like most every other processor architectural feature, ignorance of NUMA can result in sub-par application memory performance. Fortunately there are steps to be taken to mitigate any NUMA based performance issues or to even use NUMA architecture to the advantage of the parallel application. Figure 1.4 shows typical NUMA computing infrastructure.

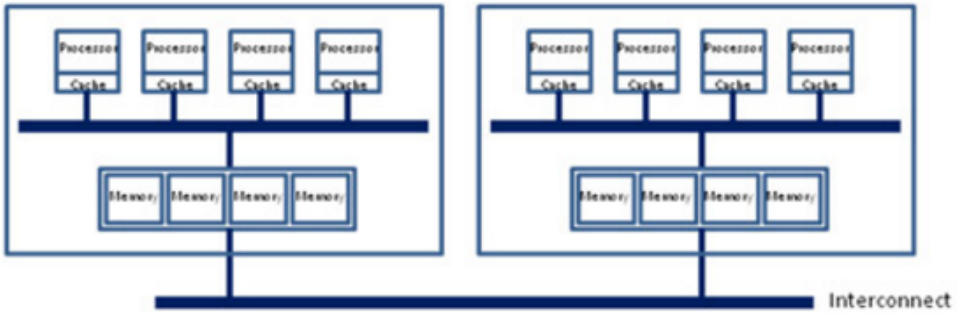


Figure 1.4: NUMA architecture

The most significant advantage of NUMA architecture is a *hierarchical* shared memory scheme to potentially improve *average case access time* through introduction of fast local memory and provide scalable solution for parallel application [6] .

1.6. Anura3D- a 3D unstructured Finite element MPM software

Anura3D is a software tool for MPM analysis developed by the MPM research community. This software is a 3D implementation of the material point method and it is used for simulating the physics involved in soil-water structure interaction and large deformation problems.

1

1.6.1. Calculation process with Anura3D

The process to perform a numerical simulation consists of three parts.

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

The scheme of the procedure is shown in Figure

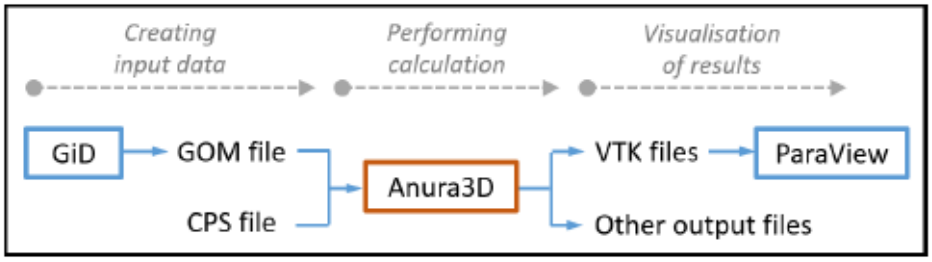


Figure 1.5: Schematic Diagram of Numerical Simulation with Anura3D

1.7. Focus of this research work

Optimization of MPM code on NUMA architecture involves the implementation of clever strategies for improved data access. This research work will focus on the implementation of few of these strategies for the particle to grid interpolation kernel in MPM code and observe its effect on scalability and parallel efficiency.

In this research work, we will be working with ccNUMA machine comprising two NUMA nodes each having 8 cores, each computing core having 32KB of L1 cache, 256 KB of L2 cache, 20 MB of shared L3 cache and 32GB of RAM.

This research work will focus on implementing three strategies for accelerating MPM code in ccNUMA architecture which are described as follows :

1. Implementation of space filling curve for improved data access pattern in unstructured 3D finite element mesh.
2. Incorporation of hyper-threading and thread affinity policies for improved management of computing resources.
3. Incorporation of first touch principle for allocating memory local to each processing core.
4. Use of special abilities of **OpenMP 4.0** for shared memory parallelization in the computational working loops.

In short the main goal of the project work is to make current MPM 3D code scalable and efficient on *Cache Coherent Non-uniform Memory access machines*, so that large problems can be solved easily making judicious use of computational resources.

1.8. Report Overview

Chapter 2 describes in depth about trends in modern computing architectures, explains single and multi-core machines and gives an overview about practicing high performance computing on these machines in an efficient manner.

Chapter 3 lists the elements of parallel computing, and explains about quantifying parallelism and put little focus on benchmarking and performance modeling of the modern computing architectures by sketching **Roofline Model** .

Chapter 4 takes deep dive into cache aware computing according to the space filling curves and various algorithms for generating it efficiently even for large finite element meshes.

Chapter 5 explains the benchmark problems and concludes the kind of analysis to be performed on those problems.

Chapter 6 gives an overview of the **Space Filling Curve toolbox** developed during period of literature survey and explains little bit in depth about it's working.

chapter 7 Discusses results about different benchmarks problem considered in Chapter 5

This document finishes with conclusion and future work provided in the Chapter 8. Mainly, we discuss here about the work completed and strategic plan to be executed in coming couples of month for accelerating ***The Material Point Method*** for modern computing architectures.

References

[1] A. E et al Candel. A massively parallel particle in cell code for simulation of field emitter based electron sources. *Nuclear Instruments and Methods in Physics Research*, 558:154–158, 2006.

[2] Kevin P. R et al. A comparison of parallelization strategies for the material point method. *World Congress on Computational Mechanics*.

[3] P. Huang et al. Shared memory openmp parallelization of explicit mpm and its applications to hypervelocity impact. *CMES*, 38(2):119–147, 2008.

[4] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Using OpenMP : portable shared memory parallel programming*. The MIT Press, 2007.

[5] Vinh Phu Nguyen. Material point method : basics and applications. 2014.

[6] D Ott. Optimizing applications for numa. 2011.

[7] S. G Parker. A component based architecture for parallel multi-physics pde simulation. *Future Generation Computer System*, 22:204–216, 2006.

2

Emerging Computing Architectures

High Performance Computing is the use of supercomputers and parallel processing techniques for solving complex computational problems. HPC technologies focus on developing parallel processing algorithms and systems by incorporating both administration and parallel computational techniques. In this chapter, we will focus on relevant modern trends in high-performance computing, architectural aspects and optimization strategies to be employed for achieving peak performance on these machines.

2.1. Single Processor Computing

In order to write efficient codes, it is very important to understand computer architectures. The difference between two codes that compute the same result can be orders of magnitude depending on how well the algorithm is coded for modern processor architecture. While computers can differ in a number of details, but they also have many aspects in common. On a high level of abstraction, they can all be described as *Von Neumann architectures* .

In scientific computing much attention is paid to movement of floating point data between memory and processing core and can be summarized basically in three instructions.

1. fetch
2. execute
3. store

Modern processing units are more complicated in a sense that, they can execute various instructions simultaneously. This is basically an idea of superscalar CPU architecture and is also referred to as *Instruction Level Parallelism (ILP)*. The main

statistics for CPUs is their Gigahertz rating implying speed of the processor. While speed obviously relates to how fast a processor can operate on data and but second most important issue becomes how fast *useful* data can be brought to the processing core which is described by memory bandwidth. In modern scientific computing, one of the biggest challenges is to provide data hungry processing unit with useful data as fast as possible and this chapter will pay plenty of attention to processes that move data between processor and memory and it's efficient ways.

2.1.1. Modern Processor

Modern processor are quite complicated and this section will give short overview of different parts of modern processor.

Instruction Handling

The *von Neumann architecture* assumes unrealistically that all instructions are handled sequentially. Over last two decades modern processor have used *out of order* instruction handling where instructions can be handled differently than specified by the program only when reordering the instruction leaves the results of the execution unchanged.

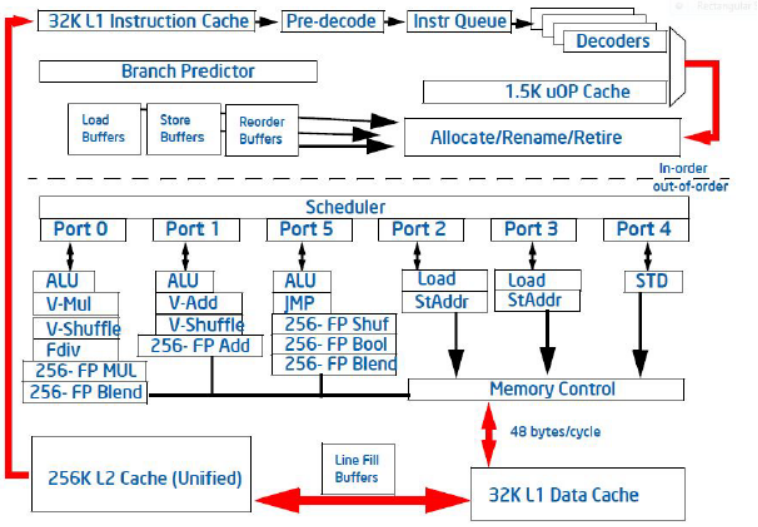


Figure 2.1: Intel Sandy Bridge micro-architecture

In the figure 2.1, we see various units concerned with instruction handling. This cost significant amount of energy and large amount of transistors to carry out **out of order** instruction handling.

Floating Point Unit

In domain of scientific computing, floating point computation is of utmost interest and for this reason, cores have considerable fragility for treatment of numerical data. Earlier, the processors used to have single floating point unit, but these days processors are capable of executing simultaneously multiple instruction per clock cycle.

One of the strategies modern processor use is *Fused Multiply Add* unit, which can execute the instruction $x \leftarrow ax + b$ in the same amount of time as separate addition or multiplication which means nowadays asymptotic speed of several floating point operations per clock cycle can be achieved easily. For some algorithm division operations are a limiting factor. Division operators can take upto 10 or 20 clock cycles, while a modern CPU can have multiple addition or multiplication units that can asymptotically produce same results per cycle.

Pipelining

The floating point add and multiply units of a processor are pipelined, which has the effect that a stream of independent operations can be performed at an asymptotic speed of one result per clock cycle.

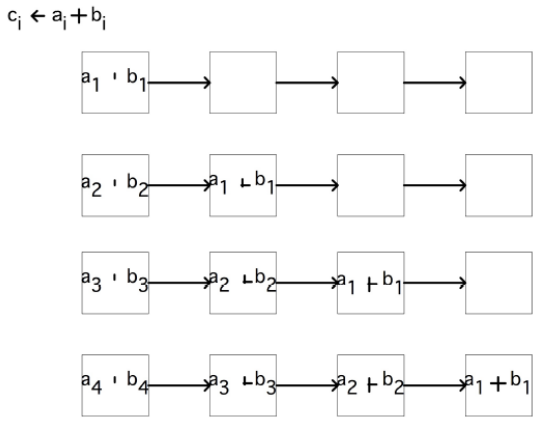


Figure 2.2: Schematic depiction of a pipelined operation

A pipelined processor can speed up the operations by a factor of 4 to 6 times as compared to earlier CPUs, but amount of speed-up one gets from pipelined CPU is limited.

Peak Performance

In modern CPUs there is simple relation between clock speed and the peak performance. Since each FPU can produce one result per cycle asymptotically, the peak performance is clock times the number of independent FPUs. Mostly these days floating point performance is measured in gigaflops, multiples of 10^9 flops.

Superscalar Processor

Superscalar processors analyze several instructions to find data dependencies and execute instructions in parallel. Some of the key points in superscalar processor are :=

1. **multiple issue** : Instructions that are independent of each other can be executed in parallel.
2. **branch prediction and speculative execution** : compiler can guess whether a conditional instruction will evaluate to true and act accordingly.
3. **out of order execution**: instructions can be reordered if they are not dependent on each other.
4. **prefetching** : data can requested before any instruction needing is actually encountered.

2.1.2. Memory Hierarchy

In modern day processor, there are various memory levels between processing core and the main memory, the registers, the caches are together called memory hierarchy.

Buses

These wires are mainly responsible for moving data around the computer, from the memory to the CPU. The most important for us is *Front side bus* which connects the processor to memory. This is distinguished by north and south bridge. The figure 2.3 typically shows bus structure of modern processor.

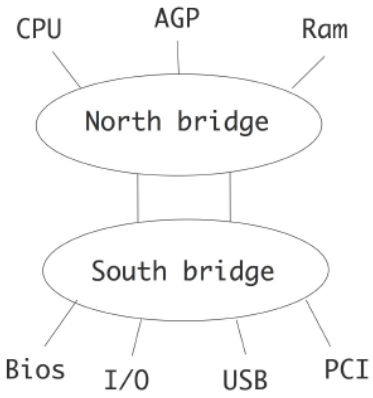


Figure 2.3: Schematic Bus structure of a processor

The bus is typically slower than processor operating with clock frequency of O(1GHz) which is way slower than CPU clock frequency. This is one of main reason that memory hierarchy are needed with the fact that, processor can consume many data items per clock ticks.

2.1.3. Latency and Bandwidth

There are two important concept to describe the movement of data : *latency and bandwidth*. The idea is requesting a data incurs a initial delay, if this item was first in stream of data, the remainder of the stream will arrive with no further delay at regular amount of time.

1. **Latency** : It is the delay between the processor issuing a request for a memory item and the item actually arriving. There are various types of latencies involving transfer from memory to caches, caches to registers and can be summarized into latency between memory and processor. Latency can be measured in nano seconds .

2. **Bandwidth** : Bandwidth is the rate at which memory arrives after the initial latency. Bandwidth is measured in bytes per seconds. It is typically a product of bus speed and bus width : the number of bits that can be simultaneously transferred every clock cycle.

One of the main important issue is to program in such a way that, processor uses data from cache or register avoiding repeated interaction with the main memory.

2.1.4. Registers

Every processor has small amount of memory that is internal to processor specially called **On Chip**, the *registers* , or together the register file. Registers have low latency and high bandwidth because they are part of the processor. Data movement between registers and processing core is instantaneous. Typically processor has 16 or 32 floating point registers. Memory transfer between hierarchies is really expensive and therefore simple optimization would be to keep data is register whenever possible.

There is a limit to how many quantities can be kept in registers, trying to keep too many quantities in register is called register *spill* and lowers the performance of the code. This optimization is typically referred to as compiler optimization.

2.1.5. Caches

The memory where lots of data can reside for longer amount of time are various level of *caches* that have lower latency and high bandwidth where data are kept for an intermediate amount of time.

Data from memory travels through the caches upto the registers. The advantage to having cache memory is that if a data is reused shortly after it was first needed, it will still be in cache, and therefore can be accessed much faster than if it would have to be brought in again from the main memory.

There is important difference between cache memory and registers, while the data is moved into registers by explicit assembly instructions, the move from main memory to cache is entirely done by hardware. Thus cache use and reuse is outside the programmer control.

