Implementation and Evaluation of Scientific Simulations on High Performance Computing Architectures

By

Bibrak Qamar (2007-NUST-BIT-106)

Jahanzeb Maqbool (2007-NUST-BIT-118)

A project report submitted in partial fulfilment of

the requirement for the degree of

Bachelors of Information Technology

School of Electrical Engineering and Computer Science (SEECS)

National University of Sciences and Technology (NUST)

Islamabad, Pakistan

(2011)

Approval Page

It is certified that the contents and form of project report entitled "Implementation and Evaluation of Scientific Simulations on HPC Architectures" submitted by **Bibrak Qamar** and **Jahanzeb Maqbool** has been found satisfactory for the requirement of the degree.

Advisors:

Mr. Mohsan Jameel ____________________

Dr. Leila Faiz Islamil ____________________

Mr. Akber Mehdi ____________________

Dr. Aamir Shafi ____________________

Dedication

Bibrak’s dedication:

Dedicated to ***Salman Taseer*** who showed me dream of equal Pakistan.

Jahanzeb’s dedication:

Dedicated to my beloved father ***Mr. Syed Maqbool Hussain Hashmi***. Perhaps I could not become the one you want, but I would try to be as perfect as one can be in what I've chosen.

With Love,

Syed Jahanzeb Maqbool Hashmi

Acknowledgements

We thank Mr. Mohsan Jameel, Mr. Akbar Mehdi and Dr. Aamir Shafi from NUST and Dr. Leila Islmail Faez from UAE University, for their kind support from idea to design and development - for their precious time spent in the completion of the project.

Moreover, we also thank Mr. Hammad Siddique and Mr. Umar Butt, system administrators at High Performance Computing Lab, SEECS and UAE University for providing access to Clusters and resolving technical issues.

Table of Contents

CHAPTER 1: INTRODUCTION 11

CHAPTER 2: HIGH PERFORMANCE COMPUTING ARCHITECTURES.... 13

HARDWARE........ 13

Shared Memory Architectures........ 13

Distributed Memory Architectures........ 15

Hybrid Distributed-Shared Memory Architectures........ 16

General Purpose Graphics Processing Units (GPGPUs)........ 17

SOFTWARE........ 18

Message Passing Interface........ 18

OpenMP........ 20

CUDA........ 22

CHAPTER 3: BACKGROUND AND LITERATURE REVIEW........ 24

FLUIDANIMATE........ 24

Fluid Particle Simulation Methods........ 24

Fluidanimate Phases........ 25

Force Computation Methodologies........ 26

OIL RESERVOIR SIMULATION........ 27

Reservoir Simulation Process........ 27

SLE Solvers........ 28

Why Conjugate Gradient Method........ 31

The Conjugate Gradient Algorithm........ 32

BLACKSCHOLES........ 32

Blackscholes Algorithm........ 32

CHAPTER 4: CASE STUDY I: FLUIDANIMATE........ 33

FLUIDANIMATE 33
Fluidanimate On Distribute Memory Cluster 33
Fluidanimate On Shared Memory Architectures 43
Fluidanimate On Graphics Processing Units (GPUs) 51
CHAPTER 5: CASE STUDY II: OIL RESERVOIR SIMULATION 57
OIL RESERVOIR SIMULATION 57
Oil Reservoir Simulation on Distributed Memory Clusters 59
Oil Reservoir Simulation on Shared Memory Processors 80
Oil Reservoir Simulation on GPUs 90
CHAPTER 6: CASE STUDY III: BLACKSCHOLES 97
BLACKSCHOLES 97
Blackscholes On Distribute Memory Clusters 97
Blackscholes On GPUs 103
CHAPTER 7: VISUALIZATION 110
FLUIDANIMATE 110
OIL RESERVOIR SIMULATION 113
CHAPTER 8: CONCLUSION & FUTURE WORK 116
References 117
Appendix 121
A1 121
B1 121
B2 121
B3 121
B4 122

List of Tables

Table 1 56

Table 2: Total Time, Computation Time, Memory Copy Time in sec 98

LIST OF FIGURES

Figure 1: Shared Memory (UMA) 14

Figure 2: Shared Memory (NUMA) 15

Figure 3: Distributed Memory Architecture 16

Figure 4: Hybrid Distributed Shared Memory Architecture 17

Figure 5: CPU and nVidia GPU basic architecture 18

Figure 6: MPI Inter Process Communication 19

Figure 7: MPI program flow 20

Figure 8: Fork and Join model 21

Figure 9: Scalar vs. SIMD Operations 22

Figure 10: CUDA process flow 23

Figure 11: Reservoir Simulation Process 27

Figure 12: 32x32 Matrix representing a 2D reservoir in 1 Phase 29

Figure 13: Conjugate Gradient Method Algorithm 31

Figure 14: Fluidanimate Parallel Flow 34

Figure 15: Particles World Grid 35

Figure 16: Particles World Grid – 3D 35

Figure 17: FluidAnimate MPI work Division among processors 36

Figure 18: FluidAnimate MPI Communication of Ghost Cells 37

Figure 19: Fluidanimate OpenMP Optimized implementation Design 48

Figure 20: Fluidanimate CUDA Program Flow 52

Figure 21: Fluidanimate CUDA Design 52

Figure 22: Architecture Diagram of Oil Reservoir Simulator 58

Figure 23: Sample matrix A (sparsity view) 60

Figure 24: Matrix A decomposed into horizontal blocks among 4 processes 60

Figure 25: Test for Load Balance 61
Figure 26: Matrix Vector Multiplication – Naïve 62
Figure 27: Matrix A further divided into vertical blocks 68
Figure 28: Communication in a ring 69
Figure 29: Step 1 MVM 70
Figure 30: Step 2 MVM 71
Figure 31: Step 3 MVM 71
Figure 32: Step 4 MVM 72
Figure 33: Row Partition [left] and Block Partition [right] 80
Figure 34: Representation of matrix A in CSR format 81
Figure 35: How the different blocking storage formats split the input matrix into blocks 81
Figure 36: Simple Sparse MVM in CSR format 82
Figure 37: MVM with Block Partitioning of matrix A 86
Figure 38: 32x32 Matrix representing a 2D reservoir in 1 Phase. Here offset from central diagonal is nx = 4 91
Figure 39: CUDA MVM kernel 92
Figure 40: VVM and Intra Block reduction 93
Figure 41: Graphical view of intra block reduction. 93
Figure 42: Global reduction, among 4 thread blocks 94
Figure 43: VV Addition 95
Figure 44: Percentage of time spent on different operations 95
Figure 45: Basic Flow of Blackscholes in distribute memory 98
Figure 47: MPI based parallelization approach to Blackscholes 100
Figure 47: GPU based parallelization approach to Blackscholes 103

Figure 48: Options in the form of Option Data Structure (Naïve approach) assuming x, y option variables 104

Figure 50: Memory Accesses in non-coalesced fashion 106

Figure 50: Memory Accesses in Perfectly Coalesced fashion 106

Figure 51: Blackscholes GPU naïve implementation non-colaescing Problem ... 107

Figure 52: Blackscholes GPU Optimized implementation strategy 107

Figure 53: Blackscholes GPU Optimized Coalesced Memory Access 108

Figure 54: Initial stage, Fluidanimate demo 110

Figure 55: After 53 frames, Fluidanimate demo 111

Figure 56: After 90 frames, Fluidanimate demo 112

Figure 57: Initial stage, Oil Reservoir Simulation demo 113

Figure 58: After 350 days, Oil Reservoir Simulation demo 114

Figure 59: After 1125 days, Oil Reservoir Simulation demo 115

Figure 60: After 1975 days, Oil Reservoir Simulation demo 115

Chapter 1

INTRODUCTION

Computational Science is field of study in which computers are used to solve challenging scientific problems. Real or imaginary world scientific problems are converted into mathematical models and solved using numerical analysis techniques with the help of high performance computing famously called scientific computing.

As computer technology is advancing rapidly, computers are becoming increasingly powerful and increasingly available, and with the advancement of mathematics and other basic sciences, the use of robust computer simulation and modelling techniques are being recognized as a key to the economic growth and scientific advancement.

Computational science now constitutes what is famously called the third pillar of science together with theory and physical experimentation. The 2005 Report to the President of US, Computational Science: Ensuring America's Competitiveness, states that "the most scientifically important and economically promising research frontiers in the 21st century will be conquered by those most skilled with advanced computing technologies and computational science applications." (1)

Scientific simulations are typically compute intensive in nature. It takes week or days to obtain result if ordinary single processor system is used. For example, in predicting weather the amount of computation is so large that it could take ordinary computer weeks if not months. To make a simulation more feasible the use of High Performance Computing (HPC) is essential.

HPC is the use of supercomputers and complex algorithms to do parallel computing i.e. to divide large problems into smaller ones, distribute them among computers so as to solve them simultaneously. In this project we have

implemented some widely used scientific simulations namely fluid dynamics (fluid particles simulation), oil reservoir simulation and Black-Scoles (predicting price of option – finance). The aim of the project is to analyze the performance characteristics of compute intensive scientific applications on leading HPC architectures, namely distributed memory (MPI), shared memory (threads or cores) and GPUs. We have examined performance bottleneck on these architectures, how to overcome these bottlenecks and what are the optimized ways of programming these applications on HPC architectures.

In this document, after introduction we will be discussing High Performance Computing Architectures in both hardware and software perspective. Then we will describe the literature reviews of our proposed case studies. After that we will propose the design and implementation of these applications on different HPC architectures along with discussion on results. Finally we will show some visualization of the simulation and conclude our work.

Chapter 2

HIGH PERFORMANCE COMPUTING ARCHITECTURES

This chapter begins with an introduction to High Performance Computing (HPC) architectures. We will also discuss how the emergence of these architectures is affecting the mainstream hardware and software industry. Later, we will discuss two major types of HPC architectures: Shared Memory Architectures (SMA) and Distributed Memory Architectures (DMA). The emergence of multicore technology has also become the root cause for emergence of General Purpose Graphics Processing Units (GPGPUs) architectures. Moreover, recently the focus is on hybrid programming models like combining SMAs with DMAs along with GPUs to achieve the performance at its peak. Discussion has been made, about our test application of variant domains for this project, in the context of SMAs, DMAs and GPU accelerators.

HARDWARE

In this chapter our focus will be on High Performance Computing Architectures. The widely used HPC architectures are:

I. Shared Memory
II. Distributed Memory : clusters
III. Hybrid
IV. GPUs

Shared Memory Architectures

Shared memory architectures (SMA) vary widely, but generally have in common the ability for all processors to access all memory as global address space. Multiple processors can operate independently but share the same memory

resource. In SMAs the changes in a memory location made by one processor are visible to all other processors. SMAs can be further divided into two major classes based upon memory access times

1. Uniform Memory Access (UMA)
2. Non Uniform Memory Access (NUMA)

In UMA, the main physical memory is accessed by all the processors. These processors exhibit cache coherency which means if one of the processors updates a location which is in shared memory, then rest of the processors know about the update.

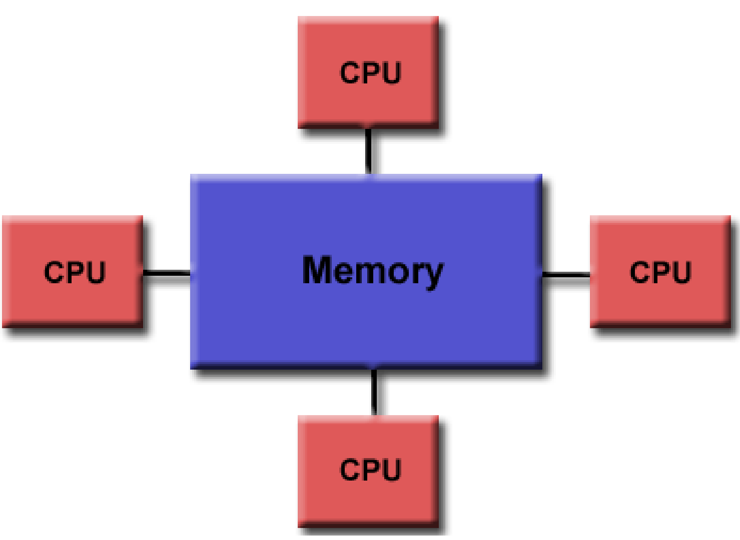


Figure 1: Shared Memory (UMA)

Source [https://computing.llnl.gov/tutorials]

The NUMA is often made by physically linking two or more symmetric multiprocessors. In this case, all the processors do not necessarily have equal access time to all the memories.

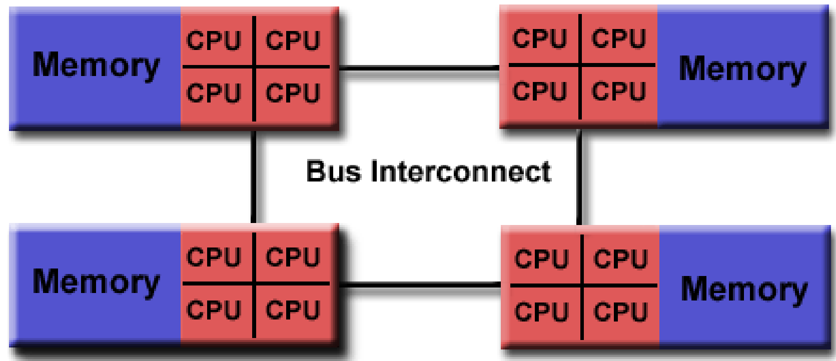


Figure 2: Shared Memory (NUMA)

Source [https://computing.llnl.gov/tutorials]

One of the major advantages of shared memory architectures is that global address space provided by it gives a user friendly programming approach to memory and the sharing of data between tasks is both fast and uniform.

On the other hand, the main disadvantage of SMAs is scalability between memory and CPUs. If we add more CPUs then it can geometrically increase the traffic on the shared memory CPU path. The programmer's job becomes tough in shared memory environment because of synchronization constructs to ensure the consistent global memory access.

Distributed Memory Architectures

In Distributed Memory Architectures (DMAs), each processor has its own local memory. The memory addresses in one processor do not map to another processor, so there is no concept of global address space shared by all the processors. Distributed Memory systems require a communication network to connect inter processor memory. Each processor operated independently on its own local memory. The changes made by a processor to its local memory do not apply to the memory of other processors. In DMAs whenever a processor need to access data, which is present in the memory space of another processor's memory, a communication across the network is needed. It becomes the task of programmer to explicitly define how and when data is communicated. The synchronization of

data among processors is also the programmer's responsibility. The network fabric which is used to transfer data between different processors in DMA can vary in nature, but it can be as simple as Ethernet.

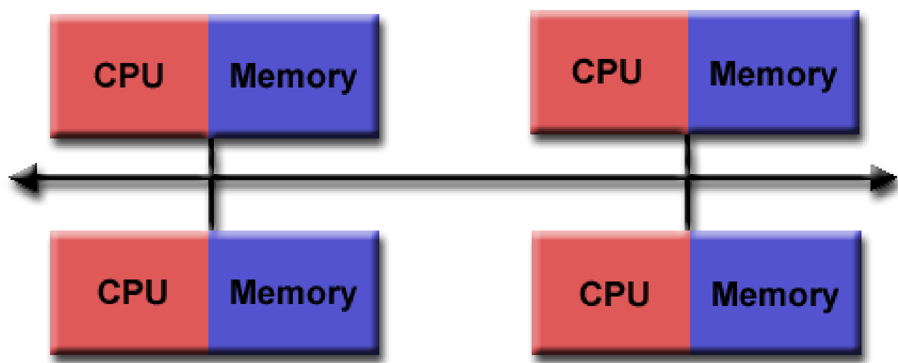


Figure 3: Distributed Memory Architecture

Source [https://computing.llnl.gov/tutorials]

The major advantage of using DMAs is memory scalability with number of processors. If we go on increasing number of processors then the size of memory increases. The cache coherency overhead is removed as each processor can rapidly access its own memory without any interference.

On the other hand, the main problem with DMA is that programmer is responsible of many of the details associated with the data communication between processors. Existing data structures based on the global address space need to be mapped to this memory organization.

Hybrid Distributed-Shared Memory Architectures

A better approach in Parallel Computer Memory Architectures is to employ both shared and distributed memory architecture. This is called Hybrid Distributed-Shared Memory Architectures (DSMA).

In HDSMA, the shared memory component is usually a cache coherent symmetric multiprocessor (SMP) machine which means the processors on that

machine can address that machine's memory as global. The distributed memory component is the network of multiple SMPs. In this case, SMPs know only about their own memory not the memory of other SMPs. Therefore, a communication through network is required to transfer data from one SMP to another SMP machine on the network. From last few years, the data from *top500.org* showed that HDSMAs have been prevailing.

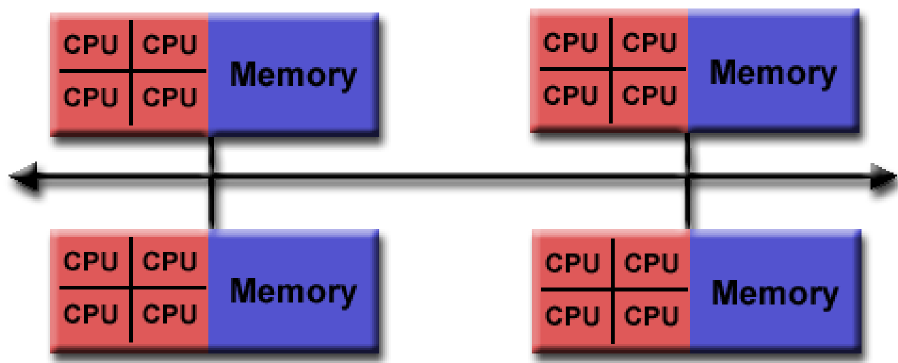


Figure 4: Hybrid Distributed Shared Memory Architecture

Source [https://computing.llnl.gov/tutorials]

General Purpose Graphics Processing Units (GPGPUs)

Few years back, GPUs were considered as specialized piece of hardware that is designed for maximum performance in graphics applications. Today, GPUs are considered as massively parallel many core processors easily available and fully programmable. In GPU market nVidia is the leading manufacturer that manufactures General Purpose GPU (GPGPU) i.e. Tesla, GeForce and Quadro series. The nVidia's proprietary programming model for GPU programming is Compute Unified Device Architecture (CUDA). We will be discussing CUDA more briefly in our programming model section. The basis for using GPUs in parallel computing is (2):

- High throughput computation
- High bandwidth memory
- High availability to all

More specifically, GPUs are well suited to address problems that can be expressed as data parallel computations. In GPUs, same program is executed on many data elements in parallel with high arithmetic intensity (3).

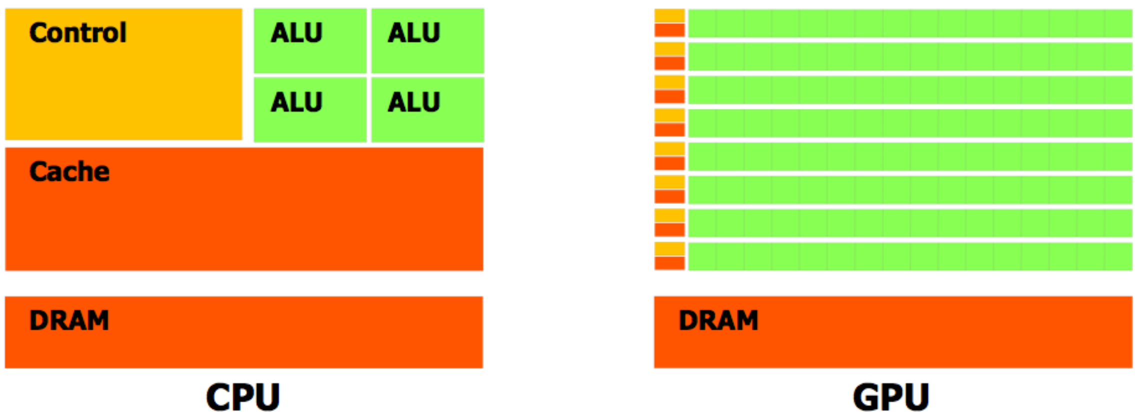


Figure 5: CPU and nVidia GPU basic architecture

SOFTWARE

Message Passing Interface

Message Passing Interface (MPI) is API specification used to program compute Clusters by doing message passing between processors. MPI is a de-facto standard of the industry and HPC community. The idea is to exchange data which is stored in the address space of another process by means of simple routines, like send and receive, see figure 6.

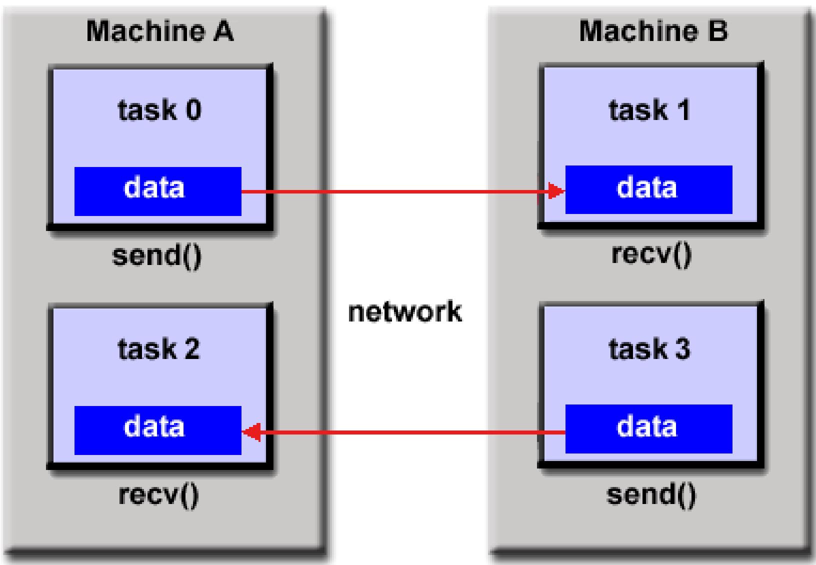


Figure 6: MPI Inter Process Communication

Source [https://computing.llnl.gov/tutorials/mpi/]

The goal of the Message Passing Interface is to provide portability, efficiency and flexibility (4). There is no need to modify the source code when shifting to different platforms which support MPI and communication logic is decoupled from the program.

Parallelism is explicit, requiring the programmer to indentify and exploit parallelism in the algorithm by using MPI routines.

The flow is a MPI program is shown in figure 7. The program need the mpi header file (mpi.h or mpif.h), to initialize the environment MPI_Initialize() is called and to terminate MPI_Finalize() is called, in between is the parallel region where the programmer can use MPI routines.

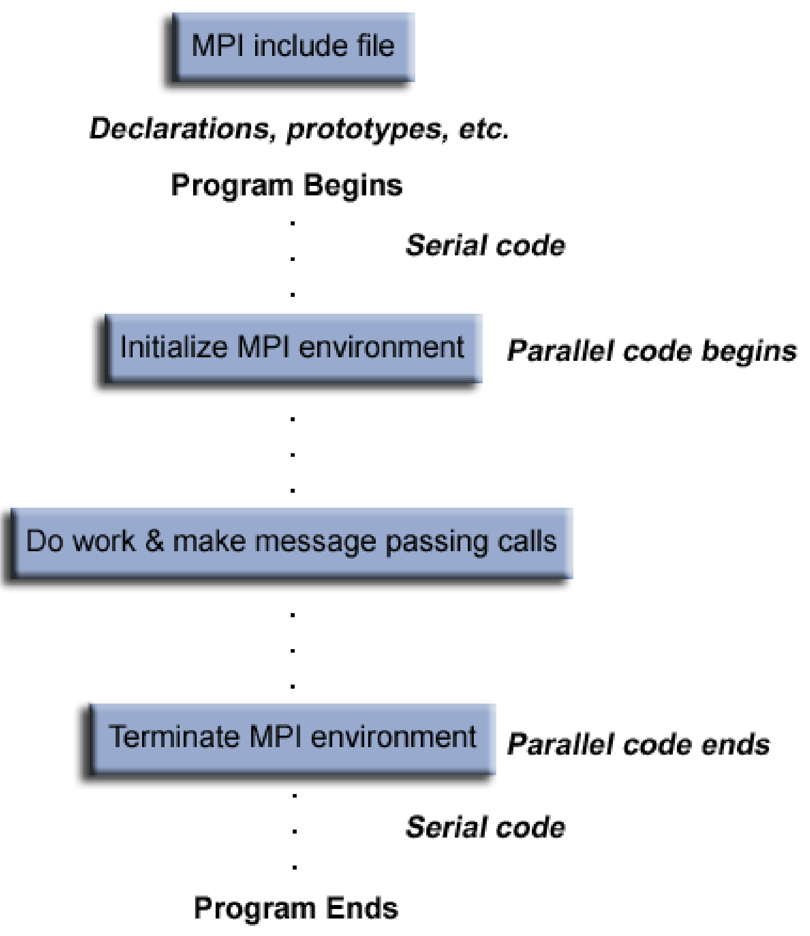


Figure 7: MPI program flow

Source [https://computing.llnl.gov/tutorials/mpi/]

OpenMP

OpenMP (5) (Open Multi-Processing) is an API for programming shared memory machines. OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

OpenMP provides set of compiler pragmas, directives, function calls and environment variables that explicitly instruct the compiler where to use parallelism.

OpenMP is based on ***fork and join*** model, the program begins as single main thread called the ***master thread***. The *master thread* runs sequentially till other threads are spawned with the help of the *fork* operation. The program starts exploiting parallelism as the *team* works in parallel as shown in figure 8.

Fork*:** the master thread creates a ***team of parallel threads.

***Join*:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

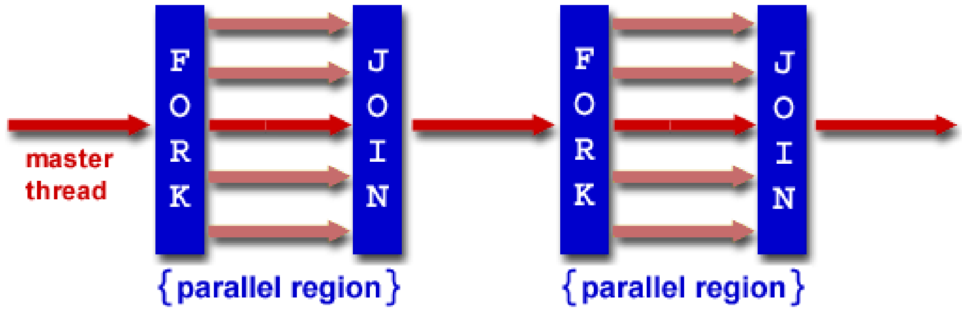


Figure 8: Fork and Join model

Source [https://computing.llnl.gov/tutorials/openMP/]

CUDA

In recent years there is a paradigm change observed with the advent of Graphics Processing Units (GPU) for general purpose computing. NVIDIA CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA (6). Using CUDA the GPUs are accessible to programmer for computation like CPU, that's why GPUs are now being called "co-processors" (7).

CUDA exploits Data Parallelism, where many threads perform the same operation on different data concurrently, also called SIMD (Single Instruction Multiple Data) illustrated in figure 9.

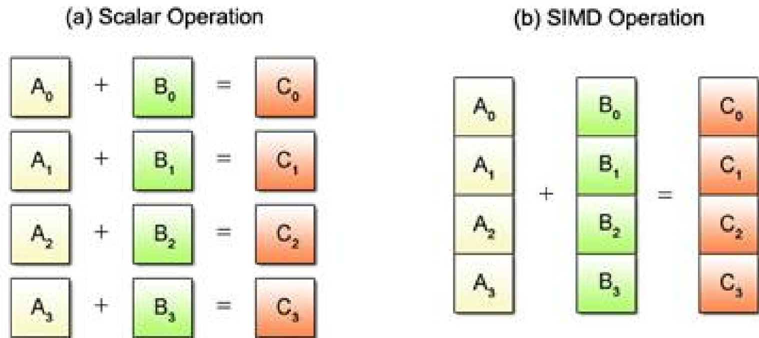


Figure 9: Scalar vs. SIMD Operations

Source [http://www.kernel.org/]

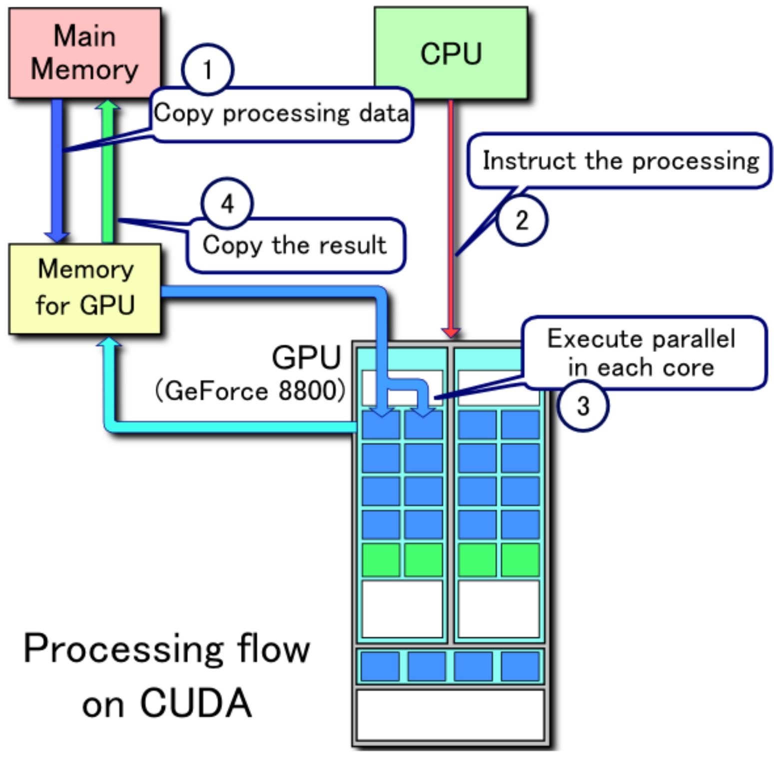


Figure 10: CUDA process flow

Source [http://en.wikipedia.org/wiki/CUDA]

A simple CUDA programs has the following flow, please refer to the figure 10

1. Copy data from main memory to GPU memory.
2. CPU instructs the process to GPU.
3. GPU execute parallel in each core.
4. Copy the result from GPU memory to main memory.

Chapter 3

BACKGROUND AND LITERATURE REVIEW

This chapter discusses the existing tools and technologies that are directly or indirectly related to our domain. Main focus will be on our proposed simulations (Fluidanimate, Oil Reservoir Simulation and Blackscholes,) and their existing implementations and working of their algorithms. We will be discussing each of the simulation one by one in depth.

FLUIDANIMATE

Due to widely growing industry of animation and computer games, the significance of fluid simulation has drastically increased. Different algorithms and techniques are being used to animate the fluid so that a real impression can be achieved. Fluid animate is a particle physics simulation used to animate flow of incompressible fluids using SPH (Smoothed Particle Hydrodynamics) (8) method. This treats fluid as small particles having properties like pressure, velocity, acceleration, density and initial position vector in space. It is an Intel RMS application from PARSEC benchmark. SPH method uses particles to model the state of the fluid at discrete locations and interpolates intermediate values. The main advantage of SPH is automatic conservation of mass due to a constant number of particles.

Fluid Particle Simulation Methods

There are various numerical approaches that are being used to simulate fluid dynamics. Some widely used numerical approaches are mentioned below (9):

Grid Based (Eulerian)

- Stable Fluids
- Particle Level Set

- Particle Based (Lagrangian)
  - SPH (Smoothed Particle Hydrodynamics)
  - MPS (Moving-Particle Semi-Imlicit)

- Height Field
  - FFT (Tessendorf)
  - Wave Propagation – e.g. Kass and Miller
  - Direct Simulation (Monte Carlo) (10)
  - Gas dynamics Flows

In this implementation of fluidanimate problem, we would prefer Particle based SPH technique due to several advantages (11).

- Conservation of Mass is trivial.
- Easy to track free surface.
- Only performs computation where necessary.
- Not necessarily constrained to a finite grid.

Fluidanimate Phases

Below are the five major steps that fluidanimate algorithm performs in each time step.

- ***Initialize Simulation:*** This is the first and foremost step involved in fluid animate. In this phase, particle data is read from file and stored in data structures associated to particles, cells and grids.

- ***Rebuild Grid:*** When the particle data is read in the initialization phase, then a rebuild grid phase starts. In this phase, particles are arranged in a logical data structure named cells and these cells further constitute a 3D grid.

- ***Compute densities and forces:*** The actual compute intensive work is performed in this phase. In this phase, Particle-Particle interactions are calculated which is done in two sub phases. In first phase, the densities of the particles residing in Grid are calculated. The neighbour particles of each particle are calculated and then their effect is computed. When the densities of a particle and its neighbours are updated, then the second phase of force computation begins. In this sub phase, same steps are performed as

compute densities and the force on each particle and its neighbours is calculated.

- ***Process Collisions:*** In this phase a particle-particle and particle to scene geometry collisions are calculated.

- ***Advance Particles***: Finally, due to updated densities and forces along with the collision of particles, the positions of these particles are updated. The particles move in the specified direction in the grid they reside.

Fluidanimate was implemented in PARSEC benchmark on shared memory architectures using pthread. The SPH solver written by PARSEC uses localized kernel, due to which a particle residing in a cell can be influenced by particles residing at maximum of its neighbour cells.

Force Computation Methodologies

There are two approaches which can be used to solve Fluidanimate problem

- Tree Based Approach using Barnes Hut Algorithm

- Sub Grid Partition Based Approach by dividing large Grid into smaller sub grids.

Due to localized effect of particles, we will use second approach in fluidanimate implementation because no *far_field_force* is being applied on particles, only *nearest_neighbour_force* is to be calculated.

OIL RESERVOIR SIMULATION

Reservoir simulation combines use of mathematics, physics and computer programming to develop a tool for predicting hydrocarbon-reservoir performance under various operating conditions (12). In hydrocarbon-recovery projects capital investment of hundreds of millions is at stack, so the risk associated with the selected development plan must be assessed and minimized, therefore, need for reservoir simulation arises.

The use of reservoir simulation is getting pivotal importance in the petroleum industry. Such pervasive acceptance can be attributed to advances in the computing facilities as discussed in chapter 1, advances in reservoir characterization technique (13).

Reservoir Simulation Process

The reservoir simulation process is shown in figure 11 which starts with the formation phase in which information about the reservoir's geological properties is gathers which is then converted into mathematical model in the form of Partial Differential Equations (PDEs) with appropriate initial and boundary conditions. These equations represent important physical processes taking place in the reservoir for example, the flow of fluids partitioned into as many as three phases (oil, water, gas), and mass transfer between the various phases, effect of viscosity etc (12). PDEs obtained are then converted into set of Linear Algebraic Equations or System of Linear Equations (SLEs) and are solved for the unknowns i.e. pressure or saturation.

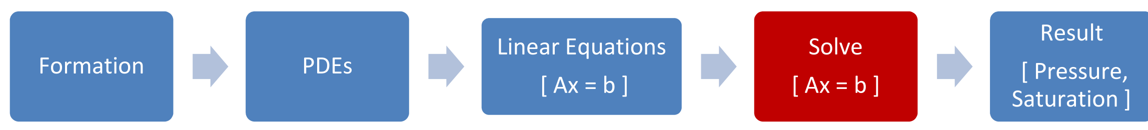


Figure 11: Reservoir Simulation Process

SLE Solvers

The SLEs can be solved using two kinds of methods, below are few examples of each.

I. Direct Methods:
   a. Gauss Elimination
   b. Cholesky decomposition
II. Iterative Methods:
   a. Gauss Seidel
   b. Jacobi Method
   c. Conjugate Gradient

The most time consuming part in the reservoir simulation is the solution of SLEs and this is what we aim to target in this project. For large simulations the number of SLEs increases consequently increasing the size of the Matrix **A** which represents the SLEs. In Oil Reservoir Simulation the matrix A is Symmetric Positive Definite Matrix illustrated in figure 12. Notice that the matrix in figure 12 is a Sparse Matrix – many zero entries. Typically Matrix formed from these PDEs is sparse symmetric positive definite matrices.

A matrix **A** is **positive-definite** if, for every nonzero vector ***x***,

$$x^T Ax > 0$$

A **symmetric matrix** is a square matrix that is equal to its transpose,

$$A = A^T$$

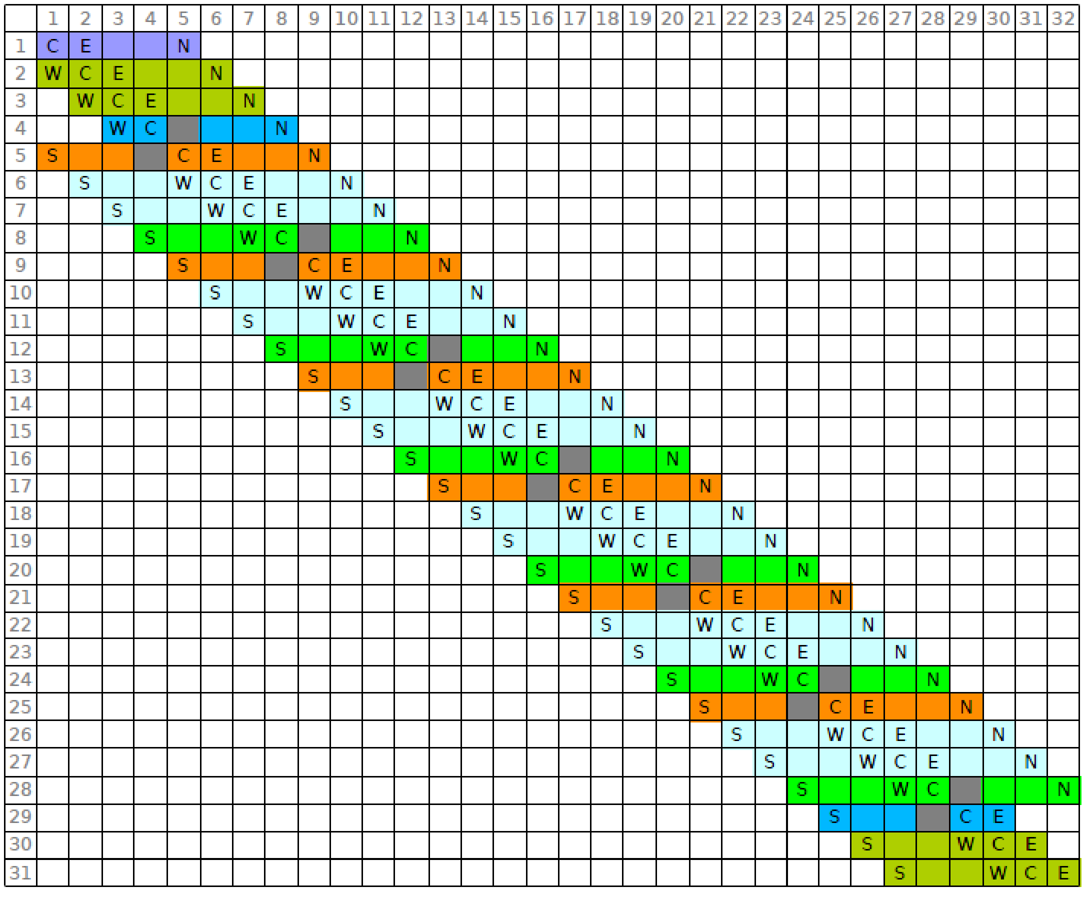


Figure 12: 32x32 Matrix representing a 2D reservoir in 1 Phase

[Figure from Ashraful and Tazrian UAEU HPC Summer School 2010 report]

Direct methods

Direct methods *attempt to solve the problem by a finite sequence of operations. In the absence of rounding errors, direct methods would deliver an exact solution* (14).

The Direct Methods solve the system in fix number of steps. Direct methods are not suitable for large sparse matrices since number of non zero increases which increases computational complexity.

Iterative methods

Iterative method *is a mathematical procedure that generates a sequence of improving approximate solutions for a class of problems. A specific implementation of an iterative method, including the termination criteria, is an algorithm of the iterative method* (14).

The iterative methods are faster than the direct methods as the aim is to approximate the solution which depends on the termination criteria as how much precise results we need.

The iterative methods are faster than the direct methods as the aim is to approximate the solution which depends on the termination criteria as how much precise results we need.