Cache levels, speed and size

The caches are called *level 1* and *level 2* (short for *L1* and *L2*) cache and some processor also have *L3* cache. The L1 and L2 cache are part of the chip whereas in most recent development L3 is off-chip.The L1 cache is small, typically around 16Kbytes to 32Kbytes but it is much faster. The L2 cache is bigger around hundreds of Kbytes to few Megabytes, but is slower and L3 is more plentiful and also slower.

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

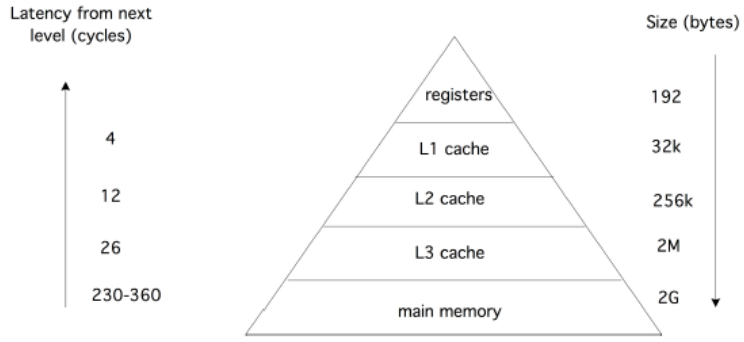


Figure 2.4: Memory Hierarchy of Intel Sandy Bridge

The Figure 2.4 illustrates the basic facts of the cache hierarchy, in this case for Intel Sandy bridge chip.The closer caches to processing core, the faster but also smaller.

1. Loading data from register is so fast that it does not contribute to time limitation for any numerical algorithm.

2. The L1 cache is small, but sustains a bandwidth of 32 bytes per cycle.

3. 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 core of a processor chip, so effectively the bandwidth is fraction of this number. Most clusters will also have more than one socket per node, typically 2 or 4, so some bandwidth is spent of maintaining cache coherence again reducing bandwidth available for each chip / processor core.

We see that the larger caches are unable to provide enough data to the processors. For this reason it is necessary to code in such a way that data is kept as much as possible in the highest cache level for longer period of time.

Types of Cache misses

There are three types of cache misses.

1. **Compulsory miss** : This occurs the first time you give reference to data, and this is unavoidable.
2. **Capacity miss** : The next type of cache misses is due to the size of working set, a *capacity cache miss* is caused by data having been overwritten because the cache simply cannot contain all of the data. If we have to avoid this type of cache miss, we need to partition data into chunks that are small enough that data can stay in cache for appreciable amount of time,assuming that data is operated multiple times.
3. **Conflict miss** : This type of miss occurs when one data gets mapped to same cache location as another, while both are still needed for computation.

Cache Line

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 is 64 bytes or 128 bytes which in context of scientific computing implies 8 or 16 double precision floating point numbers.

It is really very 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 on the cache line, since access to them is effectively free. This phenomenon is visible in codes that accesses arrays by unit stride.This is also one of the key-points of the current work.

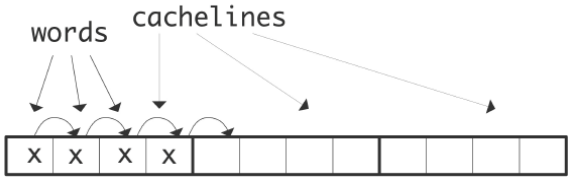


Figure 2.5: Cache block and unit stride access

Reuse is the Key

The presence of one or more caches is not immediately a guarantee for success in high performance computing, and this largely depends on memory access pattern of the code and how well it is able to exploit the cache behavior. The conclusion is there should be an opportunity for an algorithm to have data reuse, which in many problems of scientific computing are inherently presents within numerical formulation of mathematical procedures. Therefore these kinds of numerical algorithm should be coded in such a way to fully and effectively utilize ability of cache for improved floating point performance.

2.2. Multicore architectures

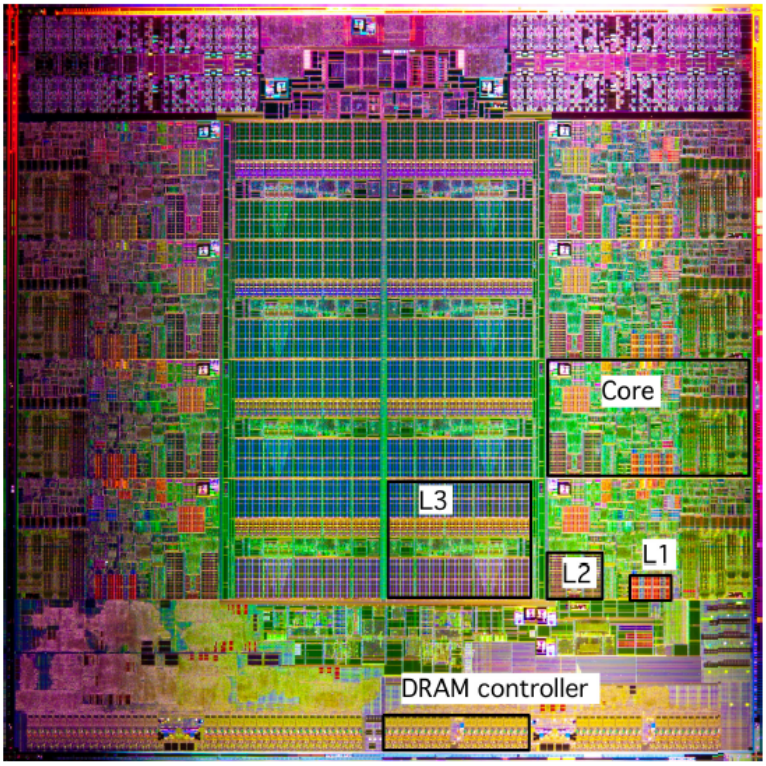


Figure 2.6: A typical multicore chip

In recent years, performance limits have been reached for traditional processors because of the following reasons :

1. Clock frequency can not be increased further due to increased energy consumption, increase in current leakage.

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 utilization of the computing resources is to move from traditional single core architecture to multiple processing cores which also introduces idea of task and data parallelism producing overall higher efficiency because two cores of lower frequency can have the same throughout as a single processor at a higher frequency: hence reducing energy consumption. A typical multi-core chip is shown in figure 2.8

With the mix of shared and private caches, the programming model for multi-core processor is becoming hybrid between shared and distributed memory.

1. **Core** : The core can have their own private L1 and L2 cache and different cores can communicate in distributed computing fashion.
2. **Socket** : On one socket, L3 cache can be shared by multiple cores.
3. **Node** : There can be several socket on one node typically 2-4 accessing shared memory also provided by other sockets.

2.2.1. Node architecture and Sockets

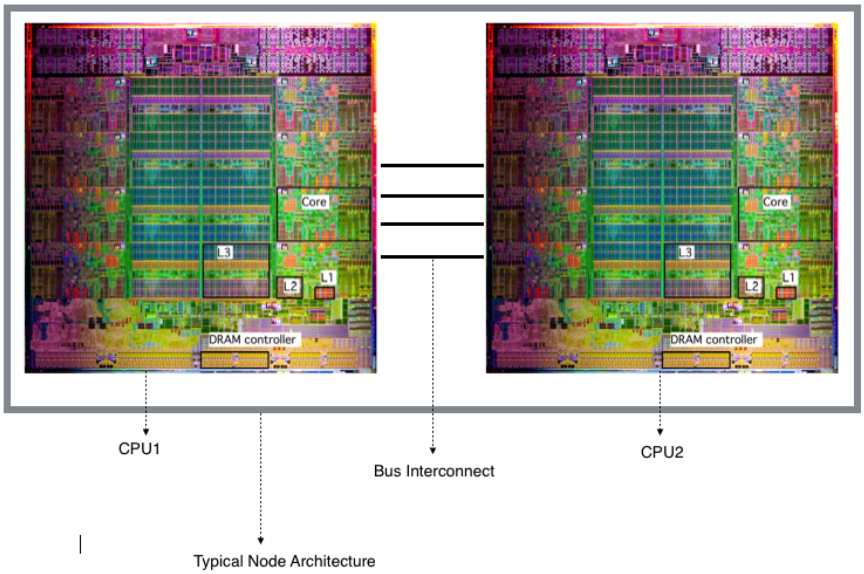


Figure 2.7: Schematic Diagram of Typical Node with 2 sockets

Uniform Memory Access

Uniform memory access is a shared memory architecture used in multicore chips.All the processor core in UMA model share the physical memory uniformly. In a UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data.

With UMA systems, the CPUs are connected via system bus to the Northbridge.The Northbridge contains the memory controller and all communication to and from memory must pass through Northbridge.The I/O controller, responsible for managing I/O to all devices, is connected to the Northbridge.Therefore every I/O has to go through the Northbridge to reach the CPU.

Multiple buses and memory channels are used to double the available bandwidth and reduce the bottleneck of the Northbridge.To increase the memory bandwidth even further some systems connected external memory controllers to the North-

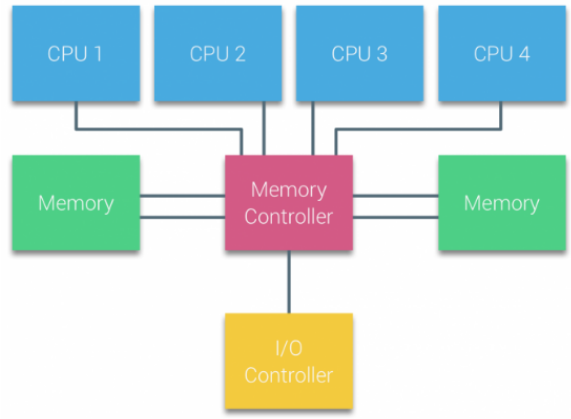


Figure 2.8: Uniform Memory Access Architecture

bridge, however due to internal bandwidth of the Northbridge UMA is considered to have a limited parallel scalability and efficiency.

Non-Uniform Memory Access

In order to improve parallel scalability and efficiency , there were number of changes made to shared memory architecture.

1. Non-Uniform memory access organization.
2. Point to Point Interconnect topology.
3. Scalable cache coherence solutions.

NUMA moves away from a centralized pool of memory and introduces topological properties. By classifying memory location bases on signal path length of the processor to the memory, latency and bandwidth bottlenecks can be avoided.

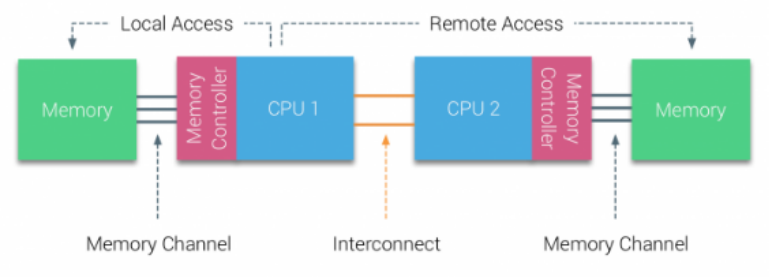


Figure 2.9: Non Uniform Memory Access Architecture

The Figure 2.9 typically explain NUMA organization. Each CPU has its own memory address space. The memory connected to the memory controller of the CPU1 is

considered to be local memory. Memory connected to another CPU socket (CPU2) is considered to be foreign or remote for CPU1. Remote memory access has additional latency overhead to local memory, as it has to traverse an interconnect and connect to the remote memory controller.

Scalability of Cache in NUMA

The intel Sandy bridge Architecture has scalable ring on die interconnect for L3 cache. In this way each core has private path to L3 cache and this allowed intel to partition and distribute L3 cache in equal slices. This provides higher bandwidth and associativity. The Figure 2.10 shows die configuration of typical intel Xeon CPU. Each slice is 2.5MB and every other core is allowed to access other slice as well.

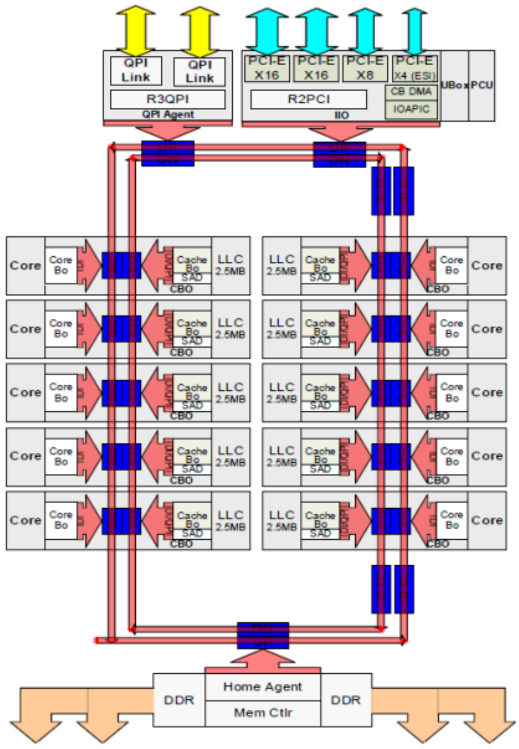


Figure 2.10: Die Configuration of Intel Xeon CPU of Braodwell Microarchitecture v4

The advantage of the NUMA architecture as a hierarchical shared memory scheme is its potential to improve average case access time through the introduction of fast, local memory.

2.2.2. Intel's HyperThreading/ Simultaneous Multi-threading

Hyper-threading or SMT is a term used by Intel to describe their technology which makes a single CPU core appear to the operating system as two CPU cores. The

operating system thus treats the core like it would treat any multi-core by sending it multiple threads simultaneously. Essentially Intel has duplicated highly used portions of the CPU core and allowed these sections to be used by separate threads simultaneously. These hyper threading capable cores are not exactly the same as multi-cores; nor just any any thread can be executed simultaneously with another thread, it has to use a separate part of the core for it's operation.

Advantages of Hyper-threading

Intel claims that the duplication of the certain sections of the CPU core increase the size of the core by about 5 percent, while giving a performance boost by 30 percent.

Disadvantages of Hyper-threading

One disadvantage in many application is the high power consumption of hyper-threaded core and indeed all SMT cores. Furthermore comparing hyper-threaded CPU core with a non hyper-threaded CPU one sees more effect of cache thrashing in hyper-threaded core.

2.2.3. Performance Issues in NUMA machines

Cache Coherency

With respect to parallel processing which we will discuss in next chapter, there is huge potential for conflict if multiple processing core has the same copy of data in in its cache. The problem of ensuring that all cached data are an accurate copy of main memory is referred to as *cache coherence*.