Why Conjugate Gradient Method

Considering this problem we chose to use an iterative method to solve the SLEs, we chose the **Conjugate Gradient Method (CG)** figure 13, because of it being highly optimized for symmetric positive definite matrices and being an iterative method it can be applied to sparse systems to exploit the sparsity, otherwise such systems are too large to be handled by direct methods.

1) $x_0 = 0$
2) $r_0 := d - Ax_0$
3) $p_0 := r_0$
4) $k := 0$
5) $Kmax :=$ maximum number of iterations to be done
6) if $k < kmax$ then perform 8 to 16
7) if $k = kmax$ then exit
8) calculate $v = Ap_k$
9) $\alpha_k := \frac{r_k^T r_k}{p_k^T v}$
10) $x_{k+1} := x_k + \alpha_k p_k$
11) $r_{k+1} := r_k - \alpha_k v$
12) if r_{k+1} is sufficiently small then go to 14 end if
13) $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
14) $p_{k+1} := r_{k+1} + \beta_k p_k$
15) $k := k + 1$
16) $result = x_{k+1}$

Figure 13: Conjugate Gradient Method Algorithm

The Conjugate Gradient Algorithm

Within CG the most compute intensive operation is the Matrix Vector Multiplication (line no. 8 in figure 13) which is $O(n^2)$. From the memory and communication point of view the update of vector ***P*** at the end of each iteration (line no. 14 in figure 13) in distributed environments is time consuming as ***P*** needs to be shared by all processes (15). In the process of optimization these factors will be considered vital.

BLACKSCHOLES

The Blackscholes application is an Intel RMS benchmark used to calculate the prices for a portfolio of European options by using Black-Scholes partial differential equation. Black-Scholes formula is used in computing the value of an option. In some cases, e.g. European options, it gives exact solutions, but for others, more complex, numerical attempts are made in order to obtain an approximation of the solutions. Several numerical methods are used for solving the Black-Scholes equation, e.g. Finite element method (16) and Monte Carlo Option Model.

Blackscholes Algorithm

Blackscholes formula is widely used method for calculating the option prices for a given portfolio of options. There are several assumptions underlying the Blackscholes model of calculating options pricing (17). The Blackscholes model also assumes stocks move in a manner referred to as a random walk; at any given moment, they are as likely to move up as they are to move down. These assumptions are combined with the principle that the options pricing should provide no immediate gains to either seller or buyer.

Chapter 4

CASE STUDY I: FLUIDANIMATE

This chapter includes all the relevant details of the design and analysis of the system, which comprises of main modules that will be implemented and optimized. We will also discuss the significance of the approach followed in parallelizing.

FLUIDANIMATE

By performing the analysis of fluidanimate algorithm and profiling the serial implementation of fluidanimate provided in PARSEC benchmark, we concluded that its parallelization is somewhat more difficult due to 3D particle-particle interaction and high data dependency. The algorithm follow particle-in-cell approach, which means the particles reside in Cells and these Cells then combine and form Grid. The challenge was to divide the Grid or cells into smaller chunks in such a way that work load must be balanced.

Fluidanimate On Distribute Memory Cluster

The parallelization of fluidanimate on DMC (Distribute Memory Cluster) requires much effort. The input file for fluidanimate contains particles attributes e.g. density, viscosity and external acceleration. The data is read from the files and then Cells are formed which contain particles. This happens when simulation is initialized, after that the effect of other particles like force and density are calculated on current particles. Then, based on effect of these affects the particles are moved to new location in 3D grid.

Fluidanimate programe flow for DMCs

A basic flow of fluidanimate is shown in the figure 14 below; the most compute intensive part of the code is highlighted with red colour.

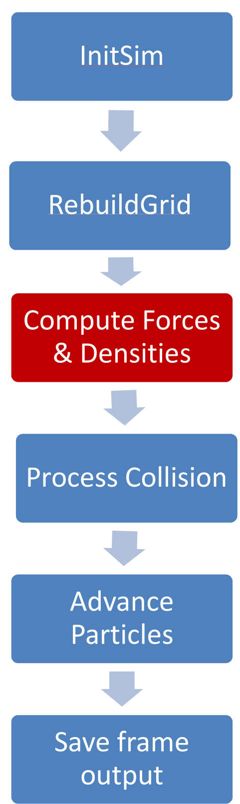


Figure 14: Fluidanimate Parallel Flow

The profiling of Fluidanimate serial implementation resulted that the most compute intensive function is *ComputeForcesDensities* which needs to be parallelized. We will discuss the details of parallel design of fluidanimate later in this chapter.

Fluidanimate design for distribute memory cluster

As the particles reside in the cells which are part of the grid. So, first of all we need to understand the fluidanimate design. Consider the following hierarchy.

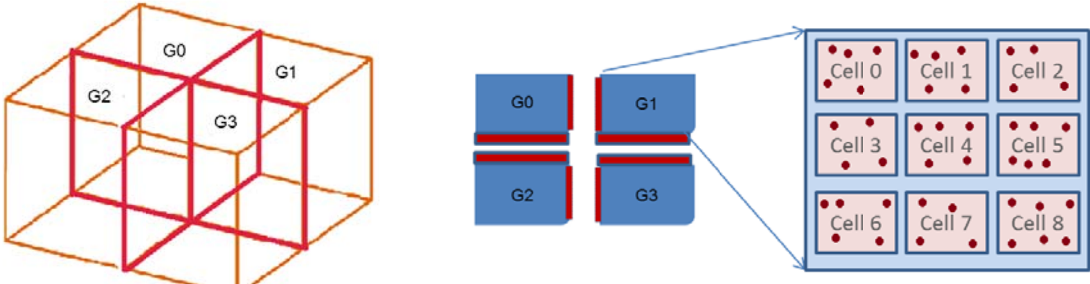
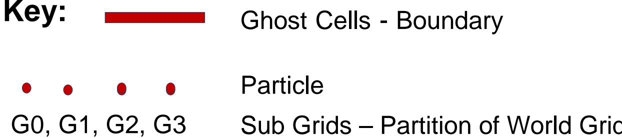


Figure 16: Particles World Grid – 3D

Figure 15: Particles World Grid



In fluidanimate design, above figures illustrate that world Grid is divided into further 3D Sub Grids. Each Sub Grid contains multiple Cells, and each Cell further contains actual fluid particles. The particles in a cell can only be influenced by the particles residing in the neighbouring cells. The effect does not travel more than one cell and no *far_force* is being applied on particles; that is the reason why we stick to the Grid based partition approach rather than tree based partition approach.

Fluidanimate mpi based implementation for DMCs

As discussed in the design of fluidanimate that the major task in parallelization of fluidanimate is to divide World Grid into sub grids and also the boundary cells so that work load should be balanced. We discussed in design that how we partition the World Grid of cells into equal chunks of Sub Grids.

In MPI based parallelization, we assign each of the Sub Grid to corresponding processor, and each processor will be responsible for calculation of particle residing in its own sub grid. The challenge here is that after the work division among processors the data resides in distributed memory which means that the particle data of one processor is not visible to other processor's cell. The only way to share the data is to communicate the required data over the network. The processors in MPI implementation are arranged in Cartesian Topology order, where they will be sharing their boundary cells with their immediate neighbours arranged in Cartesian Topology (18).

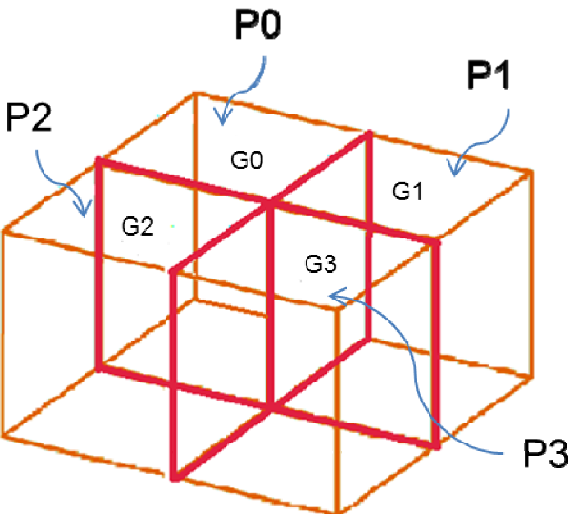


Figure 17: FluidAnimate MPI work Division among processors

In figure 17, we see that there were four processors available, so we divided our World Grid into 4-Sub Grids and assigned each of the Sub Grid to corresponding processor. The Red boundaries represent Ghost Cells also called Boundary Cells that need to be communicated over the network.

A code snippet of Grid distribution among processors is shown below.

```
for(int i = 0; i < NUM_GRIDS; ++i) {
    for(int iz = grids[i].sz; iz < grids[i].ez; ++iz)
        for(int iy = grids[i].sy; iy < grids[i].ey; ++iy)
            for(int ix = grids[i].sx; ix < grids[i].ex; ++ix)
            {
                // Distribute Grids Data to processors....
                // each grids[i] will be sent to ith processor

            }
```

When the data is distributed among the processors in cartesian topology manner. Then each processor find its neighbour processors through cartesian rank and communicate the list of ghost cells. To understand the communication pattern followed by processors, see the communication matrix shown in figure 18 below:

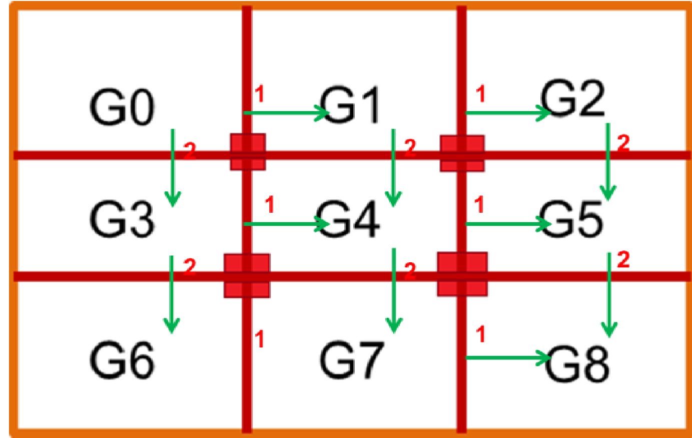


Figure 18: FluidAnimate MPI Communication of Ghost Cells

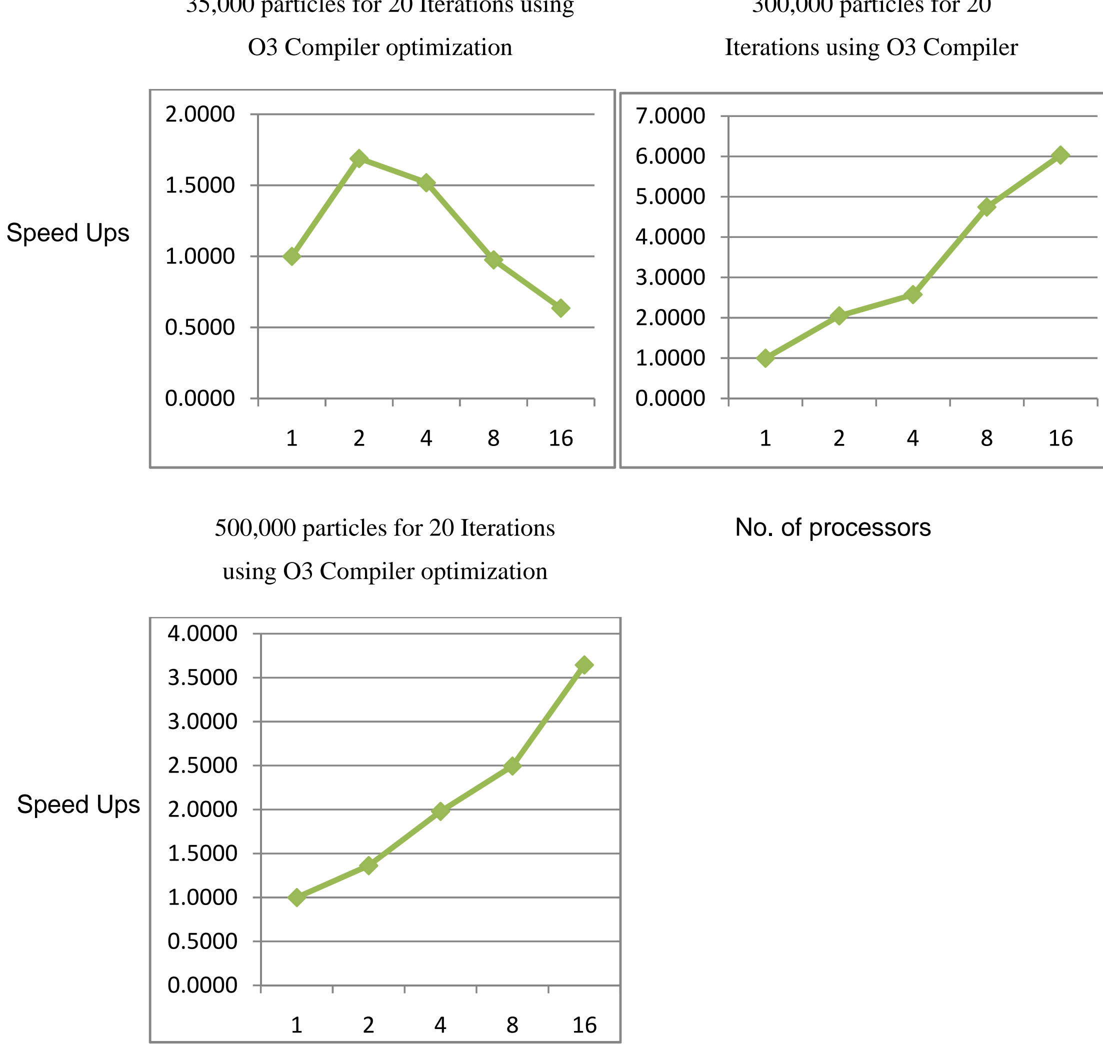
Each processor will calculate its neighbour processors and then start communicating ghost cell lists. For example, in this case, there were nine sub grids assigned to nine processors. Each of them will calculate its peer processors e.g. P0 containing G0 will receive from none in x-direction but will send its boundary cell lists to adjacent processor P1 which contains G1 and same happens with P1, but here P1 will receive from its left neighbour and send to right neighbour P2. Similarly, the same pattern happens in y-direction (in 2-D).

When the communication of ghost cells is done, then each processor will compute effect of particles on its own cells as well as neighbour cells in the form of forces and densities. After that, when current frame finishes and next frame starts, then in *RebuildGrid* the communication of ghost cells happens again. These ghost cells are now updated in previous frame. *RebuildGrid* is responsible for placement of particles in cells based on their updated attributes. One thing should be kept in mind that all the communications that are happening are *Blocking Communications* (19).

Fluidanimate MPI performance evaluation with blocking communication

The benchmarks were conducted on Barq cluster at NUST-SEECS, see appendix B2 for details of the machine.

The data sets for Fluidanimate were varying in nature. The maximum data set was 0.5 million particles. When the experiments were performed, we obtained these results.



Fluidanimate MPI Speed up Results on Varying Data Sets

As we see that we achieve speedups as we increase the number of processors. On smallest dataset of 35,000 particles of fluid, we see irregular behaviour. The reason is that the computation to communication ratio decreases as we increase the processors. Fluidanimate MPI design consists of necessary communications which tend to decrease the performance on small datasets. Although the speedup gain seems acceptable on largest data set of 0.5 million particles but we can't consider it a good speedup because on 16 processors, the speedup gain is about 3.7 which is not enough.

Fluidanimate MPI optimization

As we analysed the code, we concluded that we can overcome the blocking communication by using non-blocking communication. Non-Blocking communication in MPI has several advantages over blocking communication but harder to implement in code (20).

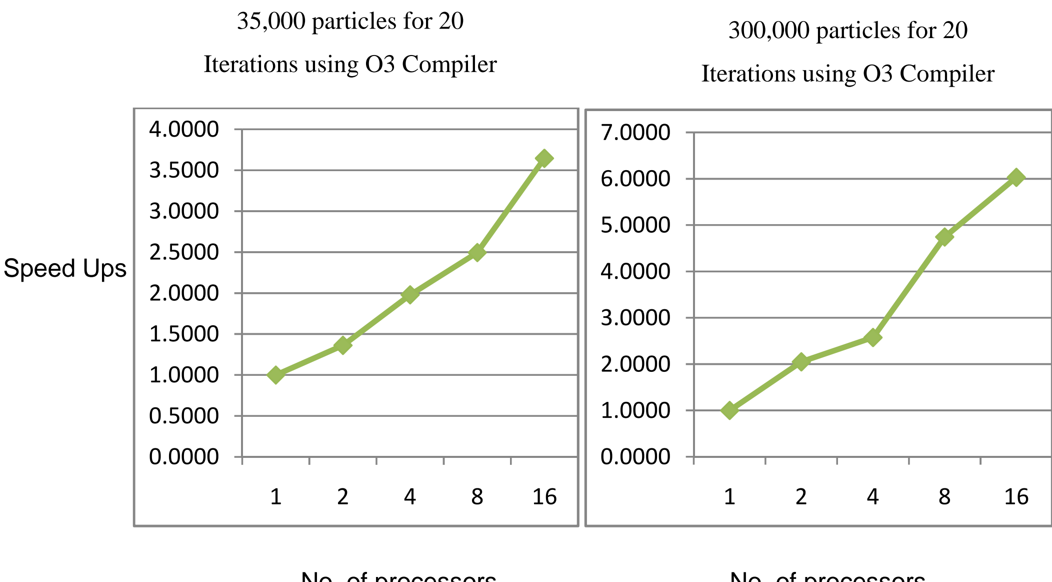
As we implemented the non-blocking communication, a code snippet of non-blocking send is shown below:

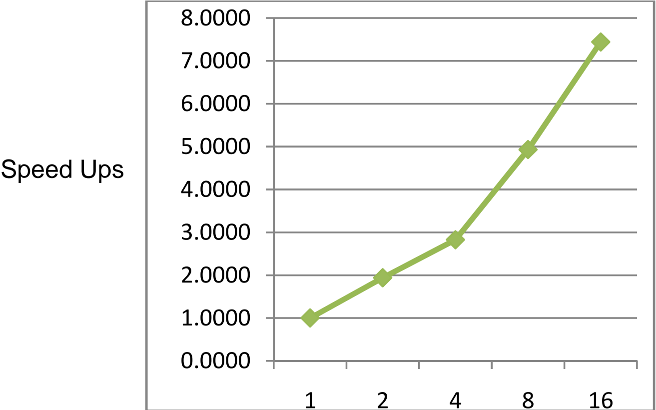
```
MPI_Request request,request2;
MPI_Status status;

MPI_Isend( sBuff, iChunk, cellDataType, i, 500,
           MPI_COMM_WORLD,request );
MPI_Isend(cnumPars2_s, iChunk, MPI_INT, i, 600, MPI_COMM_WORLD,
           &request2 );

MPI_Wait(&request,&status);
MPI_Wait(&request2,&status);
```

In this code, data distribution is being done in non-blocking manner.

Fluidanimate MPI performance evaluation with non-blocking communication

As we implemented the code with non-blocking communication and analysed it, we observed good speedups. We performed the experiments on Barq *(see appendix B2)*. The maximum data set was 0.5 million particles. When the experiments were performed, we obtained these results.