Suppose two cores have a copy of the same data in their private L1 cache, and one modifies its copy. Now other has cached data which is no longer validated or has inaccurate copy of the counterpart : the processor will invalidate the copy of the item, and in fact the whole cacheline.

This process of updating or invalidating cachelines is known as maintaining *cache coherence*, and it is done on a very low level of the processor, with no programmer involvement needed. However, it will slow down the computation, and it wastes bandwidth that would otherwise be used for loading or storing the operands.

False sharing

The cache coherence problem can even occur if the core accesses different items, but they fall on the same cacheline. The most common case of false sharing happens when processing cores update consecutive location of an array.

While there is no actual data race condition, the code having above characteristics will have low performance since the cacheline continuously will be invalidated.

Remote memory access

What gives NUMA its name is the memory access time varies with the location of the data to be accessed, if data resides in local memory to the processing core, the access is fast and if data resides in remote memory the access will be sub-optimal.

While access to local memory involves only 10 of instruction cycles, remote memory access involves 100-130 instruction cycles which is way inefficient and should be avoided for optimal performance.

Remote memory access can also occur, if while allocating memory master thread allocates on one node and workers threads accessing it from other node creating lot of penalties.

Thread migration

Today's complex operating system assigns application threads to processor core using scheduler. A scheduler will take into account system state and various policy objectives before assigning applications threads to different physical cores. A given threads will execute for some period of time before being swapped out of the core to wait as other threads are given chance to execute.

Thread migration from one core to another poses a problem for NUMA shared memory architecture because of the way it disassociates a threads from its local memory allocations causing remote memory access and significant increase in total computational time.

2.2.4. Optimization for NUMA machines

Like most other architectural features, ignorance of NUMA can result in sub par application memory performance. Fortunately there are steps which can be taken to mitigate any NUMA based performance issues.

Data Placement using implicit memory allocation policies

In simple case, many operating systems transparently provide support for NUMA friendly data placement. When a single threaded application allocates memory, the processor will simply assign memory pages to the physical memory associated with the requesting thread's node(CPU Package).

Some operating system will wait for the first memory access before committing on memory page assignment. The advantage here is multi-threaded application will be benefited since memory pages will be effectively placed local to worker threads that will access the data.Summarizing operating system will observe the first access request and commit page assignments based on the requester's node location.

The two policies *local to first access and local to first request* together depict the very importance of programmer knowledge of underlying operating system's policy.Memory allocation should be done in such a way that, it should be local to threads accessing it rather than local to master or control thread because multiple threads accessing the same data are best co-located on same node so as to avoid sub-optimal remote memory access and allow NUMA system to provide characteristic performance speed-up.

Processor affinity

Processor affinity is the persistence of associating a thread/process with particular resource instance, despite availability of other instances. It is the programmer's responsibility to decide whether processor affinity solutions are right for particular

applications because restricting scheduler options may significantly harm the system performance where better resource management could have been adopted. Exercising processor affinity ensures memory allocations remain local to the threads that need them.

The key issue in determining whether NUMA architecture could be realized for high performance computing is ***data placement***. The more often that data can be effectively be placed in memory local to the processing core, the more overall access time will be benefited. Conversely, the more data fails to be local to the node that will access it, the more memory performance will suffer from the architecture.

Generally in NUMA model, the time required to retrieve data from an adjacent node within the NUMA model will be significantly higher than that required to access local memory. Shortly as the distance from a processor increases, the cost of accessing memory increases.

References

[1] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. The Saylor Foundation, 2016.

3

Parallel Computing

Parallel computing is a type of computing in which several processors execute or process an application simultaneously. Parallel computing helps in performing large computations by dividing the workload into several chunks which can individually be processed by more than one processor. This chapter, we describe elements of parallel computing, its concepts, parallel programming paradigm and performance models.

3.1. Functional Parallelism vs Data Parallelism

Data parallelism is applying similar kind of independent operations on many data elements. This is the common scenario in scientific computing and it stems from the fact that the data set can be spread over many processors and each working on its part of the data.

It is also possible to find independence, not based on data elements but based on the instructions themselves such that independent instructions can be given to separate floating point unit or reordered for example to optimize register usage. This is the case of functional parallelism.

3.2. Shared memory computing vs Distributed memory computing

3.2.1. Shared Memory Computing

A shared memory computer has multiple cores that have access to the same physical memory. If access to the memory locations is equally fast from all cores, the machine is called symmetric multiprocessor or uniform memory access and if multiple chips are involved, access is not necessarily uniform, meaning that from perspective of particular core, some physical memory locations can be accessed with lower latency than others and this situation is called non-uniform memory access.

Programs for shared memory computers typically use multiple threads in the same process. Typically one can use OpenMP directives to parallelize the serial program. Figure 7.3 and Figure 7.4 illustrates two kinds of shared memory organization that is uniform memory access and Non-uniform memory access.

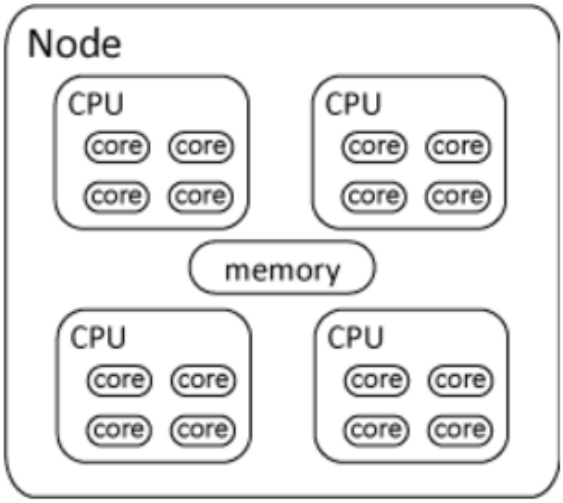


Figure 3.1: Typical Shared Memory machine in NUMA organization

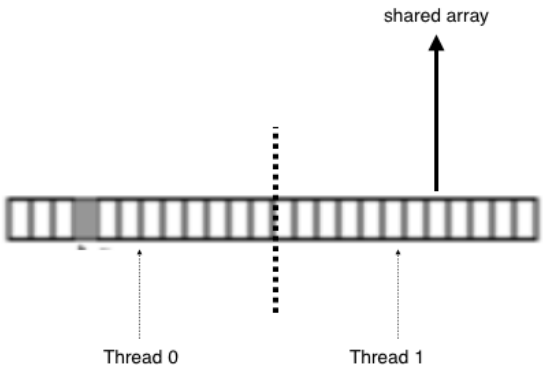


Figure 3.2: Array updating in shared memory environment

Figures 3.1 and 3.2 explains infrastructure and array updating in shared memory environment.

Shared memory computing is the utilization of threads to split up the work in a program into several smaller units that can run parallel within a node. These threads share access to a certain portion of memory- hence the name shared memory computing.

3.2.2. Distributed Memory Computing

The idea of physically distributing processes across a computer cluster results in a new level of complexity when parallelizing problems. Every problem needs to be split into pieces - the data needs to be split and the corresponding tasks need to be distributed as well. To this end, data or information required by other processes will be gathered into chunks that are then exchanged between processes by sending and receiving messages by message passing protocol.

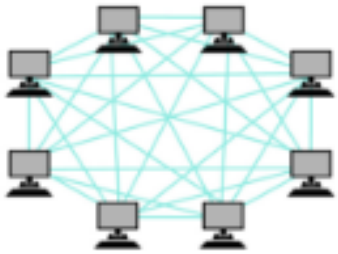


Figure 3.3: Communication network in distributed memory cluster

Distributed memory computing has many advantages :

1. We can add more compute power, either in form of additional cores , sockets,or nodes in a cluster.
2. With every additional node added to the cluster, more memory is available and compute arbitrarily large models can be solved.
3. Scalability of distributed memory system is much higher than shared memory system that is speed-up will saturate at a much larger number of processes compared to number of threads used.

Complexity and amount of communication in parallelizing an application increases as the number of nodes, sockets, cores increases and this create additional burden for optimized parallel programming approach.

3.3. Quantifying Parallelism

There are many reasons to use parallel computers, but the two most important ones are memory gain and higher performance.

3.3.1. Speed Up and Efficiency

Speed Up

A simple way to define speed-up is to compare the time taken by the same program on the serial and the parallel computer. With T_s denoting the serial time and T_p denoting time taken on the parallel computer with p processors, the speed-up S can be defined as

$$S = \frac{T_s}{T_p} \tag{3.1}$$

Ideally the speed-up can be p, but in practice we cannot expect it to attain because of many issues and one of the very important is communication and the other one is synchronization which together are heavy source of inefficiencies.

Embarrassingly Parallel

There are many problems which are solved with minimum or no communication and such programs are very efficient on parallel machines. Such problems, in effect consist of number of completely independent calculations and it will have close to perfect parallel speed-up and efficiency.

Super Linear Speed-up

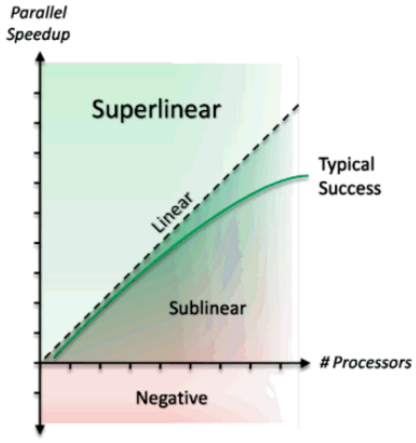


Figure 3.4: Parallel Scalability

It is also possible to achieve super-linear speed in some very specialized cases where single processor computing involved regular interaction with RAM, but multi-core computing properly fits all data into highest level of cache improving its cache behavior and overall scalability.

Parallel Efficiency

An important performance metric for parallel program is parallel efficiency. Parallel efficiency E_p with P as number of processor is defined by

$$E_p = \frac{T_s}{P * T_p} \tag{3.2}$$

where T_s and T_p are defined above.

Amdahl's Law

One of the main reason for less than perfect speed-up is the presence of inherently serial fraction which cannot be parallelized at all. Amdahl's law relate speed-up with serial and parallel fraction of the code. Let's suppose that F_s be the serial fraction of the code and F_p be the parallel portion of the code such that $F_s + F_p = 1$. Then T_p parallel time according to Amdahl's law is

$$T_p = \frac{T_s}{F_s + \frac{F_p}{P}} \tag{3.3}$$

where P is number of processors.

According to Amdahl's law as number of processor grows, the T_p approaches $T_s F_s$. The conclusion is speed-up is limited by $S \leq F_s$ and efficiency is decreasing function of P ;$E \approx \frac{1}{P}$

In a way, Amdahl's law is optimistic way of expressing speed-up. In reality communication overhead will lower the speed up attained.

Amdahl's Law and Hybrid Programming

Suppose we have p nodes with c cores each and F_p describes the fraction of the code that uses c-way thread parallelism. Then the Amdahl's law would describe parallel time as

$$T_p = \frac{T_1(1 + F_s(c-1))}{pc} \tag{3.4}$$

3.4. Strong and Weak Scalability

Splitting a problem over more and more number of processors does not provide any significant benefit since there is just not enough work for each processor.

3.4.1. Strong Scalability

Strong scalability has the same effect as speed-up. Application is scalable if speedup is perfect or near perfect, that is execution time goes down with increasing number of processor for fixed problem size.

3.4.2. Weak Scalability

Weak scalability is more vaguely defined term, since as problem size and number of processor grow in such a way that amount of data per processor stays constant and execution time also stays constant.

3.5. Roofline : Performance Model for Multi-core Architectures

Conventional wisdom in computer architecture produced similar designs, Nearly every desktop and server computer uses caches, pipelining, superscalar instructions, and out of order execution.

3.5.1. Roofline Model

In foreseeable future, off the chip memory bandwidth will be the limiting factor in floating point performance and the model which relates processor performance to the off-chip memory traffic for particular machine is called **Roofline Model**.

Operational Intensity

Operational intensity is the term used to describe operations per byte of DRAM traffic, defining total bytes as transaction between main memory and the processor filtering through cache hierarchy. Thus, operational intensity predicts the *DRAM* bandwidth needed by particular computational kernel on a specific machine. The Roofline model ties together floating point performance, operational intensity, and memory performance in 2D graph.

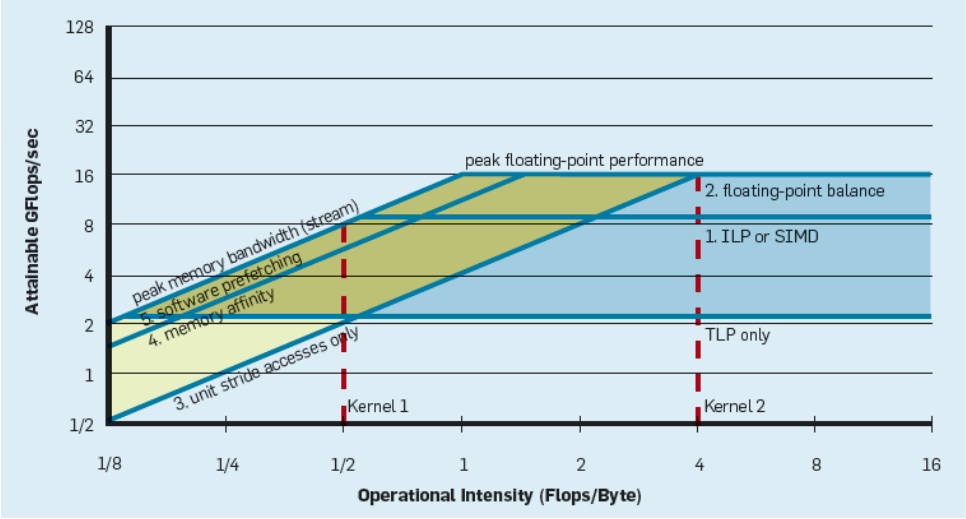


Figure 3.5: A Roofline Model alogn with optimization strategies

The **Roofline** sets an upper bound on performance depending of the kernel's operational intensity. Assuming operational intensity as column that hits the roof, it can hit the flat part depicting the kernel is compute bound otherwise it's memory bound [2]. The Figure 3.5 shows typical *Roofline model* along with the optimization strategies to be adopted individually for compute and memory bound kernels.

References

[1] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. The Saylor Foundation, 2016.

[2] Samuel Williams et al. Roofline : An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4), 2009.

4

Cache Aware Computing and Space Filling Curves

In order to mitigate the impact of the growing gap between CPU speed and the main memory performance, today's computer architectures implement hierarchical memory structures. The idea behind this is to hide both the low memory bandwidth and the latency of main memory accesses which is slow in contrast to the floating point performance. This chapter will focus on some of the techniques to utilize cache behavior tailored for the unstructured finite element code using cache aware space filling curves.

4.1. Space Filling Curves

Space filling curves are mappings from a one dimensional interval to a region in an *n*-dimensional space. Space filling curves pass through every point in the region in the *n*-dimensional space [10] . Since a space filling curve does not intersect itself, an ordering of the points in the *n*-dimensional space can be obtained through its use. Such curves are usually constructed through recursion, but infinite recursive calls are necessary to construct the ideal curve to find the ordering in an *n*-dimensional space. Since in real time scenario, only finite number of points are present, it is possible to number each and every point according to the space filling curve.

The Space filling curve is a continuous object that roughly speaking, maps a higher dimensional space for example R^2 and R^3 onto one dimensional space [10]. Mathematically speaking

$$c : (1, \ldots n)^d \Rightarrow (1, \ldots n^d) \tag{4.1}$$

These curves strongly enjoy the local properties which makes them suitable for many applications in computer science and scientific computing.

In the d dimension, the SFC provides an ordered enumeration of a virtual Cartesian grid of size 2^{dp} where is p is depth of the recursion in which our computational

domain is fully embedded. The index is found by the cube in which entity stands. In the following sections we will briefly outline the calculation of the space filling curve indexes. Figure 4.1 shows a typical space filling curve.

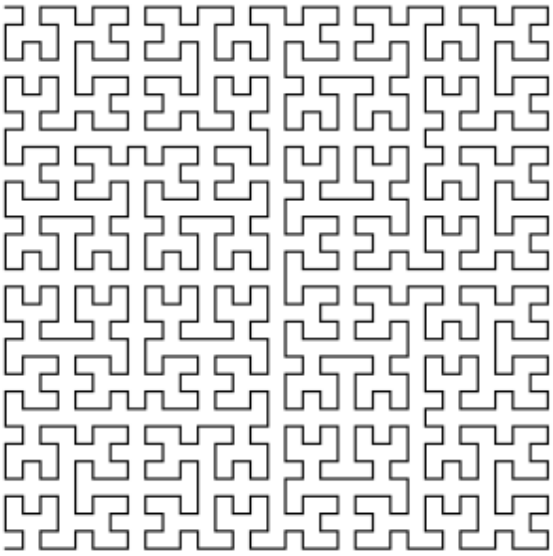


Figure 4.1: Typical Space filling curve

Meshes plays an important role in the numerical solution of the *Material Point Method* in a given geometric domain. Accuracy of the solution strongly depends on the shape and the size of the meshe. With the advent of modern computing and multi-core architectures larger meshes can easily ve solved with reasonable amount of time.

In this research work we use the space filling curves to reorder the elements and vertexes of underlying finite element mesh to exploit the memory architecture in shared memory, multi-core machines. Using space filling curves for reordering can result in improved spatial and temporal locality, therefore improving the data access pattern and cache behavior.

A mesh may be used several times with different boundary conditions and renumbering the mesh elements and vertexes is not additional burden on the solver, since mesh reordering is kind of pre-processing step.

4.2. Cache aware computing using the Space Filling Curve

Mesh renumbering using space filling curve has a significant impact on the efficiency of the code and it is even more influential for the numerical methods discretized on the unstructured meshes.

Cache aware computing for SFC includes several advantages such as :

1. Reducing the number of caches misses during indirect addressing loops.
2. Reducing the cache line overwrites and the problems of cache coherency in the shared memory computing environment specifically for the NUMA nodes.

Cache misses are due to the indirect addressing and they occur when the data required for a computation do not lie in the cache line. For instance, in particle to grid interpolation a loop traversing the elements request access to its vertexes will suffer from constant cache miss due to random numbering of the vertexes. Cache misses are far more costlier than any other operations in numerical context which can be seen below.

1. ***L1 Cache miss*** ≈15 cycles
2. ***L2 Cache miss*** ≈300 cycles

As space filling curves order the points based on their spatial proximity, it is possible to extract spatial locality in their access pattern by ordering these points appropriately. Traversing the ordering created by the space filling curve results in the higher temporal locality specially for finite element meshes.

In this master's project the space filling curves are used to reorder vertexes in a way to favour geometric closeness in the original mesh. In this way they also lie close in memory in such a way that vertexes that are close in the physical domain also lie very close in memory and hence 90 percent of the cache misses can be avoided compared to an unordered meshes.

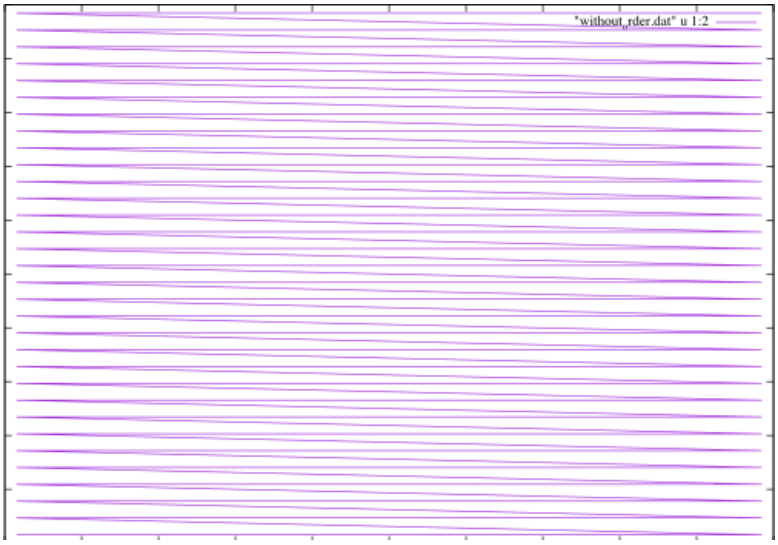


Figure 4.2: Lexicographical Ordering

Figure 4.2 shows naive lexicographical ordering, in which data points which are close geometrically are far away in memory particularly with the stride of one row,

Figure 4.3: Z Order Space filling curve

but Z-order space filling curve shown in Figure 4.3 shows geometrical closeness and closeness in memory of the data points in the computational domain.

4.3. Enhancing Shared Memory Parallelization by Space filling curve

Using meshes and matrices in scientific computing leads inevitably to complex algorithms where indirect memory accesses are needed.Typically in *particle to grid interpolation in Material Point Method* multiple threads spread across the number of elements may want to update the data associated with same vertex and therefore will leads to conflict based updating which is either done by introducing locks which decreases degree of parallelism or by phenomenon of reduction.

Renumbering according to space filling curves becomes *de facto* [4] mandatory when dealing with indirect memory access in such kernels.

4.3.1. Efficient Out of Core Parallelization using SFC

A proper mesh partitioning strategy is one of the crucial task both for shared and distributed memory computing. The critical issue of minimizing synchronization in shared memory and communication in distributed memory remains a challenge. The space filling curve reshapes the mesh into blocks which can either be given to different cores in shared memory environment or different CPUs in distributed memory as shown in Figure 4.3.

The following algorithm can be used for efficient mesh partitioning for multi-core architectures.

Algorithm 1: Efficient Mesh Partitioning for Multi-core architecture

Data: 3D Input Mesh
Result: Mesh Partition using Space filling curve

1. Get the index of each element/ vertex based on space filling curve.
2. Sort the elements to reorder the mesh.
3. Subdivide the sorted list into *p* sublist where *p* can be number of cores in ccNUMA node or can be the number of distributed memory computing processor in a cluster.

This algorithm is extremely fast and consumes only little memory. The partition is equivalent to a $renumbering + sorting$.This algorithm works very well for structured and unstructured finite element meshes.

4.4. Parallel Generation of SFC for 3D unstructured Finite Elements

This section presents simple and efficient algorithm to compute cache friendly layouts of the unstructured 3D finite element mesh. Space filling curves helps to minimize cache misses and page faults by arranging vertices, triangles or tetrahedrons in spatially structured manner. This algorithm provide comparable results to **COML** : optimal cache oblivious mesh layout, and is orders of magnitude faster and is *embarrassingly parallel*.

4.4.1. Mesh Layout in ANURA3D

The mesh representation used in ANURA3D is the element-vertex format where each mesh is represented by two separate arrays ; a vertex array V holding vertex geometries (x, y, z) location and element array E holding vertex indices. Vertex can be shared among different elements having same index in both the element. The figure 4.4 graphically shows the 10 noded 3D element used in current work.

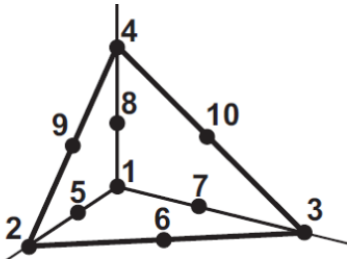


Figure 4.4: Typical 10 noded 3D Element

4.4.2. Morton Space Filling Curve

We have currently implemented *Morton order* (also known as *Z-order*) or Bit - interleaving curve [5] for simplicity and efficiency in implementation and layout computations. This layout is comparable to more sophisticated Hilbert space filling curve [5] which is better in locality preserving, is expensive and complicated for implementation and computation. Z-order curve implementation can easily be extended for large meshes and has been implemented in current research work.

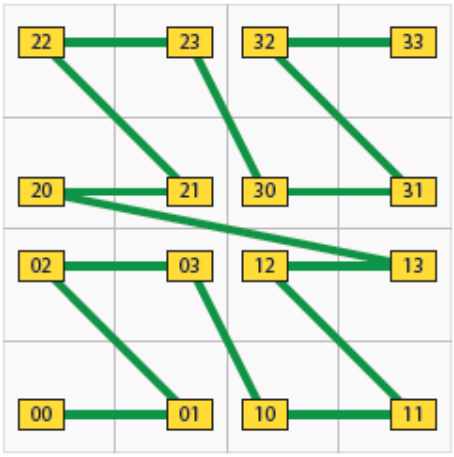


Figure 4.5: Morton order space filling curve ordering in base 4 digits in 2D

Calculating Space filling curve index for the finite element mesh

The location of tetrahedron element is defined to be its centroid. In order to compute Morton-order space filling curve for centroids, an implicit octree of depth N is constructed. The root of the tree is mesh's bounding box and leaf nodes are the cells in $2^N x 2^N x 2^N$ of the regular cartesian grid. The index of centroid is constructed by concatenating N octal digits in the octree, see figure 4.5 for concatenation.

In principle since this is not a recursive implementation, N must be large enough to ensure that each leaf in the octree contains at most one centroid. Since octal digits require exactly three bits, using 64 bit unsigned integer corresponding to choosing $N = 21$ which is equivalent to grid resolution of $(2x10^6)^3$ and accounts for even the largest mesh used here in this master's project.

Algorithm 2 describes about parallel space filling curve generation for arbitrary unstructured 3D finite element mesh. Once, all the elements have their space filling curve indexes vertexes are renumbered in such a way that, if element *A* comes before element *B*, vertexes of element *A* are placed before vertexes of element *B* [7].

Thus, the parallel space filling curve generation involves one pass over all the elements allocating them space filling curve indexes which is linear in time com-

plexity, and using standard sorting algorithm for sorting them in memory and one pass over vertexes to renumber them again according to the elements.

Algorithm 2: Parallel Space filling Generation

Input : Mesh file

1. Find out the centroids of each element.
2. Find out the bounding box, in which our computational domain can be fully embedded. The bounding box can preferably cubic.

```
   /* Initialize the number of levels N                          */
1  in parallel foreach element e in the mesh do
   |  /* Pass the element,bounding box, Number of levels        */
2  |  MortonOrderCurve(e,boundingbox,N) /* O(N)                  */
   end
   /* Sort the Elements according to their space filling
      index                                                      */
3  std::sort(element.begin(),element.end()) /* Quick sort
      algorithm takes O(NlogN)                                   */
```

Algorithm 3 describes about Z-order kernel which gives every element its space filling curve index. This algorithm has significant advantages over the other competitive alternatives. The performance of above the algorithm is comparable to more complex algorithms like one mentioned above. The **Morton Order** layout is extremely easy to compute. Particularly, the finite element datasets that have highly irregular geometric distribution may be improved using such layout [9].

However there are disadvantages associated space filling curve. Since, the ordering is based on the physical coordinates and may give poor results for physical domains with extreme aspect ratios or that are embedded in a higher dimension space e.g spherical shells (oceans) [9] .

In general, mesh construction methods are designed to produce correct meshes rather than meshes with good data locality. Reordering the mesh can improve the data locality such that data for particular element are close in memory. The *first touch* memory policy is to ensure that memory pages are bound to local memory on NUMA architectures, however this does not translate into locality of memory access if the data for physical nodes comprising an element are not located close to each

other in memory.

Algorithm 3: Morton order space filling curve index

Result: Morton Order Index for each element
Input : element, bounding box , number of Levels

```
1  Function:
2  MortonOrder(Element e,BooundingBox box,uint64 N )
3  index ← 0 ;
4  foreach each level upto N do
5      index ← index << 3 ; /* Concatenation with each level   */
       /* Set the octant using Morton order                    */
6      if e.centroidx > box.center.x then
7          index ← index + 1
8      end
9      if e.centroidy > box.center.y then
10         index ← index + 2
11     end
12     if e.centroidz > box.center.z then
13         index ← index + 4
14     end
       /* Update th bounding box to the appropriate octant  */
15     foreach direction x,y,z in bounding box do
16         if e.component > box.center.component then
17             box.min.component = box.center.component
18         else
19             box.max.component = box.center.component
20     end
21 end
```

References

[1] V. K. Decyk. Adaptable particle in cell algorithms for graphical processing units. *Computer Physics Communications*, 182(3):641–648, 2011.

[2] V. K. Decyk. Particle in cell algorithms for emerging computer architectures. *Computer Physics Communications*, 185(3):708–719, 2014.

[3] Y. Dong. A gpu parallel computing strategy for the material point method. *Computers and Geotechnics*, 66:31–38, 2015.

[4] F. Alauzet et al. On the use of space filling curves for parallel anisotropic mesh adaptation.

[5] Huy T. Vo et al. Simple and efficient mesh layout with space filling curves. *Journal of Graphics Tools*, 16(1):1–15, 2012.

[6] R. G. Joseph et al. Efficient gpu implementation for particle in cell algorithm. *IEEE Computer Society*, pages 1530–2075, 2011.

[7] Shankar Sastry et al. Mesh vertex and element reordering techniques for improved cache utilization in parallel mesh warping algorithms. *Engineers with Computers*, 30(4):535–547, 2014.