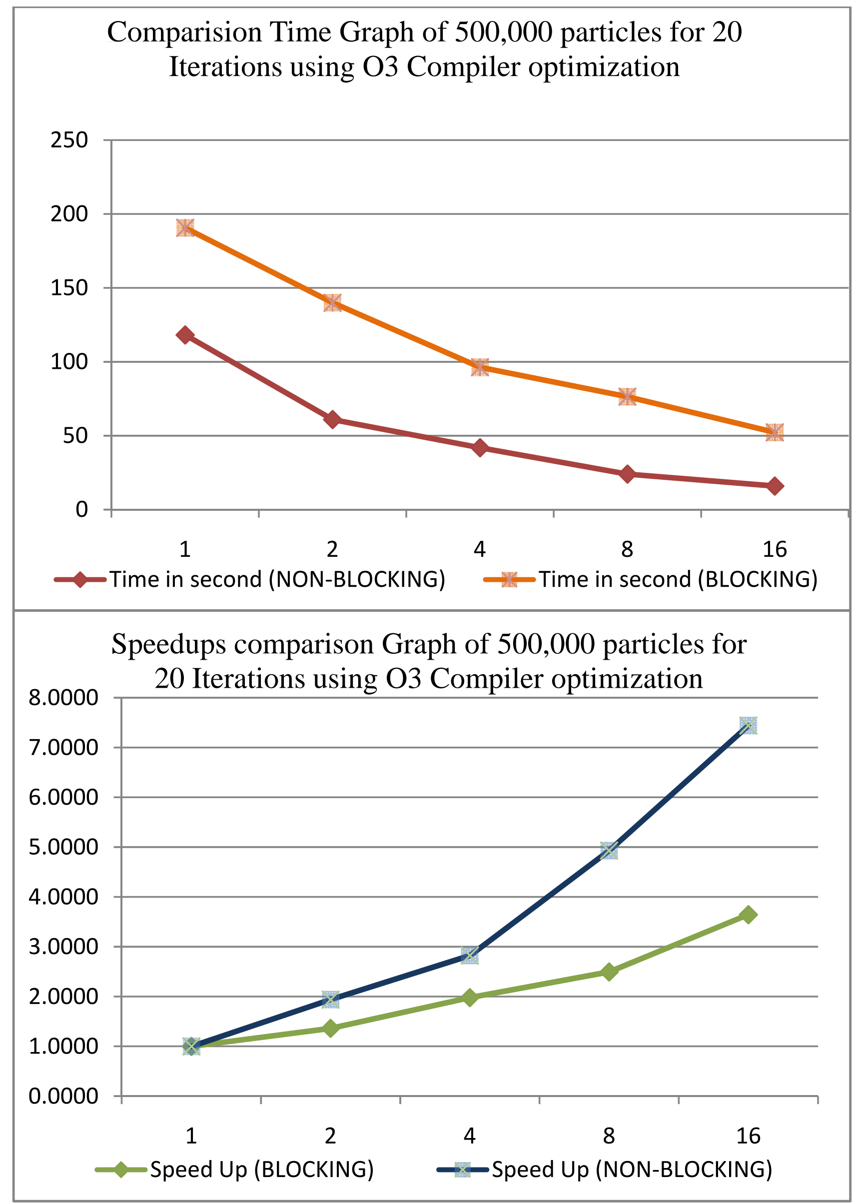




Fluidanimate MPI Non-Blocking Speed up Results on Varying Data Sets

Now as we see that using Non-Blocking communication the speedup is increased from 3.7 to 7.6 on maximum data set of 500,000 particles. As we see that we achieve speedups as we increase number of processors.

A comparison of fluidanimate Blocking and Non-Blocking approach is shown inthe graph below:



Fluidanimate MPI blocking vs. Non-Blocking Time and Speedup comparison

We clearly see that by using Non-Blocking communication in MPI implementation of fluidanimate, we achieved significant time decrease and good speedups. Hence prove our hypothesis; using non-blocking communication in application with less computation to communication ratio causes significant increase in speedups.

Fluidanimate On Shared Memory Architectures

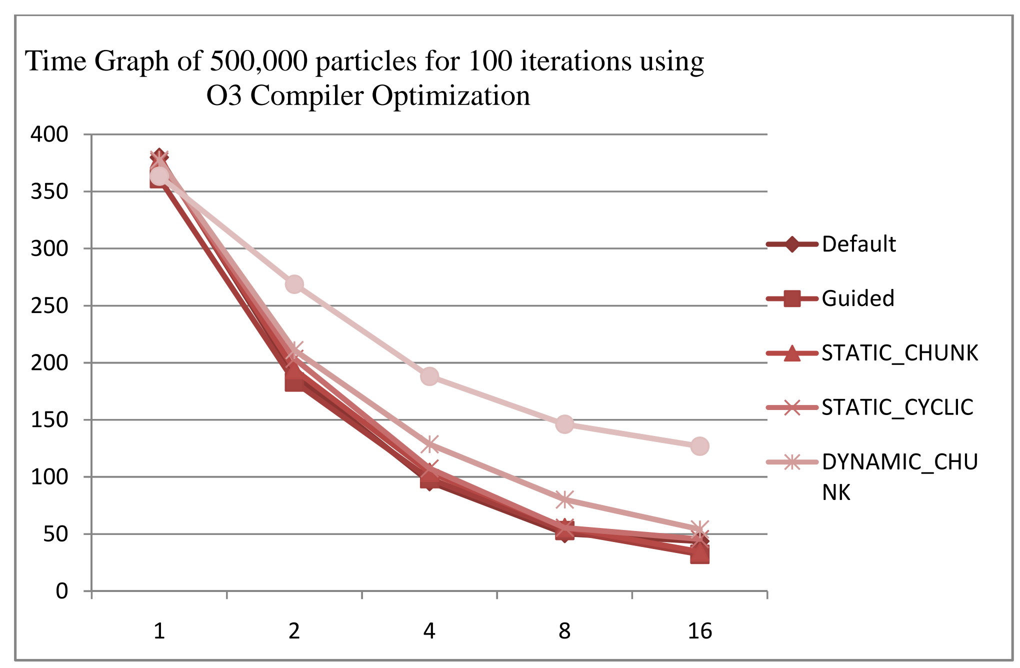
The parallelization of fluidanimate on SMA (Share Memory Architecture) can be done on different shared memory programming models like posix threads, openmp and cilk++. We have chosen OpenMP to parallelize fluidanimate application for SMAs due to *fork-join* model of OpenMP threads. The input file for fluidanimate contains particles attributes e.g. density, viscosity and external acceleration. The data is read from files and then Cells are formed which contains particles. This happens when simulation is initialized, after that the effect of other particles is calculated on current particles. Then, based on effect of these forces the particles is moved to new location in 3D grid.

Deployment testbed for SMA

We have conducted experiments on our test bed which we call *Raad*. – see appendix [B3] for detail of the machine.

Fluidanimate OpenMP design and implementation

By analysing the fluidanimate serial code, we suggest our first approach which is default parallelization using OpenMp *parallel for* constructs for loop parallelization. In this approach, OpenMP divides the total cells into chunk of cells and then assigns each chunk to corresponding thread. Each thread will be responsible for calculation on its chunk. There are different OpenMP scheduling techniques are available e.g. static, dynamic, default and guided. We implemented our proposed naïve implementation by using different scheduling techniques and then chose the best scheduling technique out of it and performed experiments.



OpenMP Scheduling techniques Comparison on Fluidanimate

The above results are from fluidanimate using different OpenMP scheduling techniques. We observed different behaviour of application under different scheduling algorithms. In fluidanimate, dynamic scheduling algorithm was performing poorly because dynamic scheduling performs well when there are non-uniform loops (21).

Since, the static scheduling technique has the least runtime overhead (21) as well as it is performing well in fluidanimate implementation so we will go with static scheduling algorithm.

A code snippet from *ComputeForces* of fluidanimate by OpenMP loop parallelization approach is shown below:

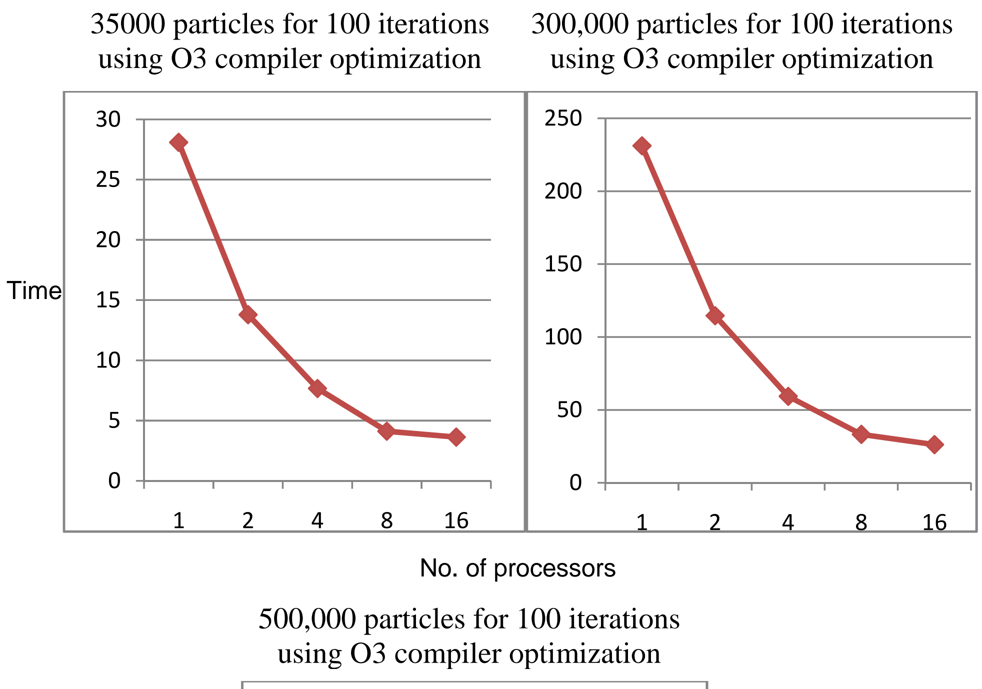
```
#pragma omp parallel for shared(numCells, cells, cnumPars, nx, ny,
nz)private (ck, cj, ci, cindex) schedule (static, CHUNK)

	for( ck = 0; ck < nz; ++ck)
		for( cj = 0; cj < ny; ++cj)
			for( ci = 0; ci < nx; ++ci) {
			// computation is done in parallel.
	}
```

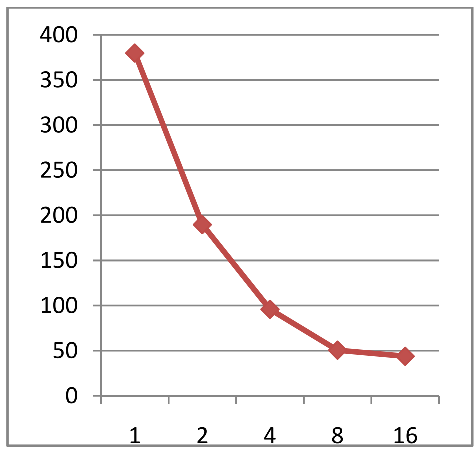
The most important thing to consider is that the memory is shared between threads, so synchronization problem can happen in shared environment. The section of the code which more likely needs synchronization is where cells are updated. We need to take care of boundary cells like when we are updating cells we have to check whether the cell is on boundary or not, if it is on boundary then we must maintain some lock to prevent access by other threads.

```
omp_lock_t my_lock;
omp_init_lock(&my_lock);

if(border[index])

{
	#pragma omp critical
	{
		omp_set_lock(&my_lock);
			cell.a[ipar] += acc;
			neigh.a[iparNeigh] -= acc;
		omp_unset_lock(&my_lock);
	}
}
else
	cell.a[ipar] += acc;
	neigh.a[iparNeigh] -= acc;

omp_destroy_lock(&my_lock);
```

Fluidanimate OpenMP performance evaluation by loop parallelization – naïve approach

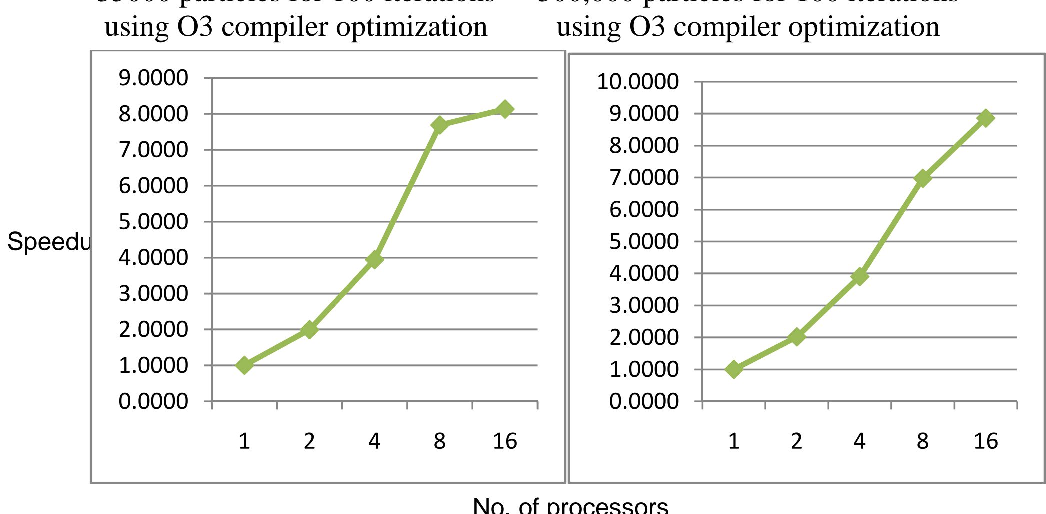
The benchmarks were conducted on Raad SMP machine at NUST-SEECS, see appendix B3 for details of the machine. The naïve approach results are given below:





Fluidanimate OpenMP naïve implementation Time Graph on varying data sets

We observe a significant decrease of execution time by increasing no. of processors. The speed up graph is shown below:



500,000 particles for 100 iterations using O3

10.0000
9.0000
8.0000
7.0000
6.0000
5.0000
4.0000
3.0000
2.0000
1.0000
0.0000

1 2 4 8 16

Fluidanimate OpenMP naïve implementation Speedup on varying data sets

As we see the speedups are increasing as the number of processors are increased. But we see that the speedups are not too much as we expected. The main reason behind it the cache locality. The analysis of the approach showed that the speedups were not up to the expectation because of larger cache miss rate.

Fluidanimate OpenMP optimized implementation

With reference to design of fluidanimate *(see fig 15 & 16)*, we proposed a local sub grid based strategy to address the problem of poor cache utilization. To do so, we will partition the World Grid into smaller Sub Grids and each OpenMP thread will be operating on its localized sub grid to better utilize the cache. See the figure 19 below;

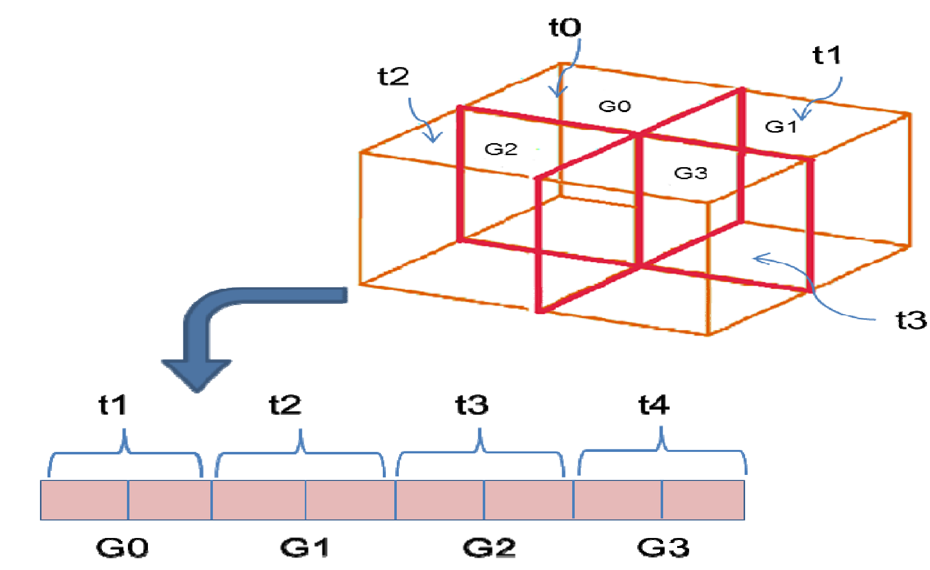
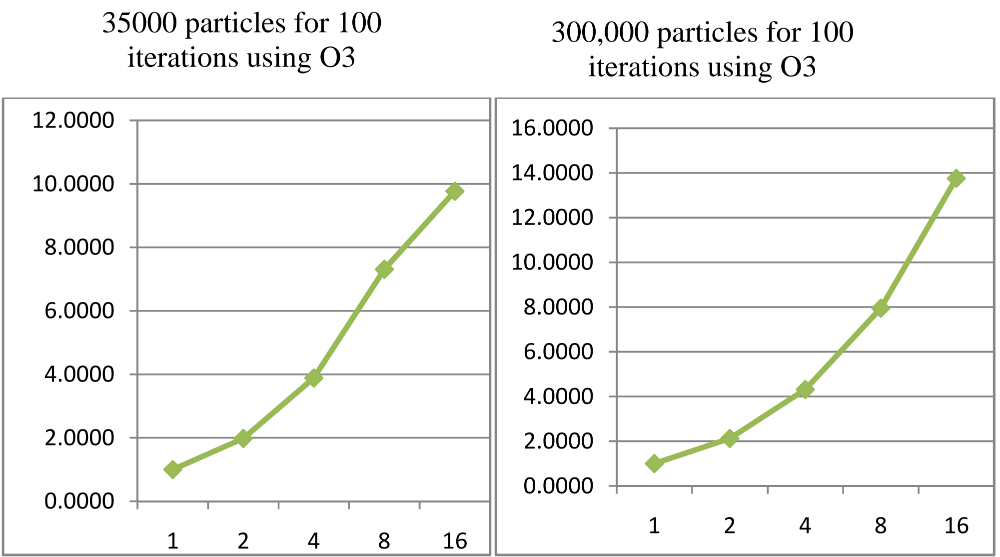


Figure 19: Fluidanimate OpenMP Optimized implementation Design

Fluidanimate OpenMP optimized implementation results

We optimized the implementation and performed experiments on our *Raad* TestBed [B3]. The results were better than the naïve approach that we implemented earlier. Due to increased cache locality, application performed well and the speed ups were close to the expectation. Consider following speedup graphs:

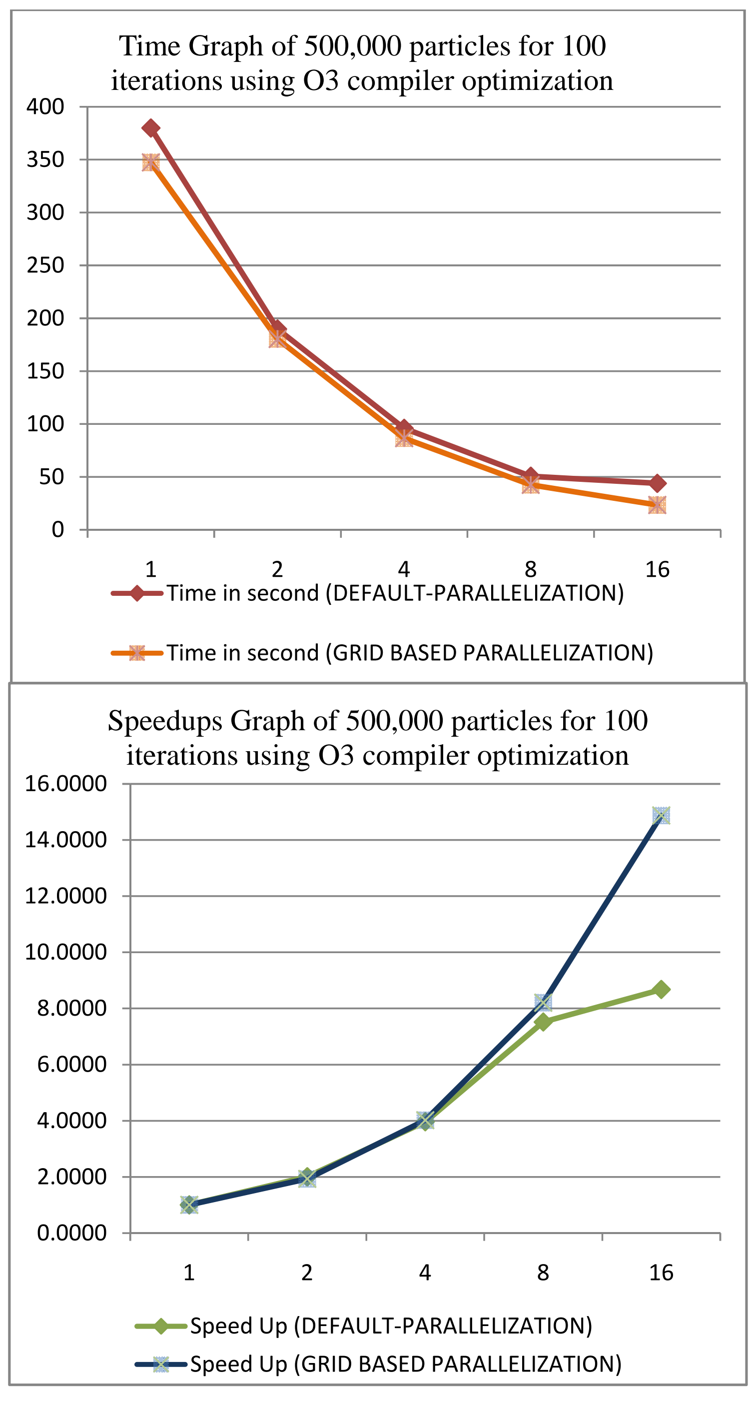


500,000 particles for 100 iterations using O3

16.0000
14.0000
12.0000
10.0000
8.0000
6.0000
4.0000
2.0000
0.0000
1 2 4 8 16

Fluidanimate OpenMP Optimized implementation Speedup on varying data sets

We see that using Sub Grid based partitioning approach and manually launching threads on each sub grids improves the cache locality and hence speedups are almost double than the naïve approach.



Fluidanimate OpenMP naïve vs. optimized Time and Speedup comparison

We clearly see that by optimizing cache locality in share memory application we can significantly increase the performance. The speedup graph of fluidanimate at 16 processors using naïve approach does not scale well, but optimized version is much scalable on greater number of processors.

Fluidanimate On Graphics Processing Units (GPUs)

GPU based implementation of fluidanimate is done using cuda. As discussed earlier in Design of fluidanimate that particles reside in Cells and then these Cells form World Grid. In our Cuda implementation of fluidanimate, each thread is being operated on a cell. It computes the forces and densities of residing particles of cell.

Fluidanimate GPU implementation phases

A basic flow of CUDA based approach is shown in the figure 20. The most compute intensive parts are highlighted with red colour.

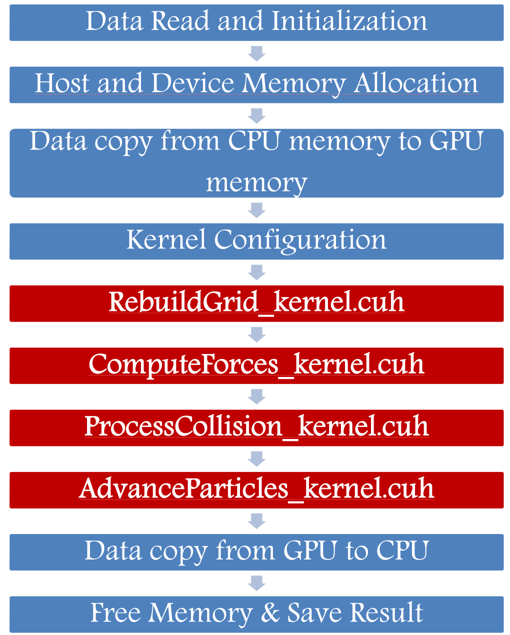


Figure 20: Fluidanimate CUDA Program Flow

Fluidanimate GPU design and implementation

The GPU execution kernel is divided into multiple kernels. Each phase of fluidanimate is a smaller kernel. The main reason to further divide the kernel is that we need synchronization of multiple thread blocks after each phase. When we divide phases into multiple kernels, then synchronization among multiple thread blocks is achieved.

The design of the application is such that CUDA threads operate on Cells and calculates the forces on the particles of that cell. Since Global memory of device is shared by all the threads, so locks are necessary to avoid race conditions.

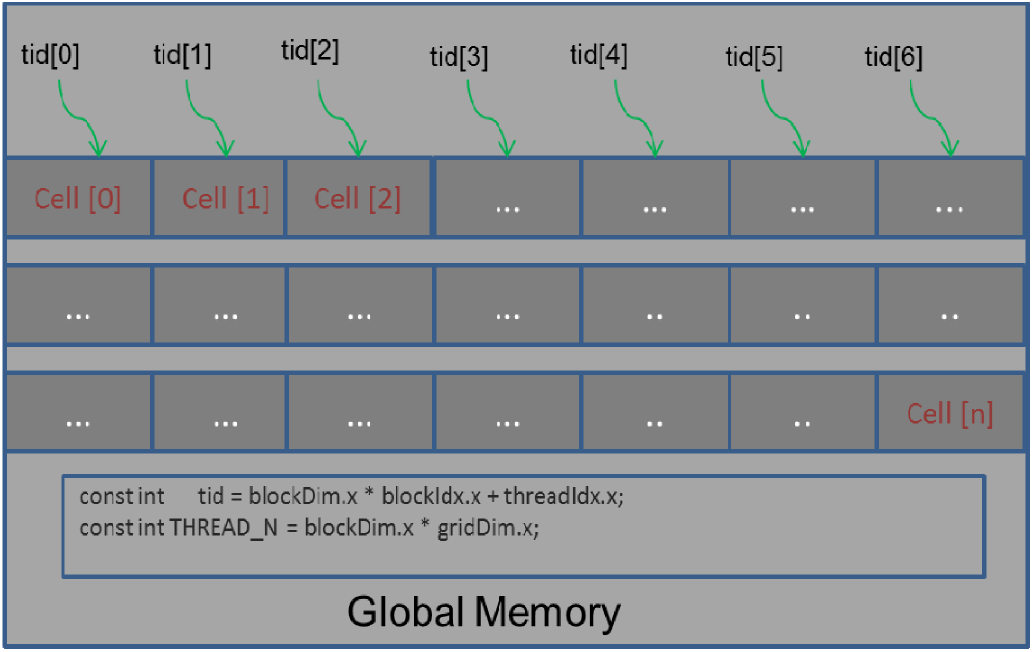


Figure 21: Fluidanimate CUDA Design

In implementation of fluidanimate CUDA version, we first allocate memory on CPU as well as on GPU. Following code allocates memory on CPU and GPU respectively:

```
cells = (Cell*) malloc(numCells*sizeof(Cell));
cnumPars = (int*) malloc(numCells*sizeof(int));
```

Memory Allocation on CPU

```
cutilSafeCall(cudaMalloc((void **)&d_cells,(numCells*sizeof(Cell)) ) );
cutilSafeCall( cudaMalloc((void **)&d_cnumPars, (numCells*sizeof(int)) ) );
```

Memory Allocation on GPU

After memory allocation, we read particles data in CPU memory. Then next step is to copy data from host (CPU) memory to device (GPU) global memory where CUDA threads can have access to it. Following code transfer memory from host to device.

```
cutilSafeCall( cudaMemcpy(d_cells, cells, numCells*sizeof(Cell), cudaMemcpyHostToDevice) );

cutilSafeCall( cudaMemcpy(d_cnumPars, cnumPars,numCells*sizeof(int), cudaMemcpyHostToDevice) );
```

Then we launch Kernels which will do the computation of different phases of fluidanimate.

```
RebuildGrid_kernel <<< numBlocks, threads_per_block >>>
      (d_cells, d_cells2, d_cnumPars, d_cnumPars2 ,
      numCells, nx, ny, nz, domainMin, domainMax, delta);


ComputeForces_kernel <<< numBlocks, threads_per_block >>>
      (d_cells, d_cnumPars, numCells, nx, ny, nz,
      domainMin, domainMax, delta, externalAcceleration, hSq,
      densityCoeff, h, pressureCoeff, doubleRestDensity,
      viscosityCoeff );


ProcessCollisions_kernel <<< numBlocks, threads_per_block >>>
      (d_cells, d_cnumPars, numCells, domainMin, domainMax,
      timeStep);


AdvanceParticles_kernel <<< numBlocks, threads_per_block >>>
      (d_cells, d_cnumPars, numCells, timeStep);
```

A code snippet from *ComputeForces_kernel* is shown below:

```
__global__ void ComputeForces_kernel (...) {

      int tid = blockIdx.x * blockDim.x + threadIdx.x;
      const int totalThreads = blockDim.x * gridDim.x;

      /* Compute Forces - phase 01 */
      /* Synchronize Threads */

      __syncthreads ();

      /* Compute Forces - phase 02 */
```

When all the phases of Fluidanimate are completed then finally the data from device global memory is copied back to host memory.

```
cutilSafeCall( cudaMemcpy(cells, d_cells, numCells*sizeof(Cell),
               cudaMemcpyDeviceToHost) );
cutilSafeCall( cudaMemcpy(cnumPars, d_cnumPars,
               numCells*sizeof(int),
               cudaMemcpyDeviceToHost) );
```

Finally, the frame output is stored in an output file and next frame is computed.

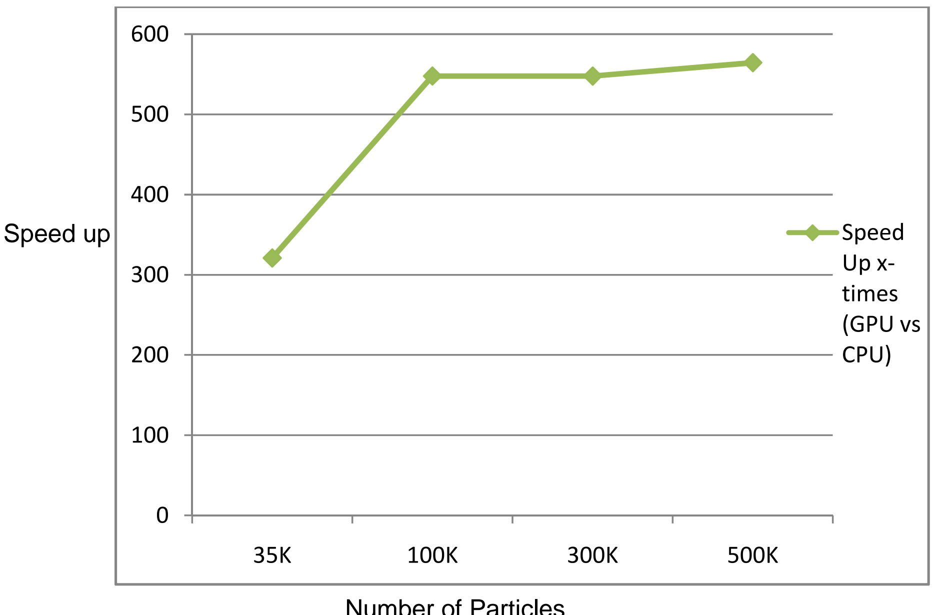
Fluidanimate GPU implementation results

The benchmarks were conducted on CUDA Test bed at NUST-SEECS, see appendix B4 for details of the machine.

The experiments were performed on three different sized data sets named *simsmall* having 35,000 particles, *simmedium* having 100,000 particles, *simlarge* having 300,000 particles and *native* having 500,000 particles. The speedups with comparison to CPU are shown below:

Table 1

Num Particles	GPU Time (msec)	CPU TIME(msec)	Speedup
35K	80.234001	25758	321.03597
100K	124.205	68053	547.90869
300K	391.069	214233	547.81381
500K	639.15302	360844	564.5659



Fluidanimate CUDA Speedup over CPU

As we see in the speedup graph that on smaller data sets of 35k the time difference between CPU and GPU is not big and hence the speed is just fine. But as the data set increases from 100,000 particles to 500,000 particles, we see a significant increase in speedups. Hence our GPU approach results better than that of GPU.

Chapter 5

CASE STUDY II: OIL RESERVOIR SIMULATION

This chapter includes all the relevant details of the design and analysis of the system, which comprises of main modules that will be implemented and optimized. We will also discuss the significance of the approach followed in parallelizing.

OIL RESERVOIR SIMULATION

` Oil Reservoir Simulation was implemented on all three HPC architectures discussed in chapter 1. In this chapter we discuss the implementation of the simulator with emphasizes on CG and its optimization and the study of benchmark results.

The implementation can be divided in two parts,

I. The Simulator
II. The Solver

The simulator is portion of the code which models the reservoir on the basis of grid dimensions, geological rock properties, properties of the fluids and forms system of linear equations which is sent to the solver for the solution. If the simulation is to be done for multiple time steps, the simulator prepares the system of linear equations again and sends to the solver for solution. Once the solution till desired time steps is found the simulator stores the results in a file, ready to be analysed. See figure 22.

Figure 22: Architecture Diagram of Oil Reservoir Simulator

The system of linear equations prepared by simulator is stored in a Matrix Market (MM) format, see appendix A1 for more details on MM format.

Oil Reservoir Simulation on Distributed Memory Clusters

The aim is to design and implement an efficient parallel CG with minimum degrading effect of inter process communication and better load balancing.

There are two versions of CG in MPI,

I. Naive - with blocking communication.
II. Overlap computation and communication

Naive - with blocking communication

The first implementation is naive in a sense that the MPI inter process communication is not done cleverly because at the end of each iteration of CG vector ***P*** is gathered at **root** node and then broadcasted and this collective communication is entirely blocking.

Domain Decomposition

Suppose the System of Linear Equations is the matrix A of figure 23. The domain is decomposed in the form of horizontal blocks. The horizontal blocks are rows distributed to processes on the basis of ***number of nonzeros***, refer to figure 24, for the sake of demonstration there are 4 MPI processes. We achieved proper load balancing; refer to the load balance test graph below, figure 25.

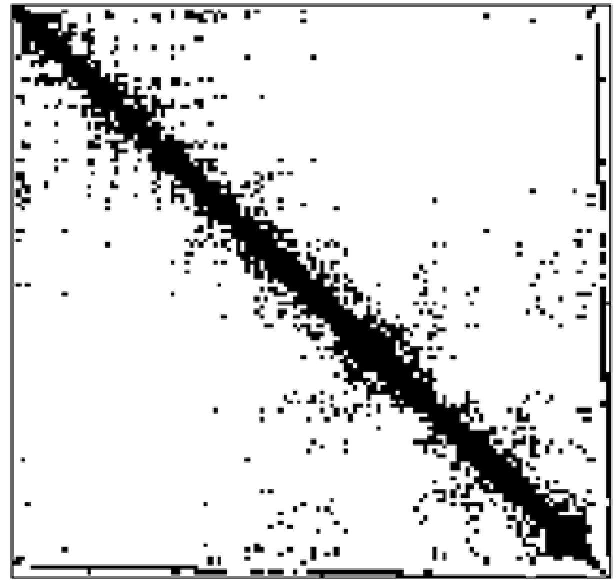


Figure 23: Sample matrix A (sparsity view).

[From the University of Florida Sparse Matrix Collection]

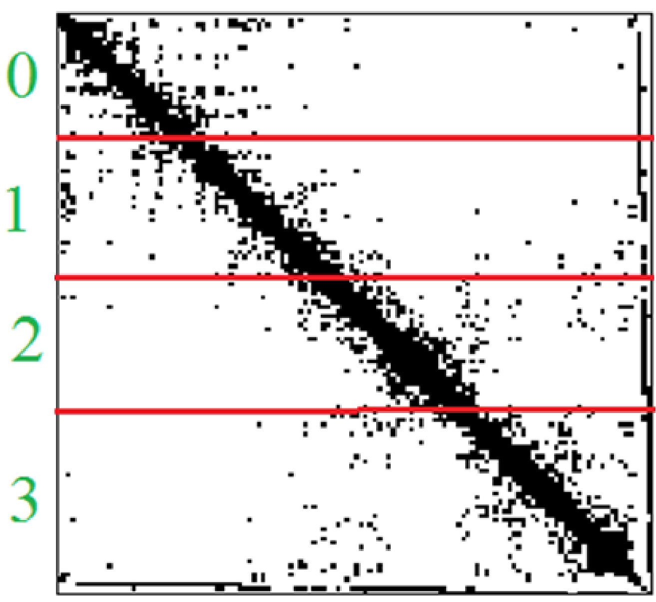


Figure 24: Matrix A decomposed into horizontal blocks among 4 processes

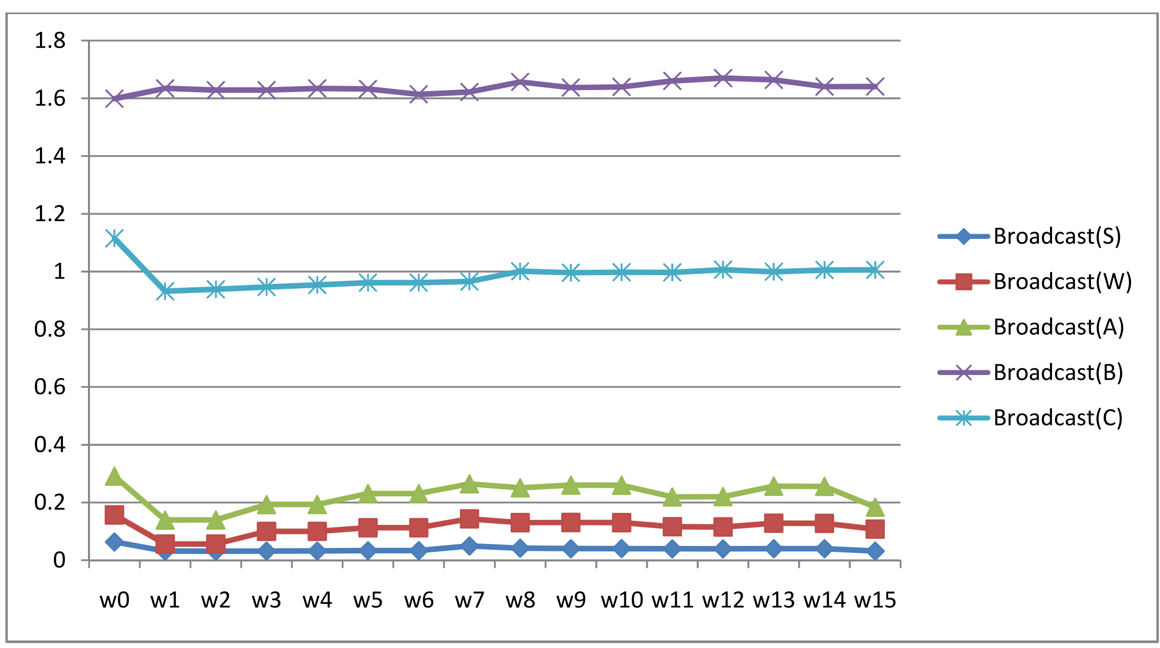


Figure 25: Test for Load Balance

Compute Intensive Sections

As discussed earlier the most compute intensive section of the matrix-vector multiplication (MVM, line no. 8 in figure 13, $\boldsymbol{v} = \boldsymbol{A}.\boldsymbol{p}$) which is $O(n^2)$, and there are four vector-vector multiplications (VVM, line no. 9 and 13 in figure 13) which is O(n). For the VVM every process calculates its own chunk of vector-vector product which is a scalar and a reduction is performed at the root node. Likewise the MVM is performed on every process which calculates its own chunk i.e. a portion of vector; this process is illustrated in figure 26.