[8] X Kong et al. Particle in cell simulatiosn with charge conserving current deposition on graphic processing units. *Journal of Computational Physics*, 230(4):1676–1685, 2011.

[9] Mark Filipiak. Mesh reordering in fluidity using hilbert space-filling curves. *EPCC, University of Edinburgh*, 2013.

[10] Sagan H. Space filling curve. *Springer*, 1994.

[11] K Madduri. Memory efficient optimization of gyrokinetic particle to grid interpolation for multicore processors. *Association for Computing Machinery*, 2009.

[12] G. Stantchev. Fast parallel particle to grid interpolation for plasma pic simulations. *Journal of Parallel and Distributed Computing*, 68(10):1339–1349, 2008.

5

Benchmark Problem

In this chapter, we will be discussing about the two types of benchmark problems and observe effects of optimization strategies for ccNUMA system on simple kernel. The first benchmark problem will explain the successful working of Space Filling Curve algorithm for arbitrary unstructured finite element mesh both for regular and adaptive grid. Second benchmark problem will show successful working of space filling curve for one simple practical ***Oedometer*** *problem.*

5.1. Space Filling Curve for Arbitrary Mesh

In this benchmark, we will be simply generating space filling curve for arbitrary unstructured finite element mesh both for regular and adaptive grids. The Figure 5.1 shows the geometry under consideration. The table 5.1 shows normalized dimension of the problem.

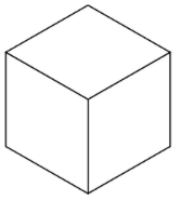


Figure 5.1: Problem Definition for Space Filling Curve

Table 5.1: Problem Dimension for SFC

Length	Breath	Height
1	1	1

5.2. Oedometer Problem

This section explains benchmark problem for simulation of one dimensional problem using two phase single point MPM formulation. The geometry of the problem is shown in Figure 5.2.

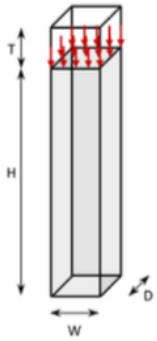


Figure 5.2: Problem Definition for Oedometer Problem

The dimension of the problem is shown in Table 5.2 and Material properties is shown in Table 5.3. This benchmark allows to verify the space filling curve implementation for practical problem.

Table 5.2: Dimension of the problem

Column Height (m)	Column Width (m)	Depth (m)	Top Height (m)
1.0	0.1	0.1	0.1

Table 5.3: Material Properties

Material Property	**Value**
Material type	2 Phase (Solid + liquid)
Initial Porosity [-]	0.4
Density Solid [Kg/m^3]	2650
Intrinsic Permeability [m^2]	$1.0214x10^{-9}$
Density liquid [Kg/m^3]	1000
Bulk Modulus Liquid [kPa]	$2.15x10^4$
Dynamic Viscosity Liquid [kPa/s]	$1.002x10^{-6}$
Material model solid	Linear Elasticity
Poisson ratio [-]	0.3

5.3. Effect of optimization strategies for ccNUMA architecture on simple kernel

This section will explain the simple computational kernel, which will be used to observe effect of thread / processor affinity and data placement. Since data placement and thread affinity are key to optimize particular kernel on *Cache Coherent Non-uniform Memory Architectures*, this example will illustrate these simple ideas and their influence on the performance. Below is the code snippet in C++ for the following computational kernel.

```
// Different variants of thread affinity policies
// will be used to show the effect

    // Memory Allocation
    double *a = new double[N];
    double *b = new double[N];
    double *c = new double[N];

    // Setting number of OpenMP threads
    omp_set_num_threads(16);

    long int Ntimes = 100;

    // First touch Principle
    #pragma omp parallel for schedule (static)
    for (size_t i = 0; i < N; i++)
    {
            a[i] = 1.0;
            b[i] = 2.0;
            c[i] = 3.0;
    }

    time_point t1 = high_resolution_clock::now();
    // Parallel Region
            #pragma omp parallel for schedule(static)
            for (size_t i = 0; i < N; ++i)
            {
                    a[i] = b[i] + 3.0*c[i];
            }
    time_point t2 = high_resolution_clock::now();
    // Time Measurement
    time_span = duration_cast<duration<double>>(t2 - t1);
```

6

Space Filling Curve BlackBox

*This chapter will give brief introduction to **Space Filling Curve BlackBox** developed during the period of literature study for generation of space filling curves for arbitrary unstructured 3D finite element mesh for both adaptive and non-adaptive grids.*

6.1. Space Filling Curve Blackbox

This section explains development of application ***SFCB*** which generates space filling curve for arbitrary three dimensional unstructured mesh. The development of application is done in C++. The basic working of application in Figure 6.1

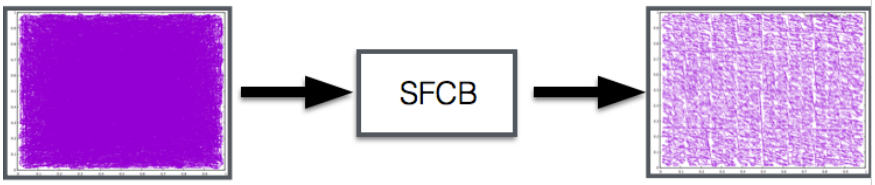


Figure 6.1: Basic Working of SFCB toolbox

Figure 6.1 pictorially shows simple working of stand alone application *SFCB* which takes arbitrary mesh file, processes it and convert it into new file mesh file with SFC ordering. This application is being used to convert **.GOM** files used as geometry file in **Anura3D** software to convert them with space filling curves. The next section explains little bit in depth working of *Space Filling Curve toolbox*.

6.2. SFCB Working

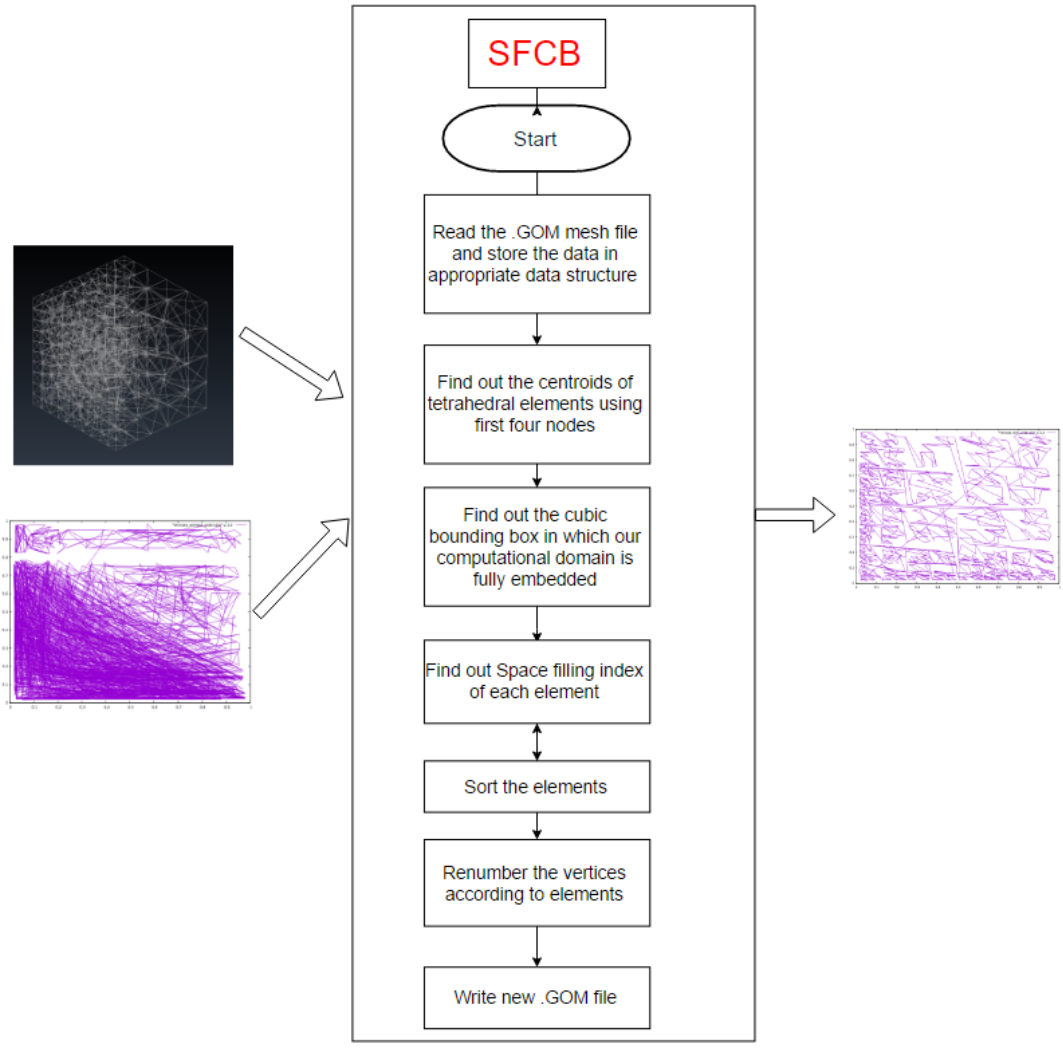


Figure 6.2: Detail Working of SFC toolbox

Figure 6.2 explains in little bit detail about how a arbitrary mesh file is converted into mesh file with space filling curve numbering.

7

Results and Discussion

In this chapter, we will be discussing about the preliminary results from the space filling curve generation and the effect of optimization strategies for the ccNUMA machine for a simple computational kernel.

7.1. Space Filling Curve Generation

This section will presents the result of the space filling curve algorithm with examples to illustrate the concept and will also discuss the various aspects such as, the time taken by regular and adaptive grid for different element sizes.

Figure 7.1, Figure 7.4, Figure 7.7 pictorially depicts the regular unstructured 3D finite element mesh with varying mesh density and Figure 7.10 shows the adaptive 3D unstructured finite element mesh. The subsequent figures shows the top view of the space filling curve. The Figure 7.2, Figure 7.5, Figure 7.8 and Figure 7.11 shows the arrangement of the finite element mesh in memory. The random access of memory can easily be justifiable from the above said figures.

The *Space Filling Curves* shown in Figure 7.3, Figure 7.6, Figure 7.9 and Figure 7.12 not only helps in maintaining data locality but also helps in achieving the efficient mesh partitioning for the shared memory multi-core architectures. The ccNUMA node can easily benefit from this kind of arrangement since parallel efficiency and scalability in the NUMA system is only attained by data locality and processor affinity. It is easily seen from the space filling curve that the spatial locality is maintained hierarchically.

The robustness of the SFC blackbox can be justified by observing its result on regular and adaptive 3D unstructured finite element mesh. The time taken by SFC blackbox to generate space filling curve for various sizes of 3D meshes is summarized in Table 7.1.