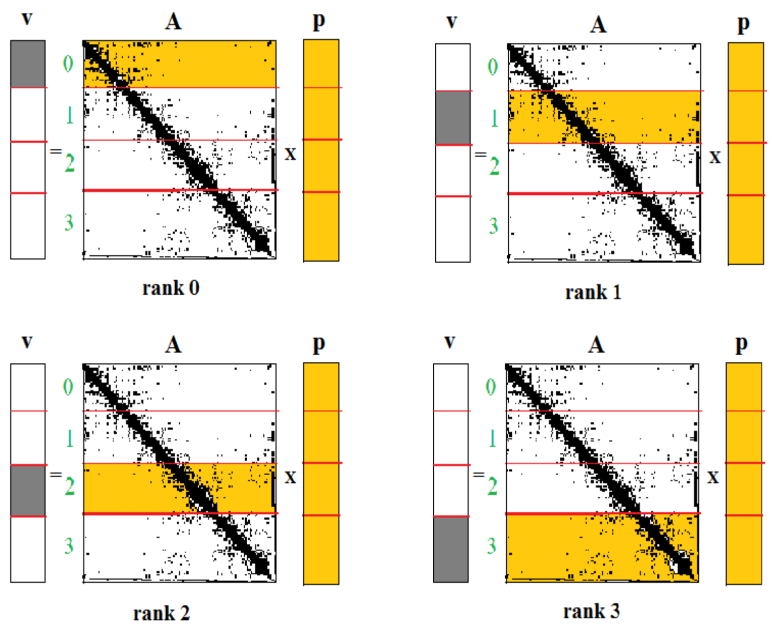
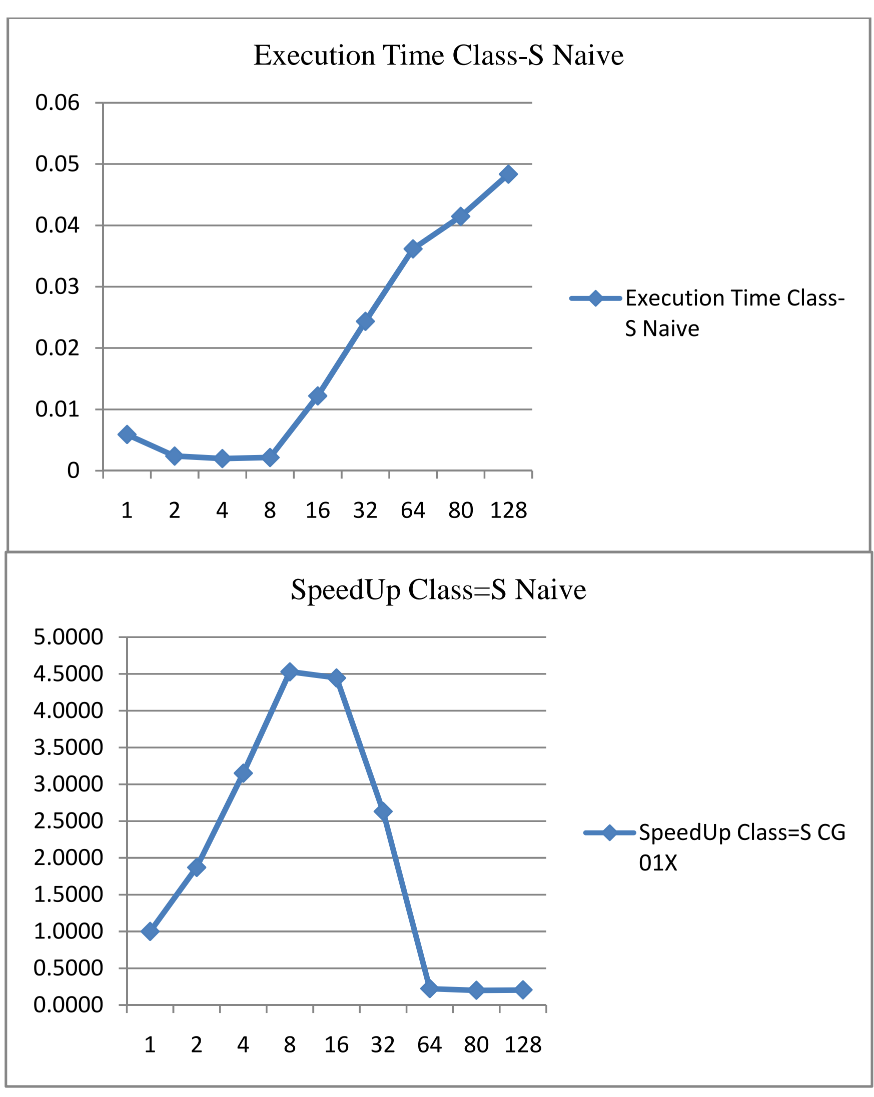


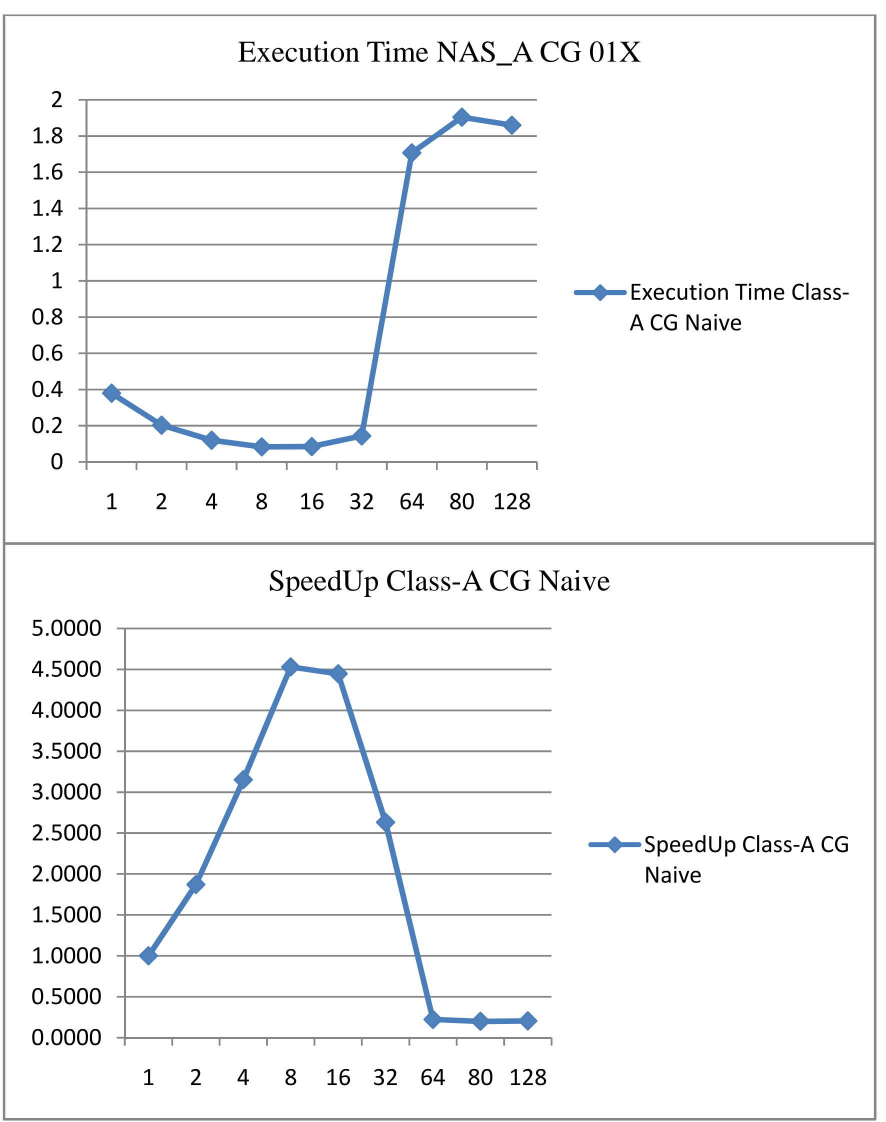
Figure 26: Matrix Vector Multiplication – Naïve

Benchmark results

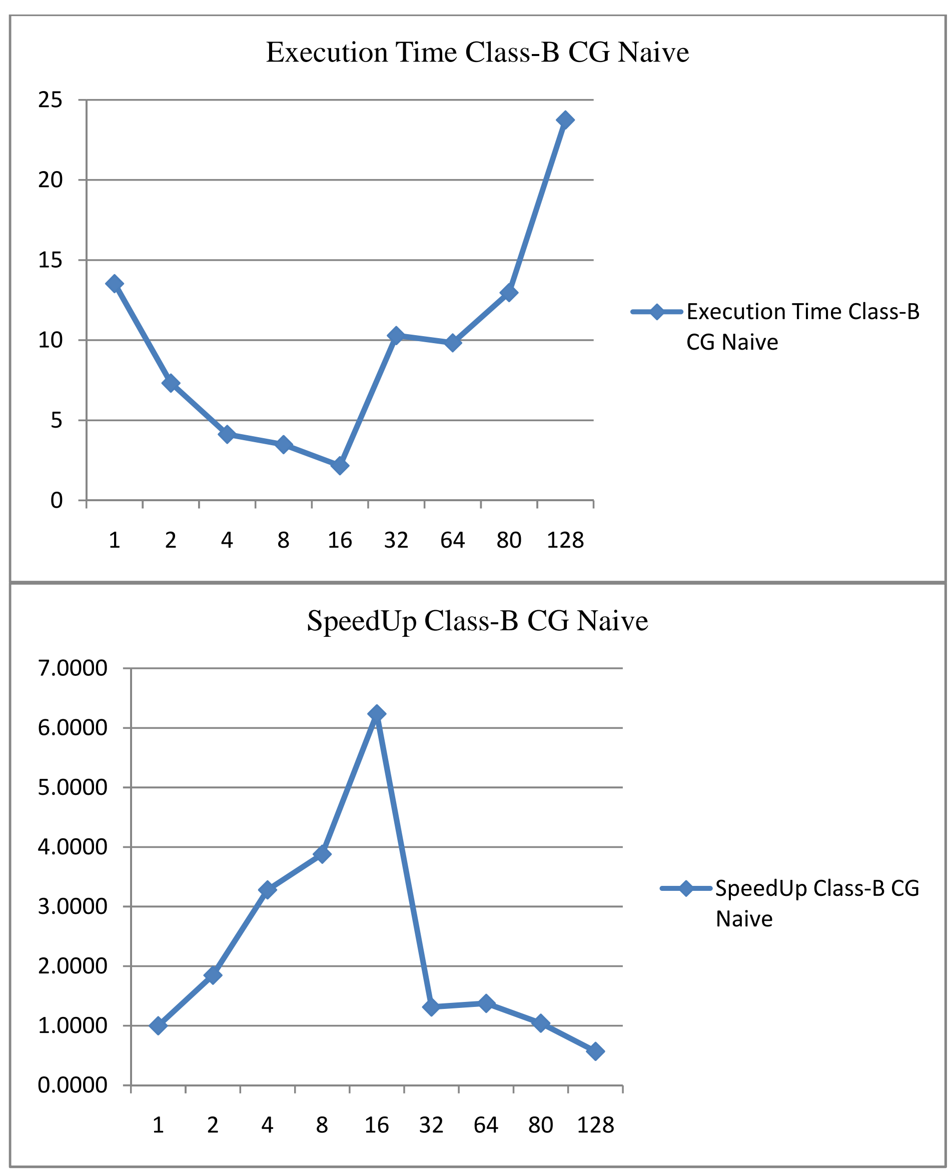
The benchmark for the solver was conducted using a standard data set from NAS Parallel Benchmarks (NPB) (22) on the supercomputing facility at UAEU, see appendix B1 for details.



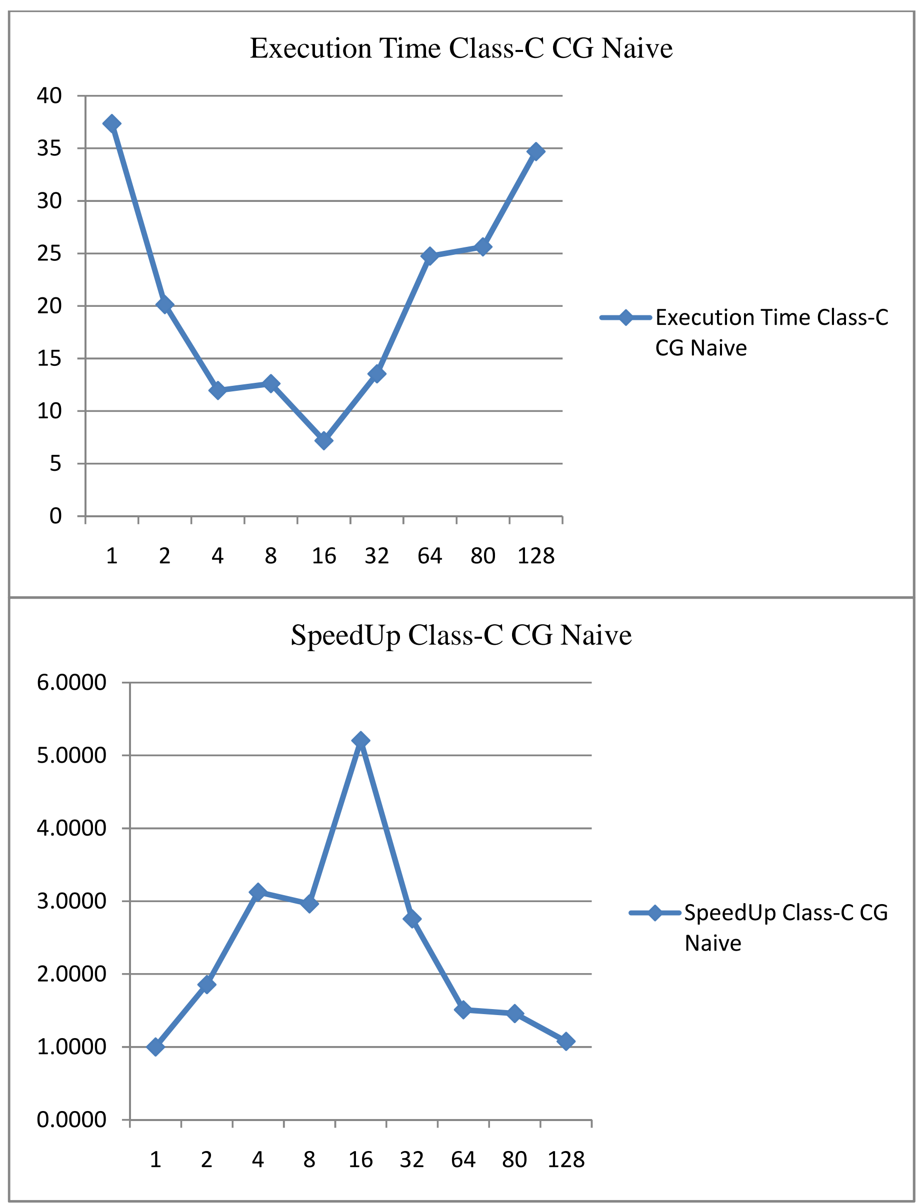
Dataset: CLASS-S, size=1400, iteration=15, NNZ = 78148



Dataset: CLASS-A, size=14000, iteration=75, NNZ = 1853104



Dataset: CLASS-B, size=75000, iteration=75, NNZ = 13708072



Dataset: CLASS-C, size=150000, iteration=75, NNZ = 36121058

Discussion on results

As anticipated the blocking communication deteriorates the performance as number of processors increase. With Class-S the scalability is observed till 8 processes, with Class-A till 32 processes but when the data set increases it affects the performance directly, larger the vector ***p*** more time the processes will consume in waiting than performing work. So with the Class-C speedups are witnessed till 16 processors. When more processes are added its performance decreases. Stable load balancing was also achieved confirming that the data (work) was distributed approximately the same.

Overlap computation and communication

To solve the problem of waiting caused by blocking communication we use non blocking communication to overlap computation and communication. With this approach processes don't have to wait for a gather and a broadcast after doing MVM but rather the communication is performed at the time of MVM. This approach is same as discussed in (23). To implement it there was a change needed in the domain decomposition strategy which will be discussed below.

Domain Decomposition

Like in the first approach (Naive) the matrix A is distributed among processes in the form of horizontal blocks, to accommodate the computation and communication overlapping strategy the horizontal blocks are divided into vertical blocks, four in our case as illustrated in figure 27.

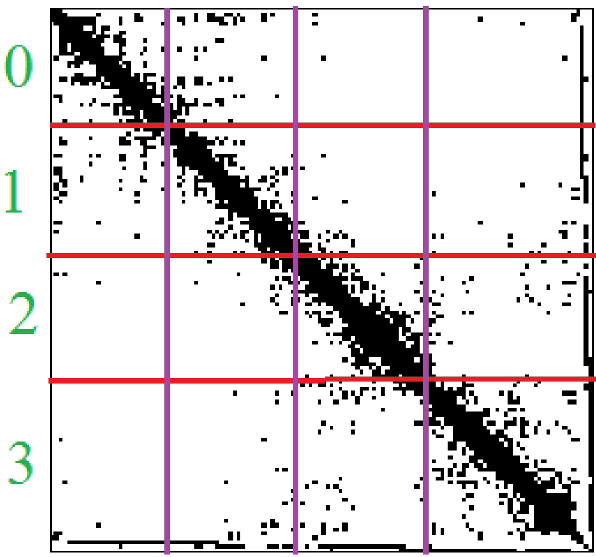


Figure 27: Matrix A further divided into vertical blocks

Compute Intensive Sections

As discussed earlier our main aim is to overlap the computation of MVM and communication of vector ***p***. This time instead of traversing the entire row at a time and multiplying it with the vector ***p*** we multiple it block after block, accumulating the result in the resultant vector. The entire MVM completes in 4 steps in our case and generally in **no. of processes** steps. The processes are logically arranged in ring fashion, at the beginning of each step, before MVM there is a non-blocking communication between neighbours in which each process sends portion of vector ***p*** to its adjacent neighbour (rank-1) and receives from the other neighbour (rank+1), illustrated in figure 28.

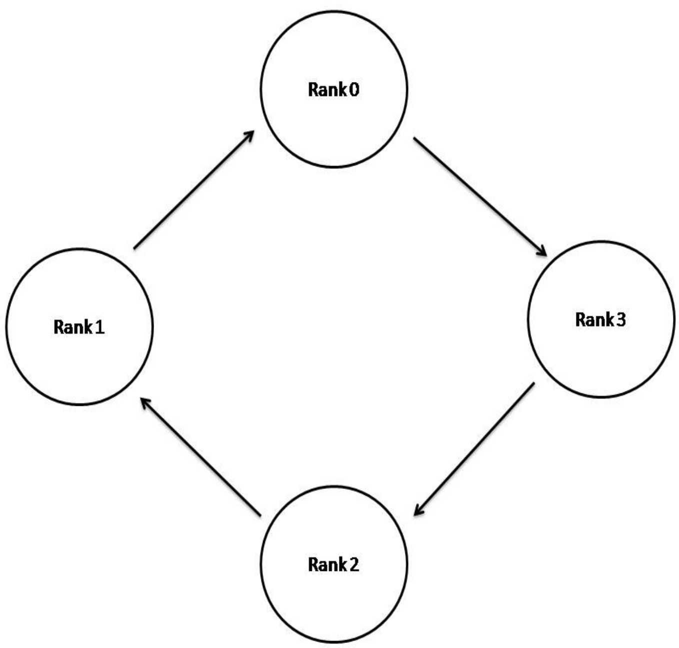


Figure 28: Communication in a ring

To put this in words:

- Initially, rank 0 has chunk p_0, rank 0 sends p_0 to rank 3.
- Initially, rank 1 has chunk p_1, rank 1 sends p_1 to rank 0.
- Initially, rank 2 has chunk p_2, rank 2 sends p_2 to rank 1.
- Initially, rank 3 has chunk p_3, rank 3 sends p_3 to rank 2.

Where vector $\boldsymbol{p} = p_0 + p_1 + p_2 + p_3.$

The MVM and communication for step 1 is illustrated in figure 29.

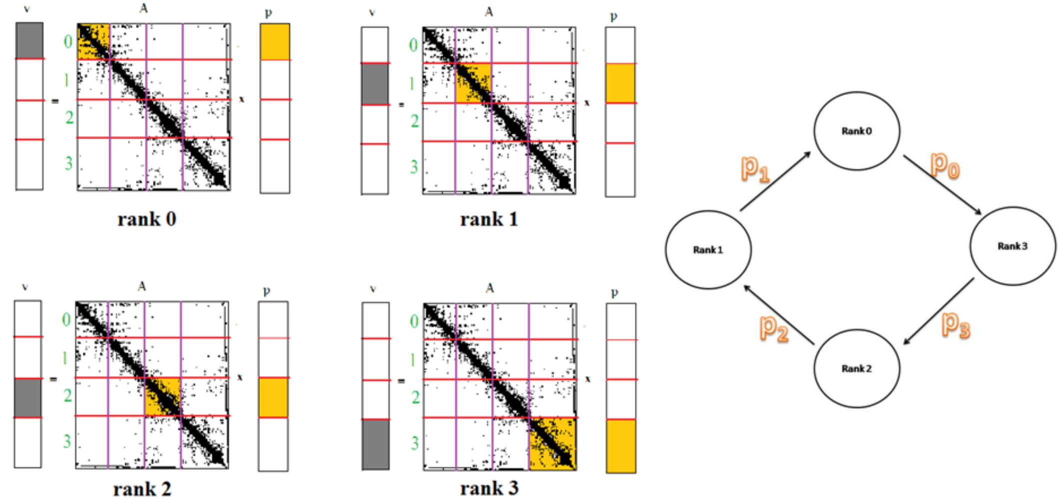


Figure 29: Step 1 MVM

Again the same process is followed but now the sender will send the values of ***p*** which it just received and the receiver will receive the values of ***p*** which will be useful for the MVM on next block, and the partial values of ***v*** found in this step are added to the values of ***v*** found in the previous step.

Figure 30 shows the progress of MVM and computation.

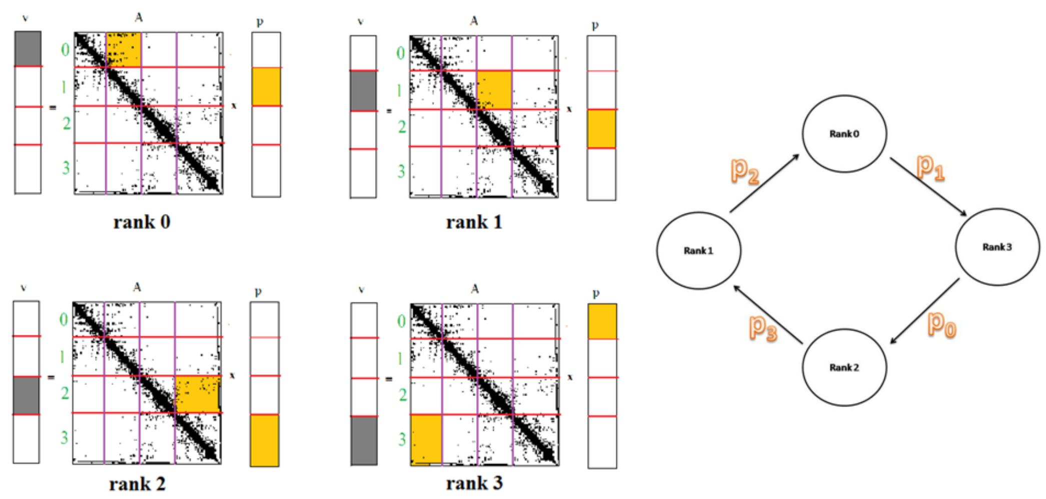


Figure 30: Step 2 MVM

Similarly,

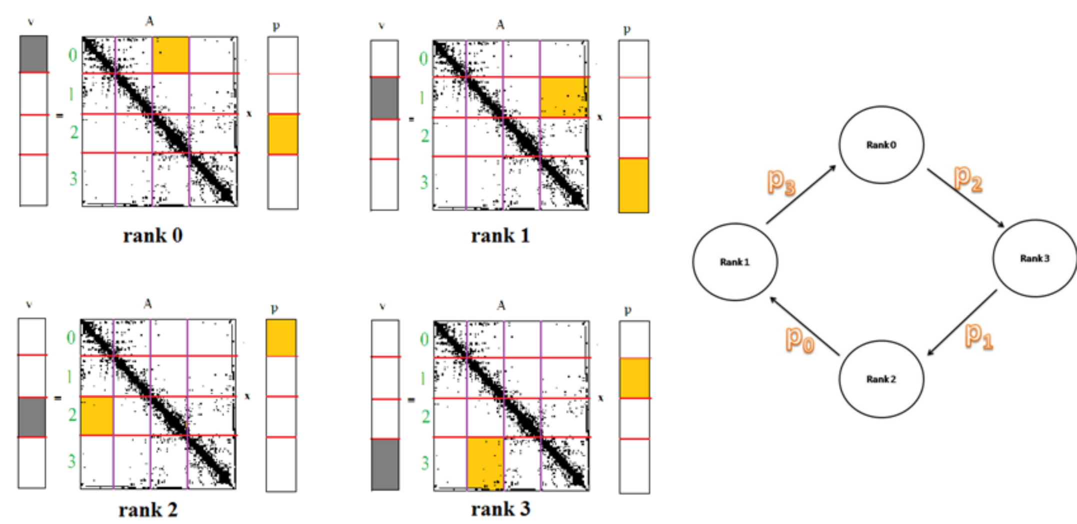


Figure 31: Step 3 MVM

Every process sends and receives ***number of processes -1*** chunks, so in the last step no further communication is needed every process has entire ***p.***

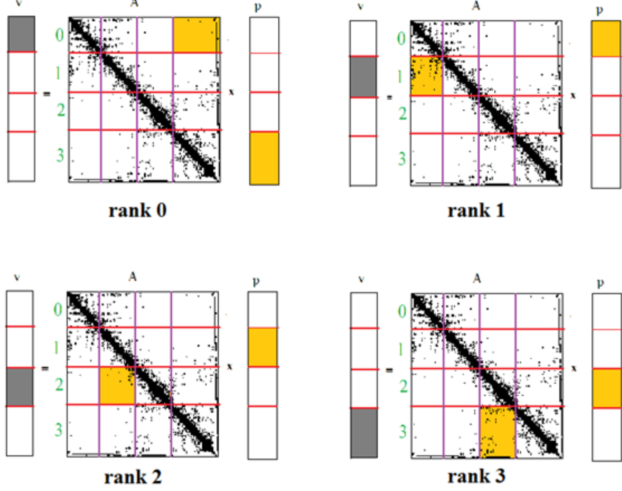


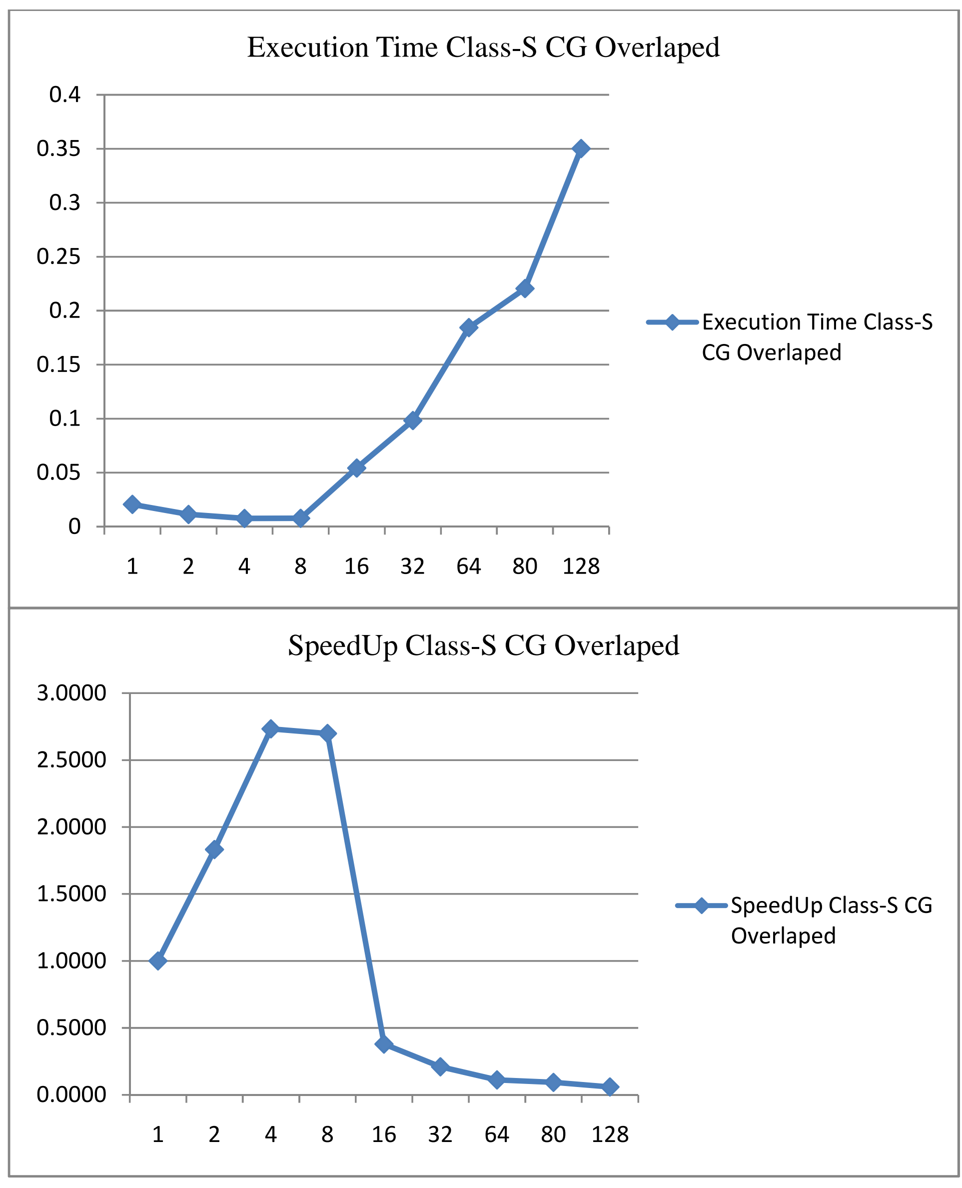
Figure 32: Step 4 MVM

After 4 steps (i.e. number of processes steps) the MVM ends, all the processes have the vector ***v*** found which will be required in the rest of CG.

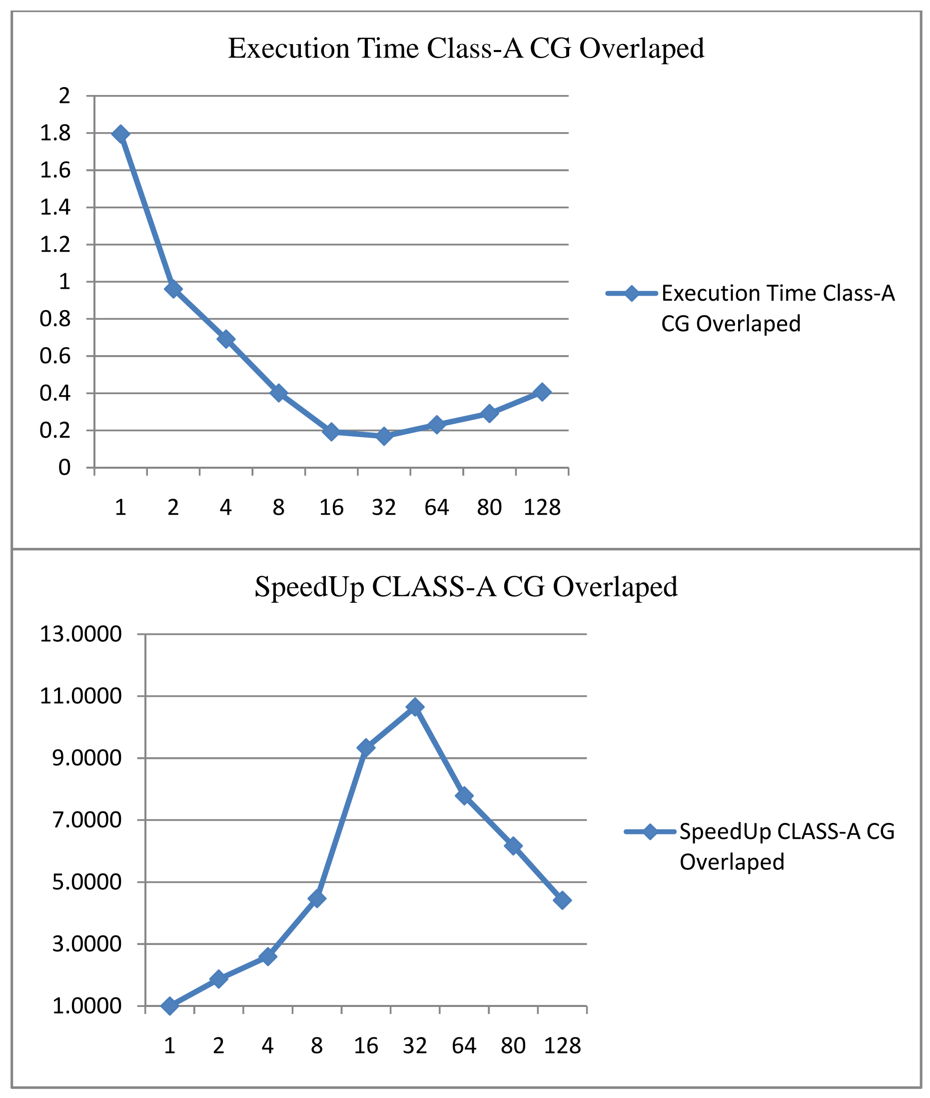
In this way we have avoided the gather and broadcast of ***p*** and overlapped communication and computation.

Benchmark results

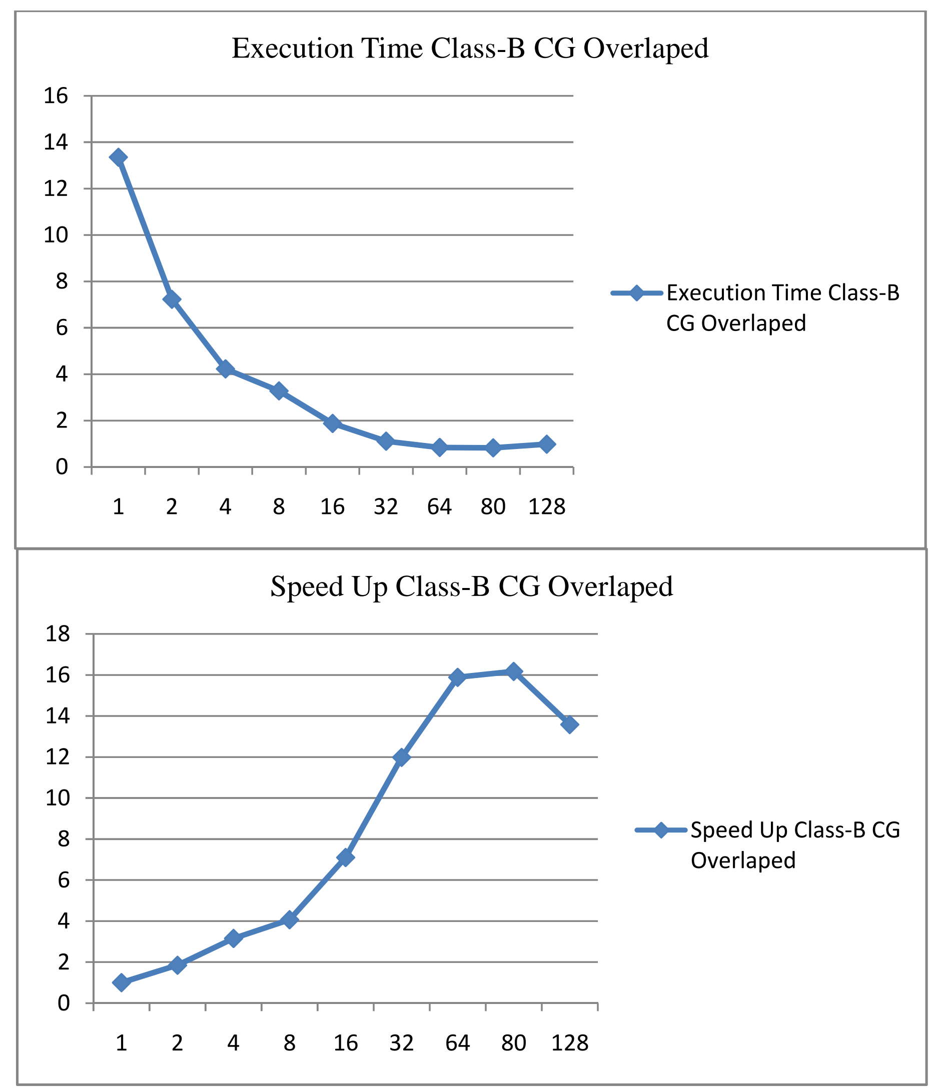
The benchmark for the solver was conducted using a standard data set from NAS Parallel Benchmarks (NPB) (22) on the supercomputing facility at UAEU, see appendix B1 for details.



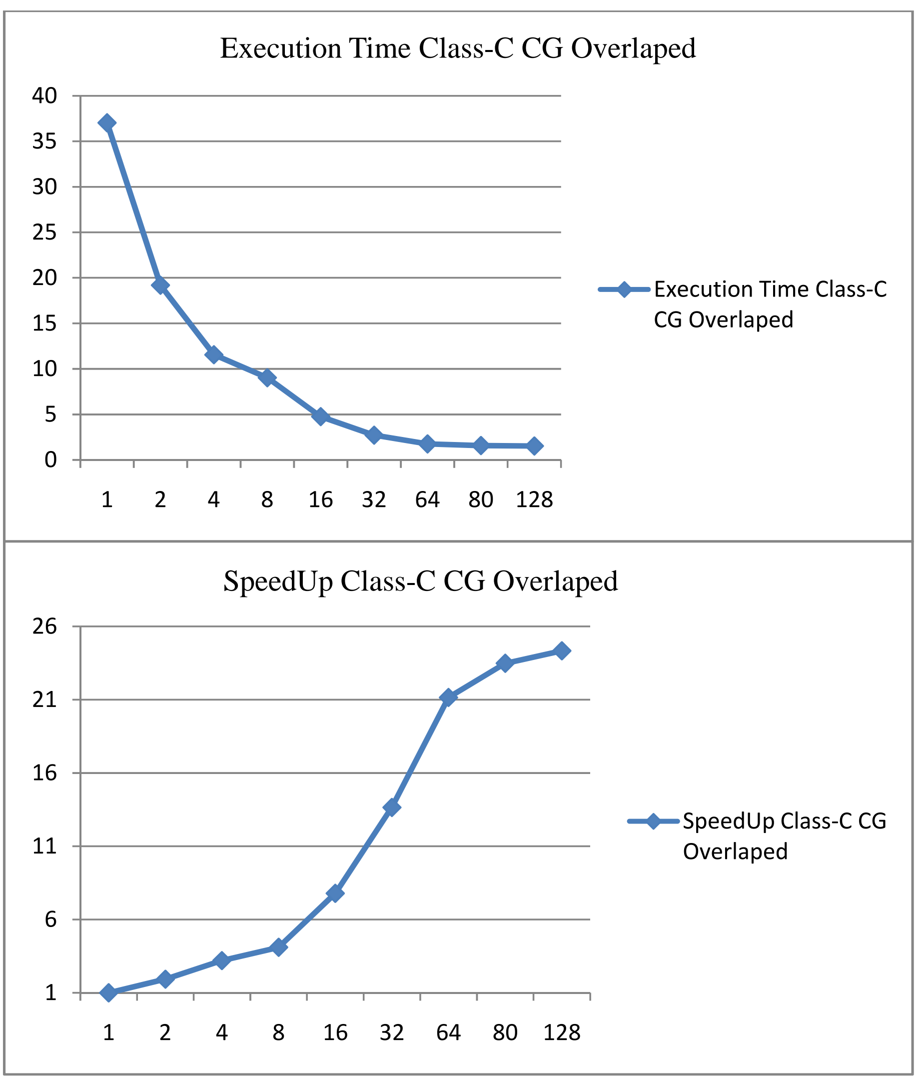
Dataset: CLASS-S, size=1400, iteration=15, NNZ = 78148



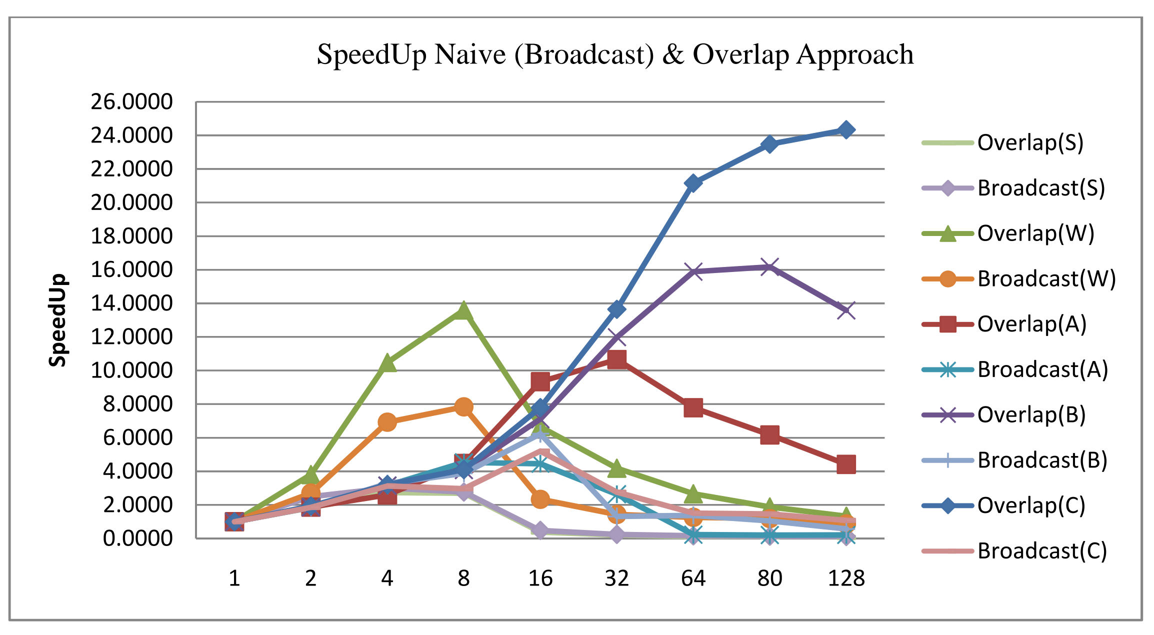
Dataset: CLASS-A, size=14000, iteration=75, NNZ = 1853104