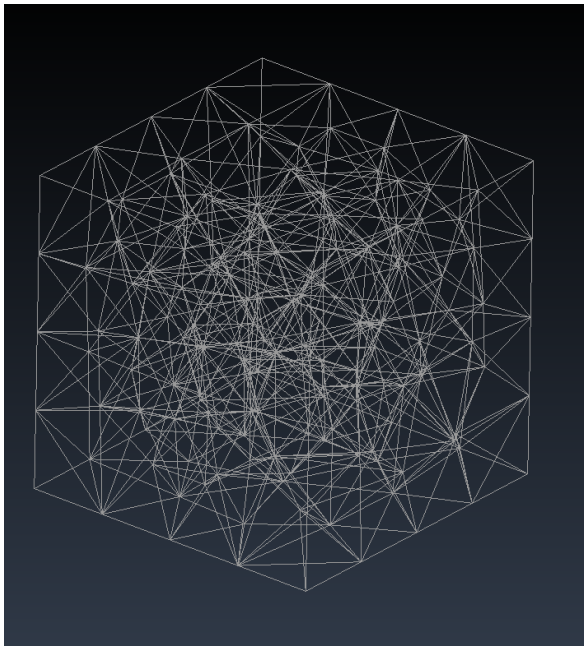


Figure 7.1: Regular Unstructured 3D FE mesh with element size = 0.25

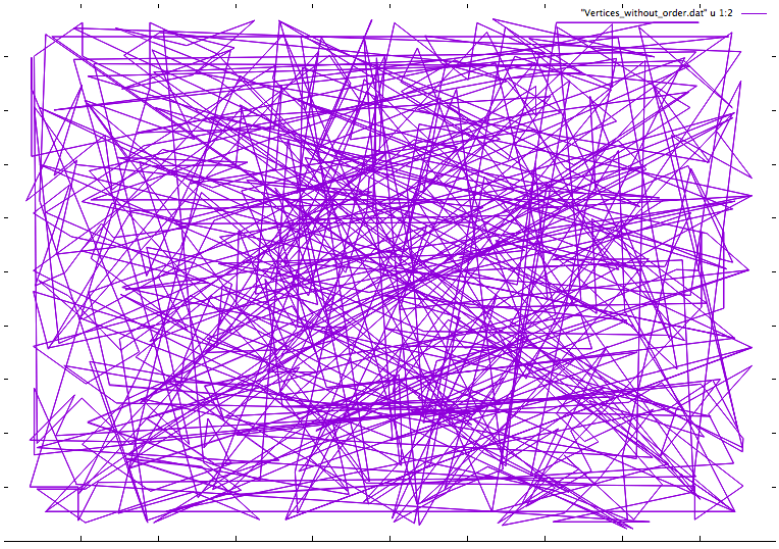


Figure 7.2: FE ordering of Figure 7.1 in memory

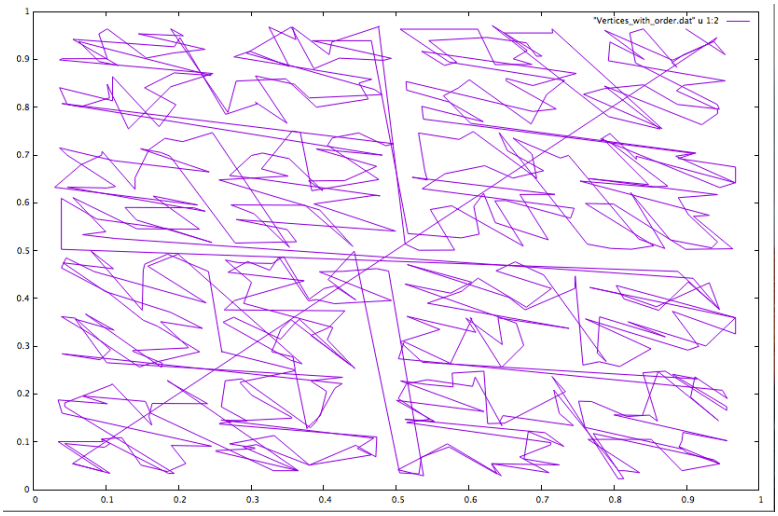


Figure 7.3: FE ordering in memory for Figure 7.1 with space filling curve

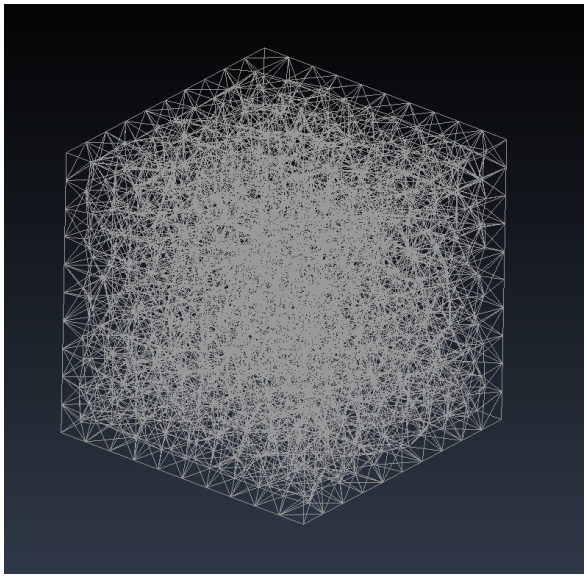


Figure 7.4: Regular Unstructured 3D FE mesh with element size = 0.1

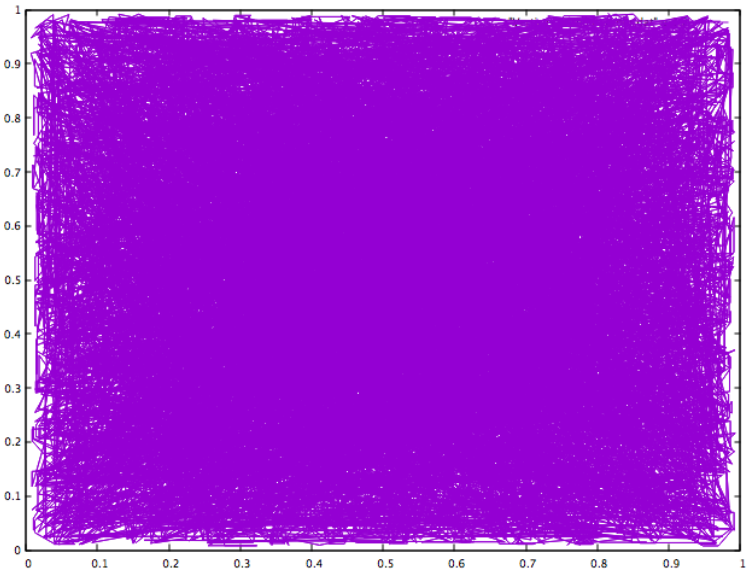


Figure 7.5: FE ordering of Figure 7.4 in memory

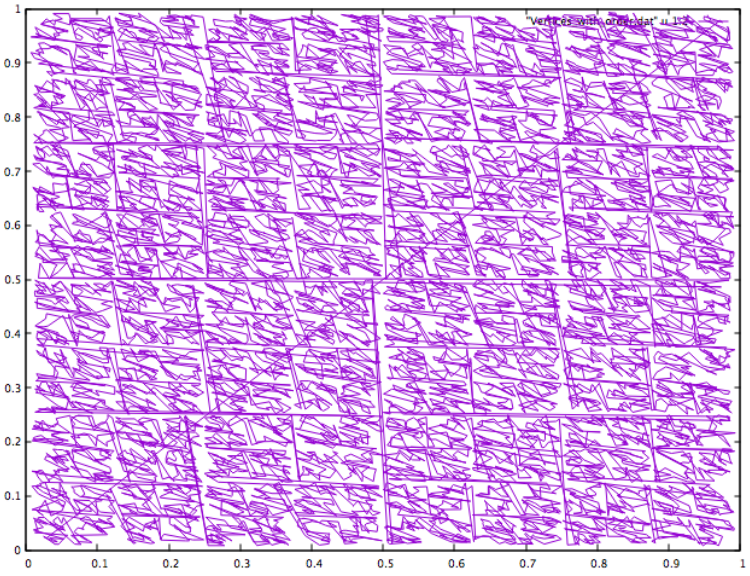


Figure 7.6: FE ordering in memory for Figure 7.4 with space filling curve

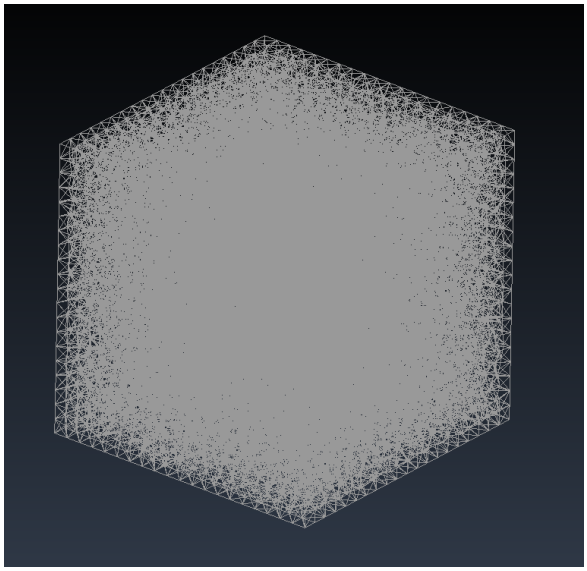


Figure 7.7: Regular Unstructured 3D FE mesh with element size = 0.05

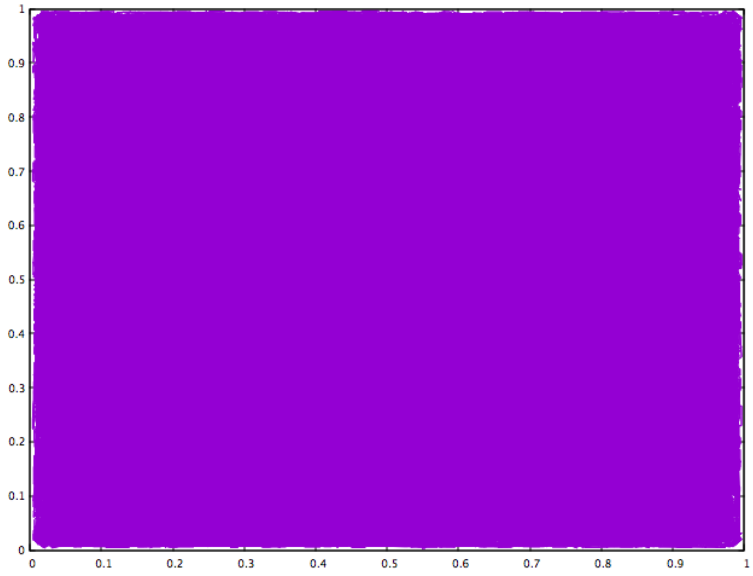


Figure 7.8: FE ordering for Figure 7.7 in memory

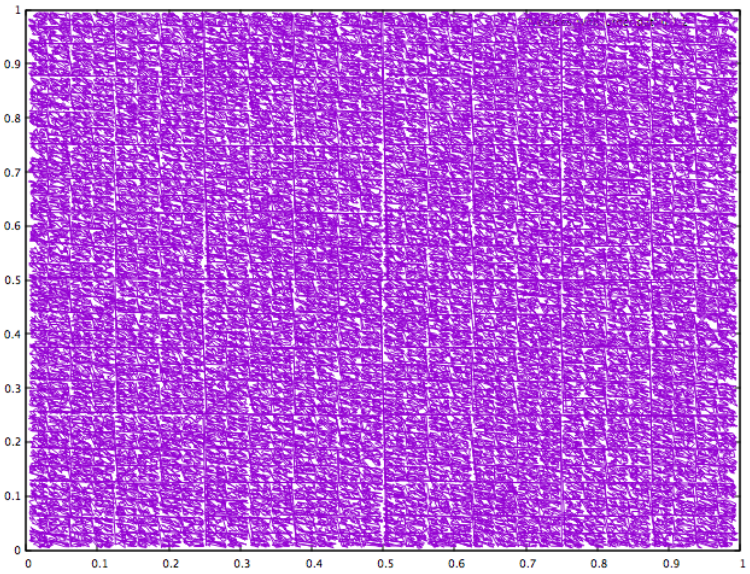


Figure 7.9: FE ordering for Figure 7.7 with space filling curve

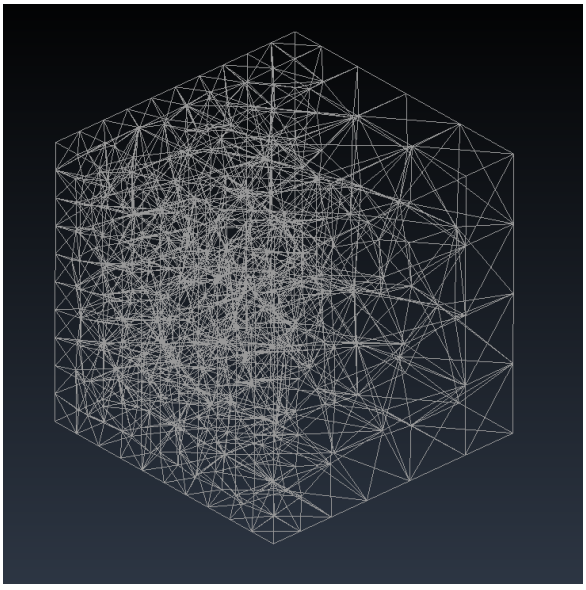


Figure 7.10: Adaptive Unstructured 3D FE mesh with element size = 0.25

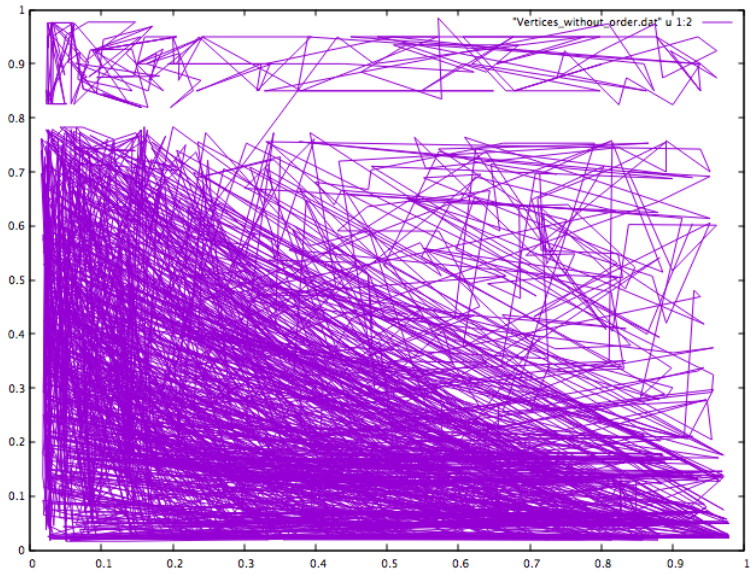


Figure 7.11: FE ordering for Figure 7.10 in memory

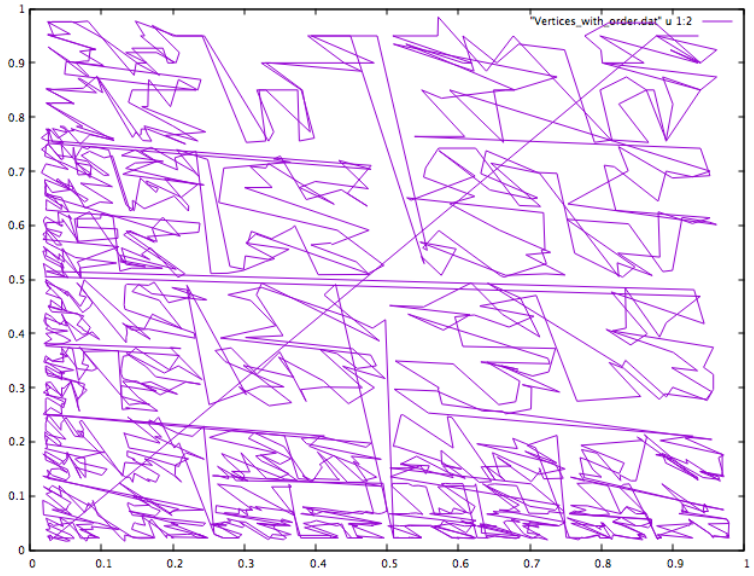


Figure 7.12: FE ordering for Figure 7.10 with space filling curve

Table 7.1: Variation in the time for generating the SFC curve for different mesh sizes

Element Size	Number of Elements	Number of Nodes	Time (seconds)	Number of Levels
0.25	235	1188	0.00038478	10
0.1	3400	19764	0.00599	12
0.05	30000	159603	0.05423	20

It can be clearly seen from the above table that the time taken to generate space filling curve even for large meshes is comparatively small and for those problems where the mesh has to be repeatedly used, it is good strategy to employ, since one time space filling curve generation will have many advantages for subsequent runs.

7.2. Oedometer Problem

This section will illustrate the results of the Oedometer problem and validate the working of space filling curve toolbox for problems of the practical interest.

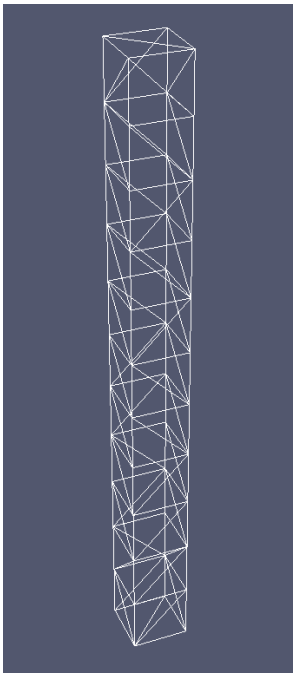
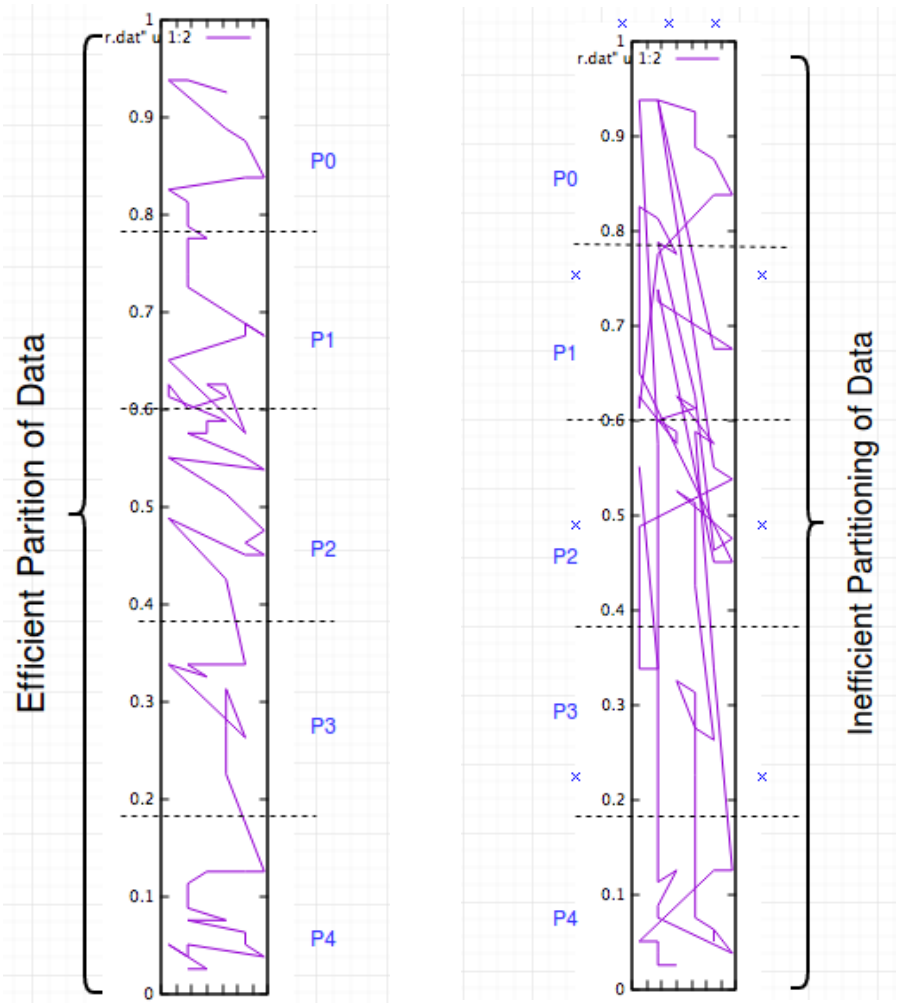


Figure 7.13: 3D Geometry of Oedometer Problem

The 3D plot in the above diagram are from the Oedometer problem described in chapter **Benchmark Problem**.



(a) SFC ordering of Oedometer problem in memory

(b) Original Ordering for Oedometer problem in memory

Figure 7.14: Comparison of Element data organized in memory with and without Space filling curve for Oedometer Problem if partitioned over multiple processor

The above figure depicts the element data placement of Figure 7.13. The Figure 7.14 (a) is Space filling curve ordering and 7.14 (b) is without space filling curve ordering. It is very clear from the above figure that, the unstructured finite element data organized by th space filling curve is clearly beneficial for the scalability and the efficiency on NUMA based machines as compared to one on the right where data is randomly stored and cannot be partitioned efficiently.

The main aim of plotting the above figure is to depict the importance of the space filling curve in rearranging the data since effective partitioning can be executed

without much overlap. The data can be easily be sliced into several parts such that, each part is executed on one or more processor without much interference or data sharing between other processor/cores, which makes this mesh renumbering strategy quite effective for achieving the scalability and efficiency for many problems in scientific computing.

7.3. Effects of Optimization Strategies for NUMA architectures

7.3.1. Effects of thread affinity and data locality

Let's first try to understand the architectural features of the machine, which is very important to mention before performing any test on it. Table 7.2 provides the specifications of the system on which testing will be performed.

Table 7.2: System Specifications

Architecture	x86_64
CPU(s)	16
On-line CPU(s)	0-15
Threads_per_core	1
Cores_per_socket	8
NUMA node(s)	2
Model Name	Intel(R) Xeon(R) CPU E5-2687W 0 @ 3.10GHz
CPU max MHz	3.8GHz
Hyper-threading	Possible
L1d Cache	31K
L1i Cache	31K
L2 Cache	256K
L3 Cache	20480K
Main Memory	32 Gigabytes
NUMA node0 CPU(s)	0-7
NUMA node1 CPU(s	8-15

Secondly, we also need to mention the compiler options used for compiling the code explained before in Chapter *Benchmark Problems*. Table 7.3 shows the compiler options used for testing.

Table 7.3: Compiler Options

icpc -std=c++11 -qopenmp -O3

Below, is the little explanation about each compiler option used.

1. **icpc** : Newest intel compiler for c++
2. **-std=c++11** : Enabling c++11 standard

3. **-qopenmp** : Generating parallel code
4. **-O3** : Highest level of optimization possible by compiler.

We will study the effects of thread affinity, first touch principle described in above chapters on a simple computational kernel.

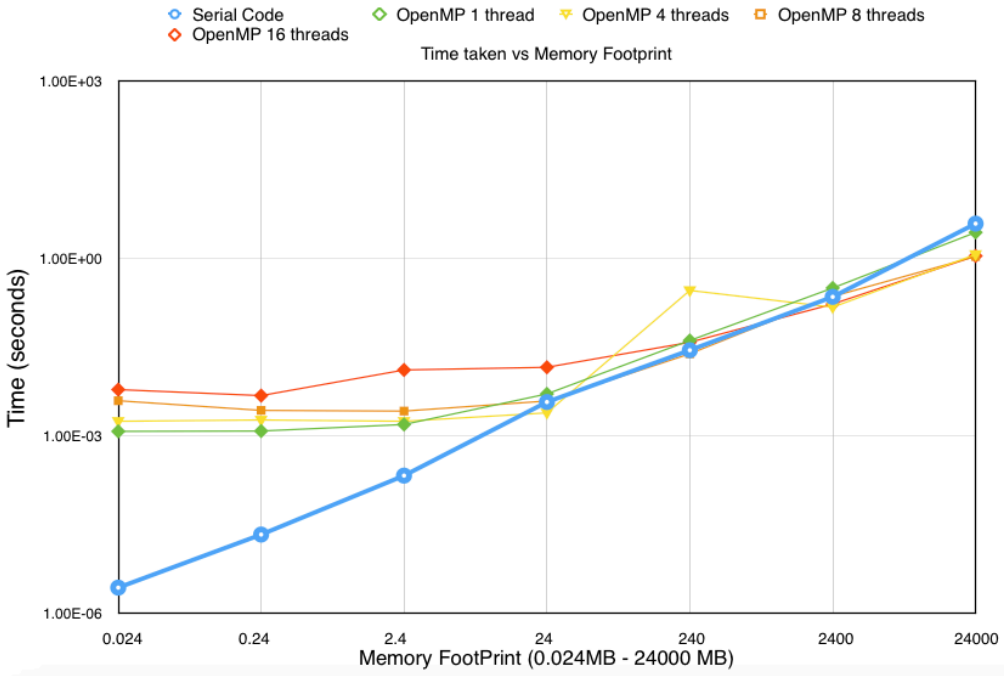


Figure 7.15: Time comparison between serial and OpenMP threads

It can be easily seen from Figure 7.15 that for the memory footprint of less than 24 MB which is the size of L3 cache, OpenMP threads create additional overhead and the serial code is way faster than the parallel one. The parallel code only shows speedup for problem size larger than 20MB. The memory footprint ranges from 24KB (size of L1 cache) to 24 GB (main memory) and to clearly see the difference between speedup, we zoom in the above figure for memory footprint of (240MB-24000MB).

It is very clear from Figure 7.16 that for larger problem sizes serial code is slower and OpenMP threads show some speedup, but we cannot see enough speedup between 4, 8 and 16 OpenMP threads.

We can explain this effect with the schematic diagram. Figure 7.17 conceptualizes the thread migration in the NUMA systems and its adverse effect on the performance if not taken care or given special attention. We have seen the performance penalty due to thread migration in Figure 7.15. Thread migration disassociates a thread from the processing core and its allocated local memory and schedules it on

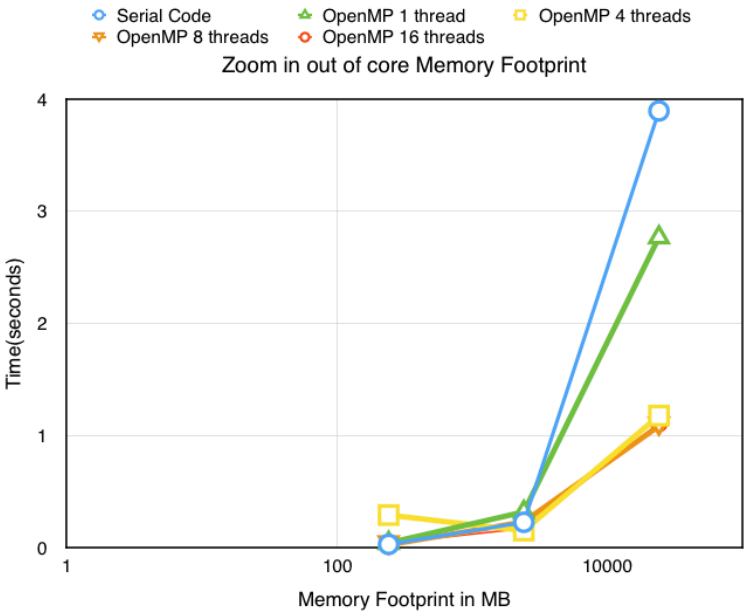


Figure 7.16: Zoom in for out of core **Memory Footprint**

another processing core. If thread migrates to the different **NUMA** node as shown in Figure 7.17, then huge penalty has to be faced due to the remote memory access which is clearly visible in Figure 7.15 and Figure 7.16.

Figure 7.18 explains the concept of thread affinity and first touch principle. When thread affinity control is used in the NUMA system, then the remote memory access is avoided to a full extent as thread cannot migrate from one processing core to another. Since the operating system scheduling policy is altered, it might also have adverse effects.

Figure 7.19 shows performance improvement when thread affinity and first touch principle is used. The environment variable **KMP_AFFINITY** binds OpenMP thread to a particular physical core or the set of physical cores. It is very clear from the Figure 7.19 that the thread affinity and data placement if ignored can lead to serious penalties and performance degradation in the computational loops.

Figure 7.20 shows comparison of performance of optimized and non-optimized version of the code for 16 OpenMP threads. It is clearly evident from the Figure 7.20 that optimized version is 2x times faster, scalable and parallely efficient than it's counterpart where thread affinity and first touch principle are not used.

7.3.2. Impact of Intel's Hyper-threading Technology

It is clear from Figure 7.21 that a non-hyper-threaded core performs better as compared to a hyper-threaded core. It is very interesting to know that, the 32 OpenMP threads in a hyper-threading enabled system behaves similar to the 16 OpenMP

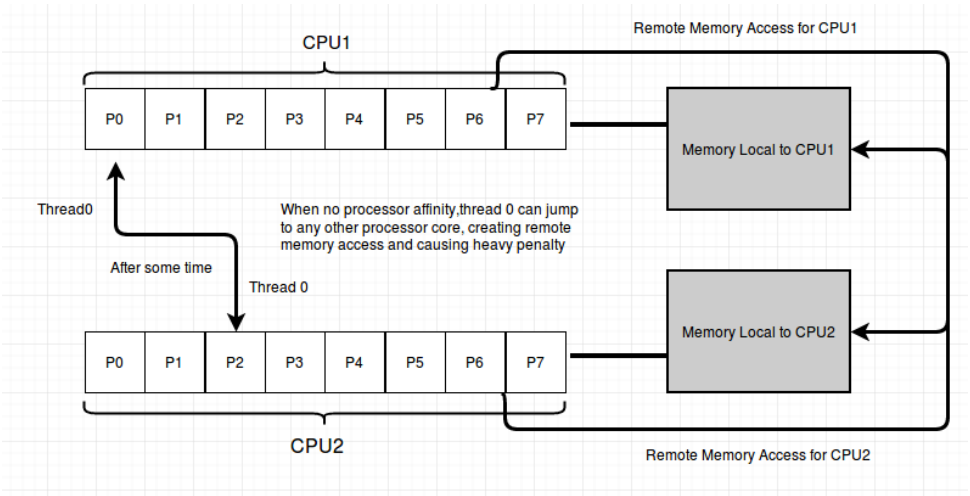


Figure 7.17: Thread Migration in NUMA system explained

threads in a hyper-threaded disabled system. Even though the hyper-threading is expected to give better performance by duplication of the floating point units, it is observed in this case that it is not contributing significantly to the performance improvement of the kernel.

It depends on the application , whether the hyper-threading should be enabled or disabled because it does not increase the physical core counts, but only gives a false impression to the operating system about double the number of the processing cores. It is also noted in the literature that the use of a hyper-threaded core increases the energy consumption as compared to a non-hyper-threaded one.

7.3.3. Single and Double Precision computations

Some of the widely used processors for the high performance computing today demonstrate much higher performance for 32 bit floating point arithmetic than for 64 bit floating point arithmetic. These modern architectures demonstrate approximately twice the performance for single precision execution when compared to the double precision.

The overall approach should be to use single precision whenever possible, especially for the most compute intensive parts of the code and then fall back to double precision whenever required. The Figure 7.22 shows the single and double precision computing. It is clearly visible from the Figure 7.22 that single precision computing takes half as time as double precision computing. It will be interesting to note the performance gain in the current MPM code by introduction of single precision arithmetic.

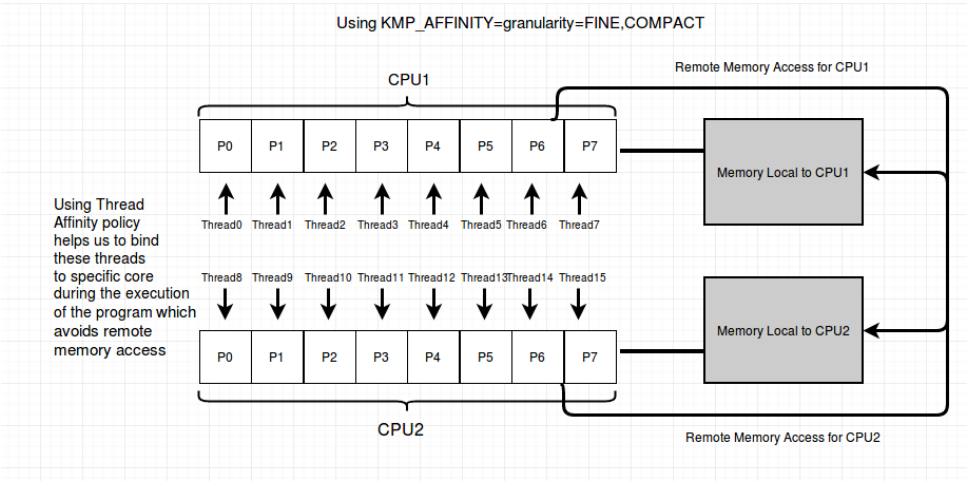


Figure 7.18: Thread affinity control in NUMA system explained

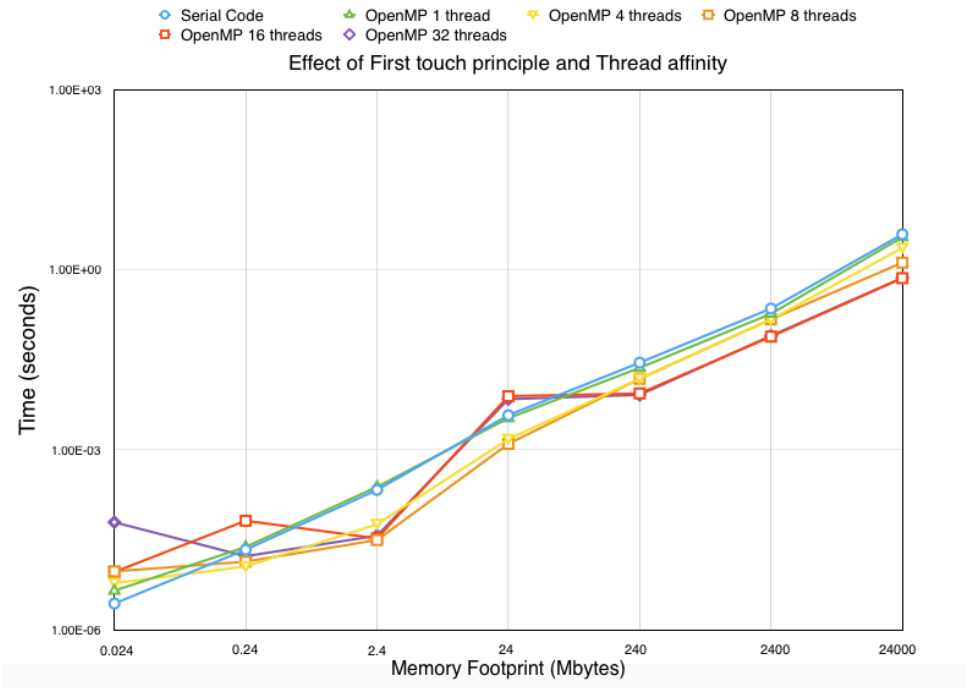


Figure 7.19: Performance Improvement with **First touch principle and thread affinity**

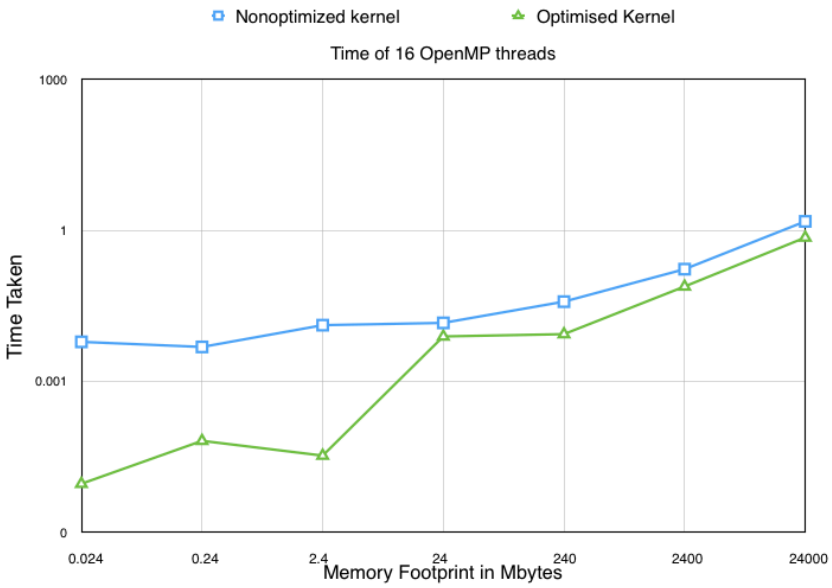


Figure 7.20: Comparison of Optimized and Non-optimized code for 16 OpenMP threads

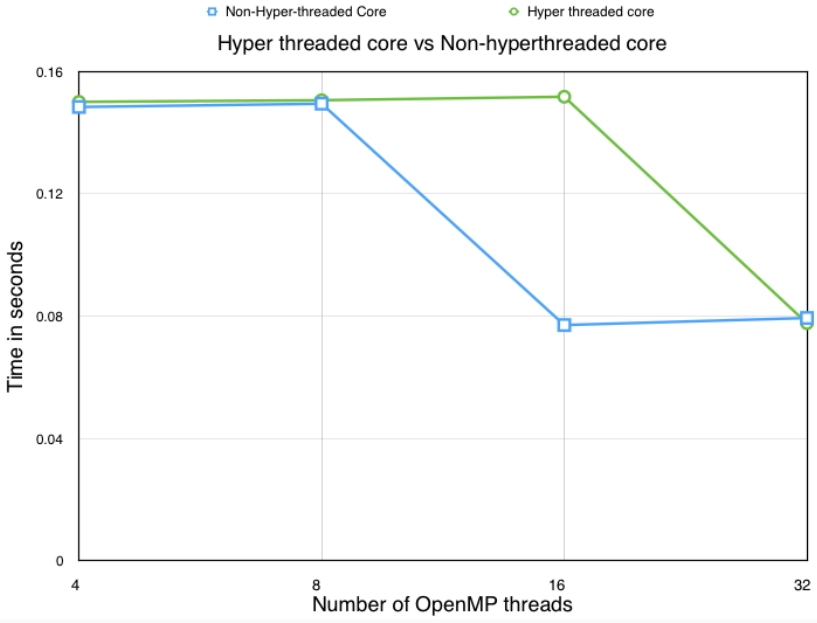


Figure 7.21: Comparison of Optimized and Non-optimized code for 16 OpenMP threads

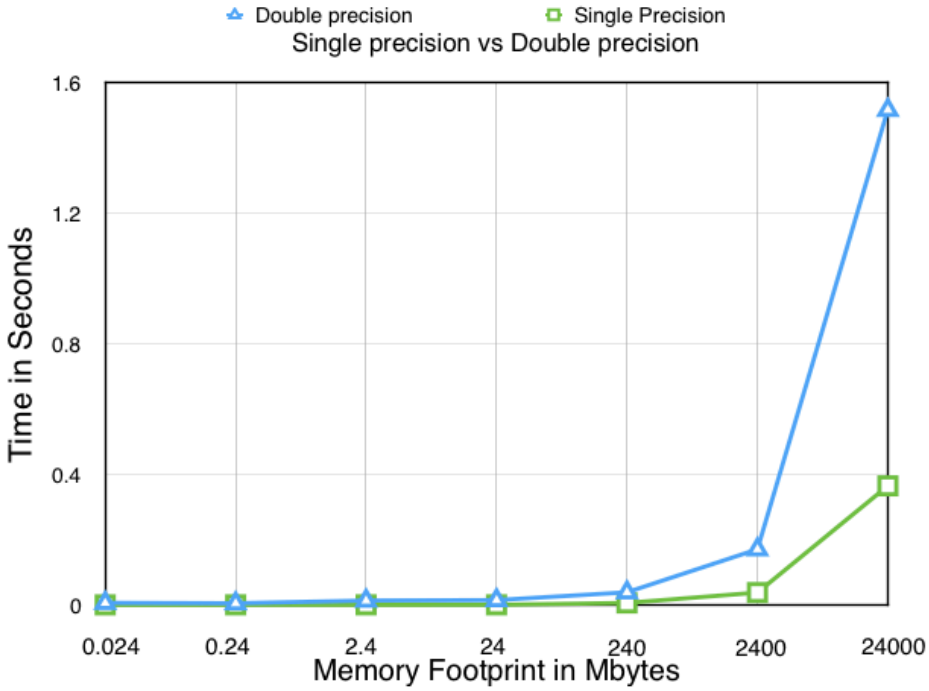


Figure 7.22: Comparison of Single and Double Precision Data

8

Conclusion and Future Work

In this chapter, we will be concluding about the research work carried out in the period of literature survey and make inference for the discussion about the future work to be done in context of improving parallel scalability and efficiency of the Material Point Method on ***Cache Coherent Non-uniform Memory Access architectures****, which will be a milestone for the current MPM 3D code to be efficient on large clusters and eventually supercomputers.*

8.1. Motivation

Deltares is currently involved in development of a 3D dynamic meshfree numerical method, the Material Point Method for advanced simulation of large deformation problems in the Geotechnical Engineering. Such simulations involve interaction between the structure and soil. A challenging research project is being executed at Deltares together with the industry partners to investigate installation of the offshore large diameter monopiles through hammering and vibration for the offshore wind farm construction. Figure 8.2 explains pictorially the variation of soil density upon driving monopile into the ground solved by **The Material Point Method**.

8.2. The Larger Perspective

Computational geo-mechanics in the near future is aiming at solving larger problems of industrial and academic interests. In order to solve problems which are outside the academic shell and typically have larger human interest need significant amount of computing resources. The Large amount of computing resources does not guarantee acceleration for solving bigger and complex problems, but efficient and judicious use of available computing resources is a key to reduce computational time, achieve high floating point performance and decrease amount of energy consumption.

In short we need to have efficient numerical algorithms and clever programming strategies for achieving above said goals. This master's project focuses on

the implementation of such methods and applications of the various techniques to accelerate the *Material Point Method* on the **emerging computing architectures**.

8.3. Work Completed

In order to realize the parallel scalability and efficiency for the **Material Point Method** on large clusters and eventually supercomputers, it is essential for it to become scalable and efficient on the ccNUMA node. Effective data placement and optimized use of computational resources is the key to achieve the high performance on these kind of machines.

Implementation of the *Space Filling Curves* for reordering finite element data is the first step to achieve that milestone in near future. We also have demonstrated that use of thread / processor affinity , Intel's hyper-threading technology, use of memory page placement policy and floating point balance are the key hardware issues which are to be given special attention for optimization of particular computational kernel.

8.4. Strategic Diagram

The Figure 8.1 explains in detail about the approach for the future work. In short, we have implemented the *space filling renumbering for arbitrary mesh* and have also tested optimization concept on *NUMA* machines for simple computational kernel. We plan to merge these two ideas for the Material Point Method in this master's project and parallelize the current 3D MPM code with **OpenMP 4.0** specifications.

8.5. Future Work

This master thesis will focus on bridging the two main pillars of the high performance computing i.e data locality and judicious use of computational resources for ***The Material Point Method*** . We want to observe the full scalability for the current MPM 3D code on the NUMA machine which is currently deployed at Deltares with capabilities and special features of **OpenMP 4.0** such as vectorization, processor affinity control. We want the current MPM 3D code to become efficient on parallel machines, so that eventually bigger and more realistic problems can be solved with reduced computational time.

The ***Space Filling Curve*** implementation will be carried forward for the pure serial MPM 3D code provided by **Deltares**. OpenMP 4.0 will be used to parallelize all computational kernels in the code and towards the end, scalability and parallel efficiency will be observed for a large scale simulation.

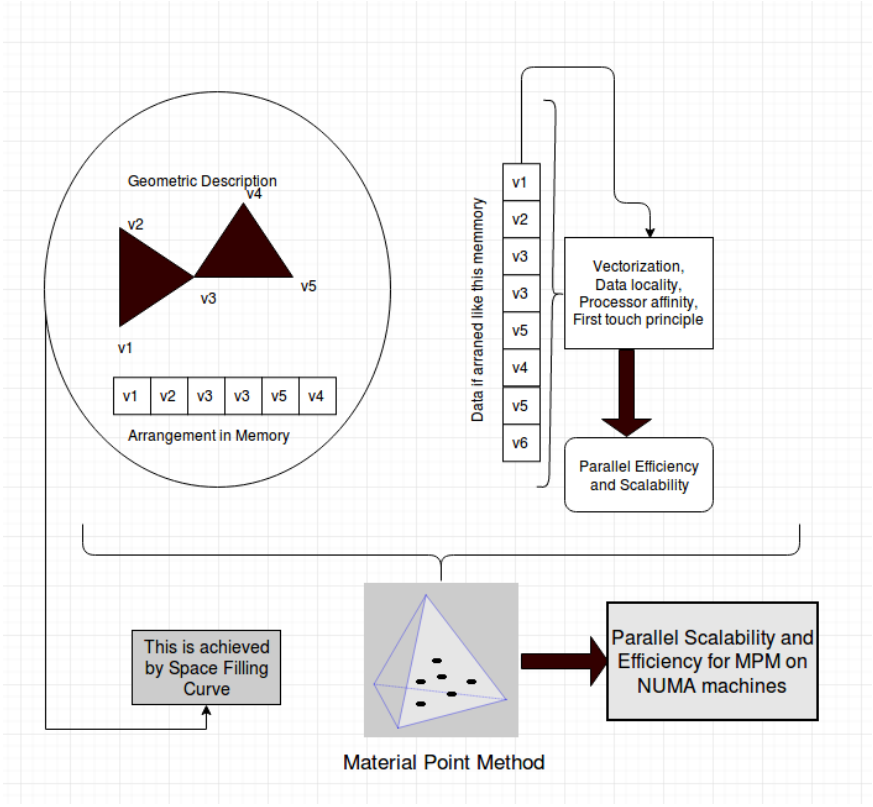


Figure 8.1: Strategic Diagram for Future Work

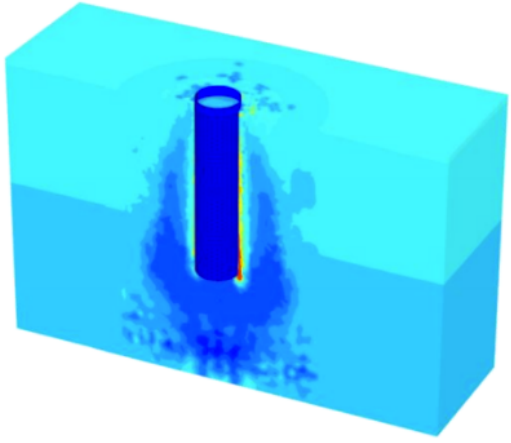


Figure 8.2: Variation of Soil Density upon driving monopile into the ground