Dataset: CLASS-B, size=75000, iteration=75, NNZ = 13708072

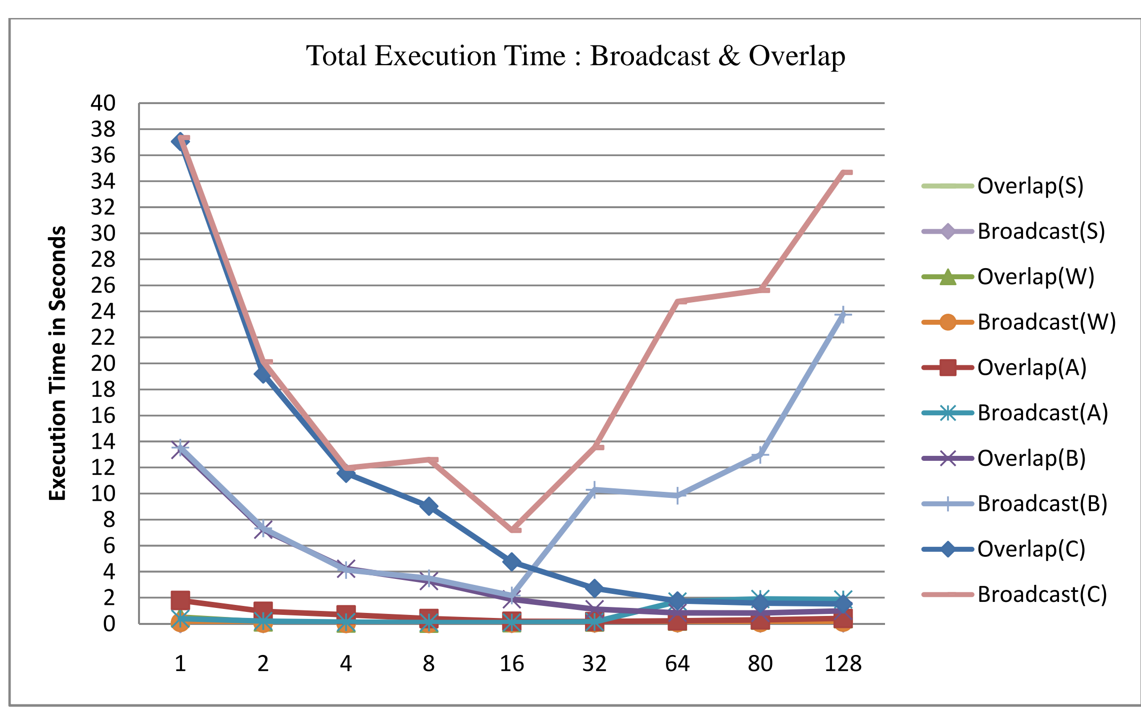


Dataset: CLASS-C, size=150000, iteration=75, NNZ = 36121058

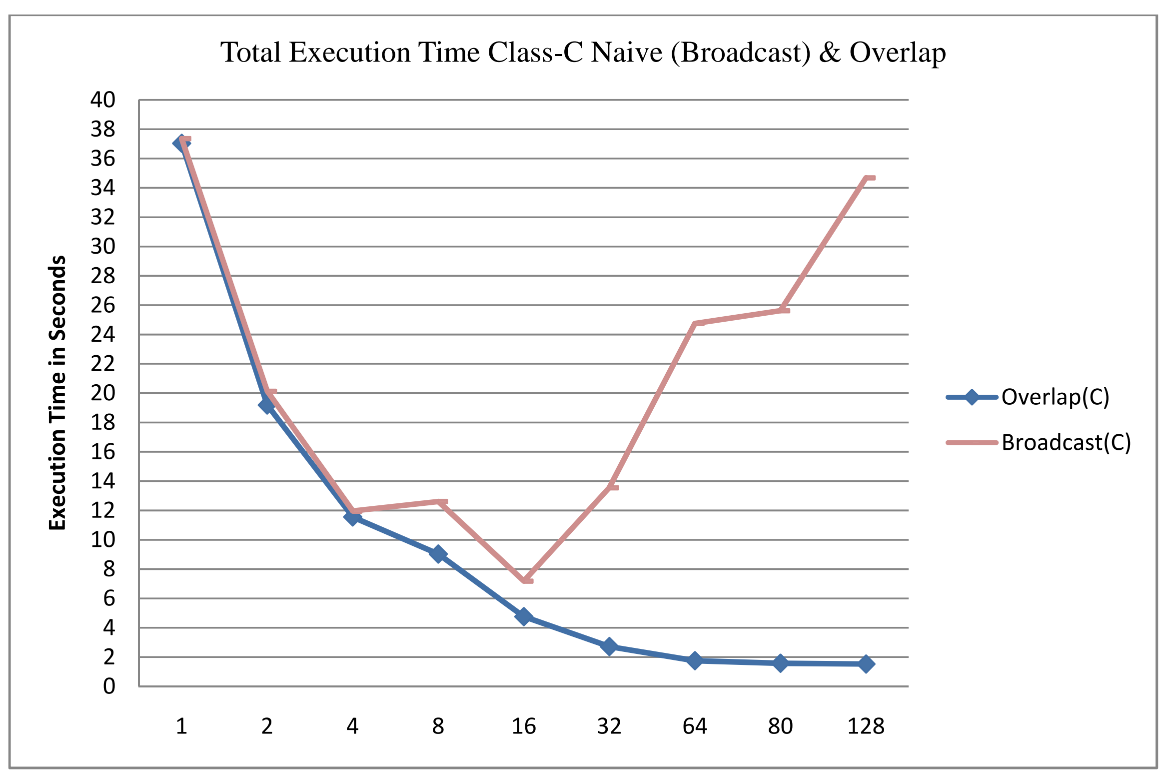


Speed up comparison

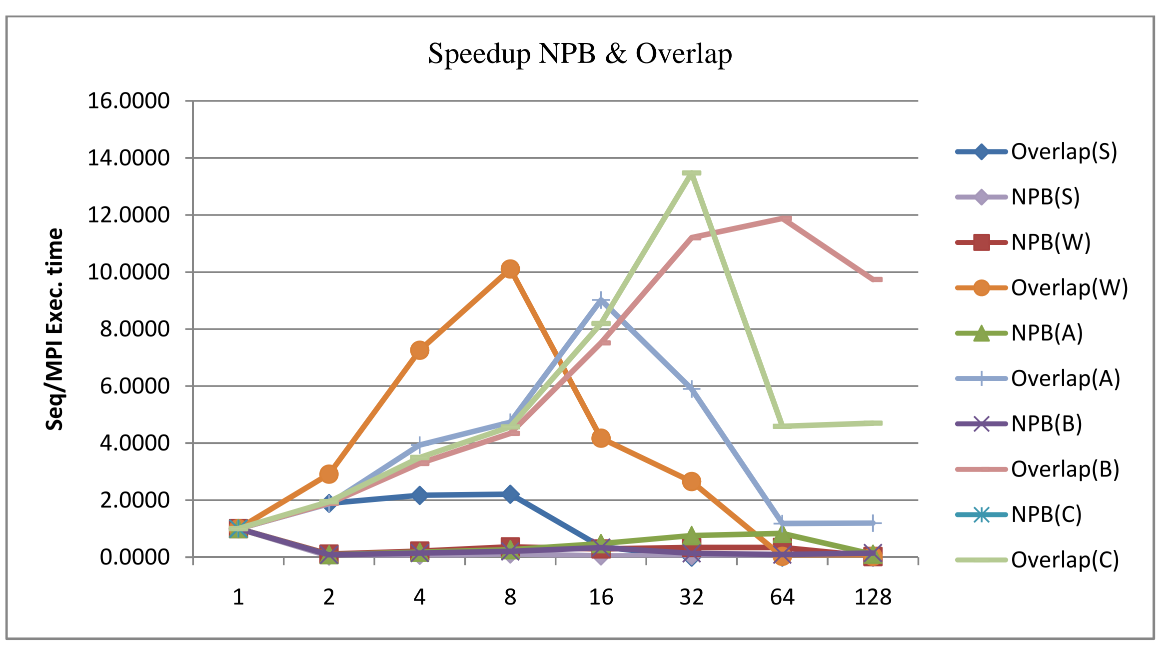
Note: CLASS-W, size=7000, iteration=15, NNZ = 508402



Total Execution Time : Broadcast & Overlap



Total Execution Time Class-C Naive (Broadcast) & Overlap



Speedup NPB & Overlap

Discussion on results

Improvements can be seen with the new approach, on larger datasets like Class-C & B good scalability was achieved but on smaller datasets like Class-S & A with increase in number of processes speedups drop because there is less work to do in MVM as compared to communication. So for real world applications where large reservoirs are modelled in which the matrices are in size of billions scalability can be achieved.

Oil Reservoir Simulation on Shared Memory Processors

The aim is to design and implement CG solver for multicore processors and SMP machines, where the main memory is shared among cores/processors, cache coherence can be a problem (24). The focus of optimization here is also on MVM.

There are two implementations in OpenMP based on the domain decomposition of source matrix ***A***, see figure 33 for demonstration suppose total number of threads are 4.

I. Row Partition
II. Block Partition

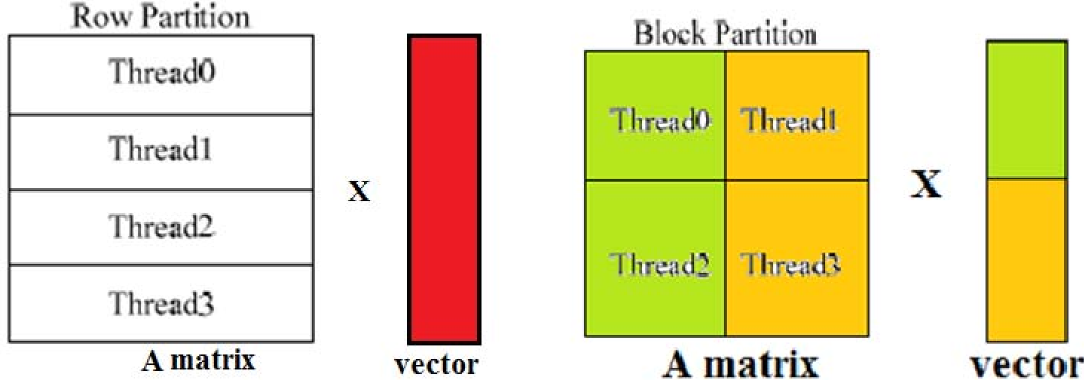


Figure 33: Row Partition [left] and Block Partition [right]

In both, the sparse matrix ***A*** is stored in CSR (Compressed Sparse Row) a widely used format for storing sparse systems (25). It only stores the nonzero elements with its column index, and the index of the first non-zero elements of each row, see figure 34.

For example, if matrix $A_1 = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 0 & 0 & 6 \end{pmatrix}$, where

m=4, n=4, nz=6, A_1 should be stored as follows.

$csrNz = \{1,2,3,4,5,6\};$

$csrCols = \{0,3,2,2,0,3\};$

$csrRowStart = \{0,2,3,4,6\}.$

Figure 34: Representation of matrix A in CSR format

From (26)

There are other sparse storage formats other than CSR which include CSC (Compressed Sparse Column) same as of CSR but row index are stored, BCSR (Blocking with Padding), BCSD (Blocked Compressed Sparse Diagonal), 1D-VBL (One-dimensional Variable Block Length), VBR (Variable Block Row) and more, refer to figure 35 for a graphical look of these different storage formats. Studies (27) suggest that the change of internal storage format as compared to CSR has little positive effect on the performance of Sparse MVM. CSR is also simple to use as does not require painstaking processes of finding optimum block size for submatrices and aligning the rows and columns afterwards.

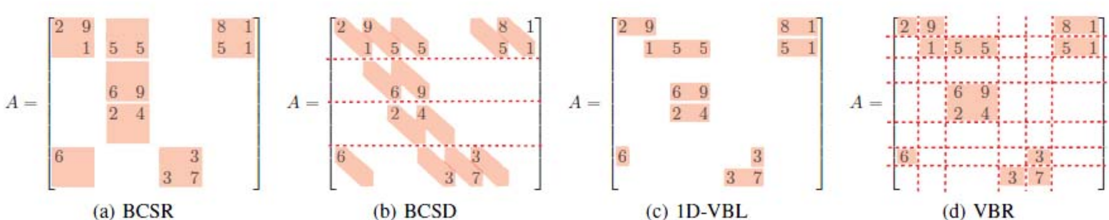


Figure 35: How the different blocking storage formats split the input matrix into blocks

From (27)

Row Partition

When the matrix ***A*** is partitioned into rows, each thread is assigned an iteration in which it gets one row at a time and multiplies it with the vector ***p*** and stores the result in vector ***v***, see figure 36 which shows how it is implemented.

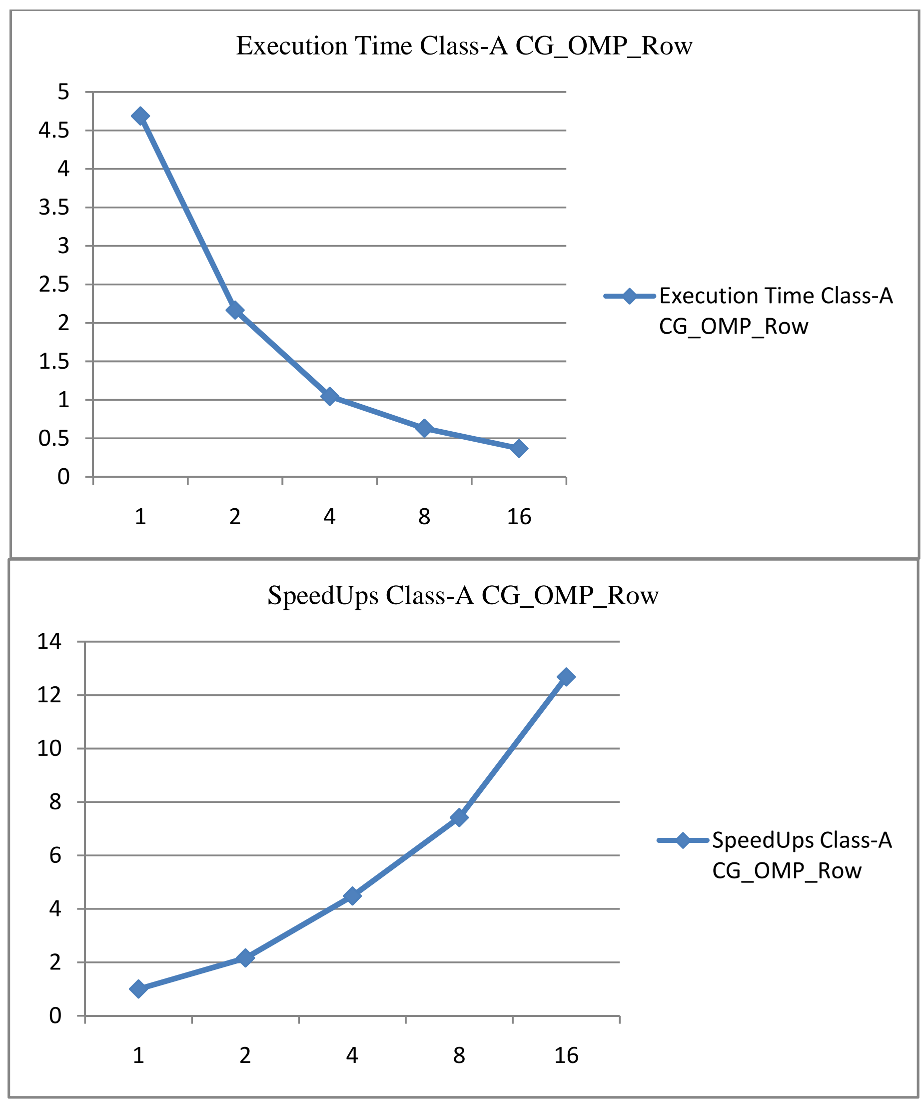
```
int i,j;

#pragma omp parallel for
for(i =0; i < N;i++){
    v[i] = 0.0;
}
#pragma omp parallel for private(j)
for(i = 0;i<N;i++){
    for(j = CRS_row[i]; j < CRS_row[i+1]; j++){
        result[i] += A[j] * vector[ CRS_Col[j] ];
   }
}
```

Figure 36: Simple Sparse MVM in CSR format

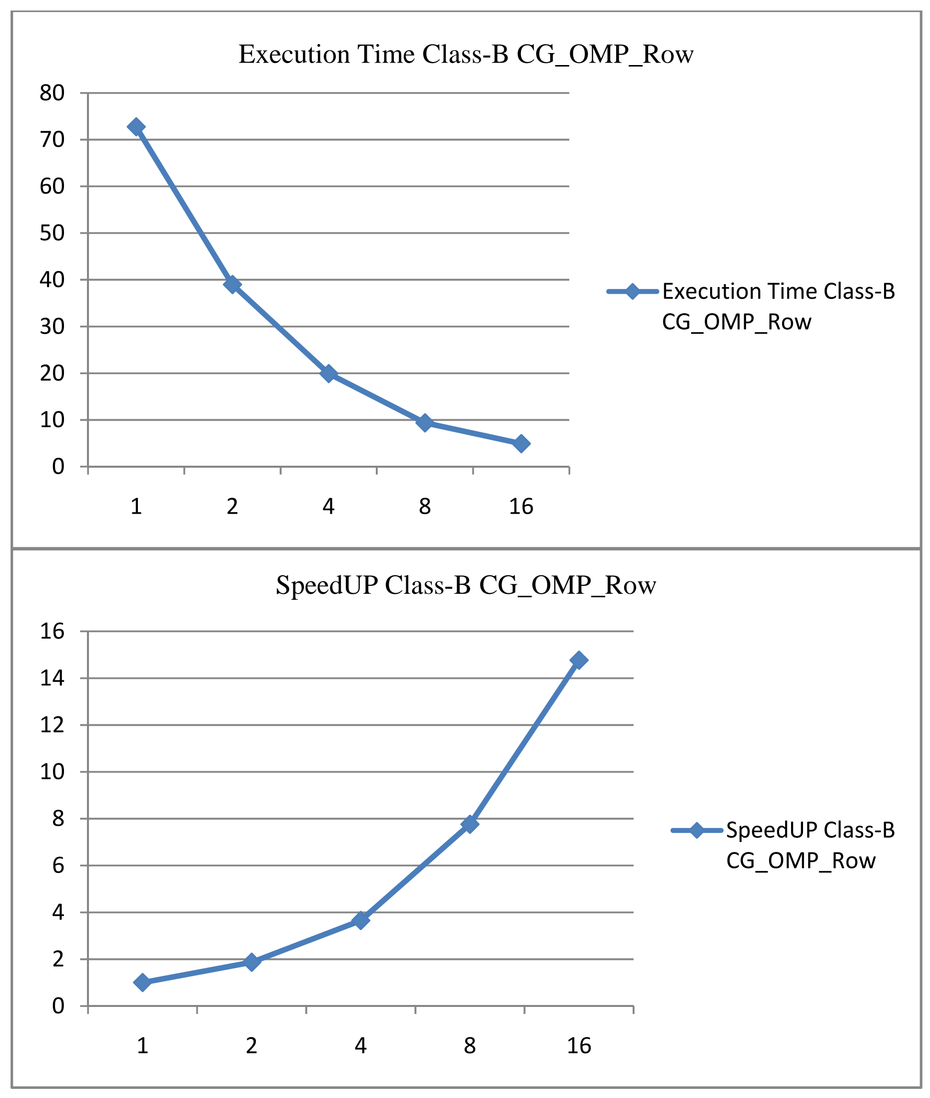
The problem with this approach is that the entire vector ***p*** is accessed in an irregular fashion, therefore when vector ***p*** is larger more cache misses are anticipated.

Benchmark results

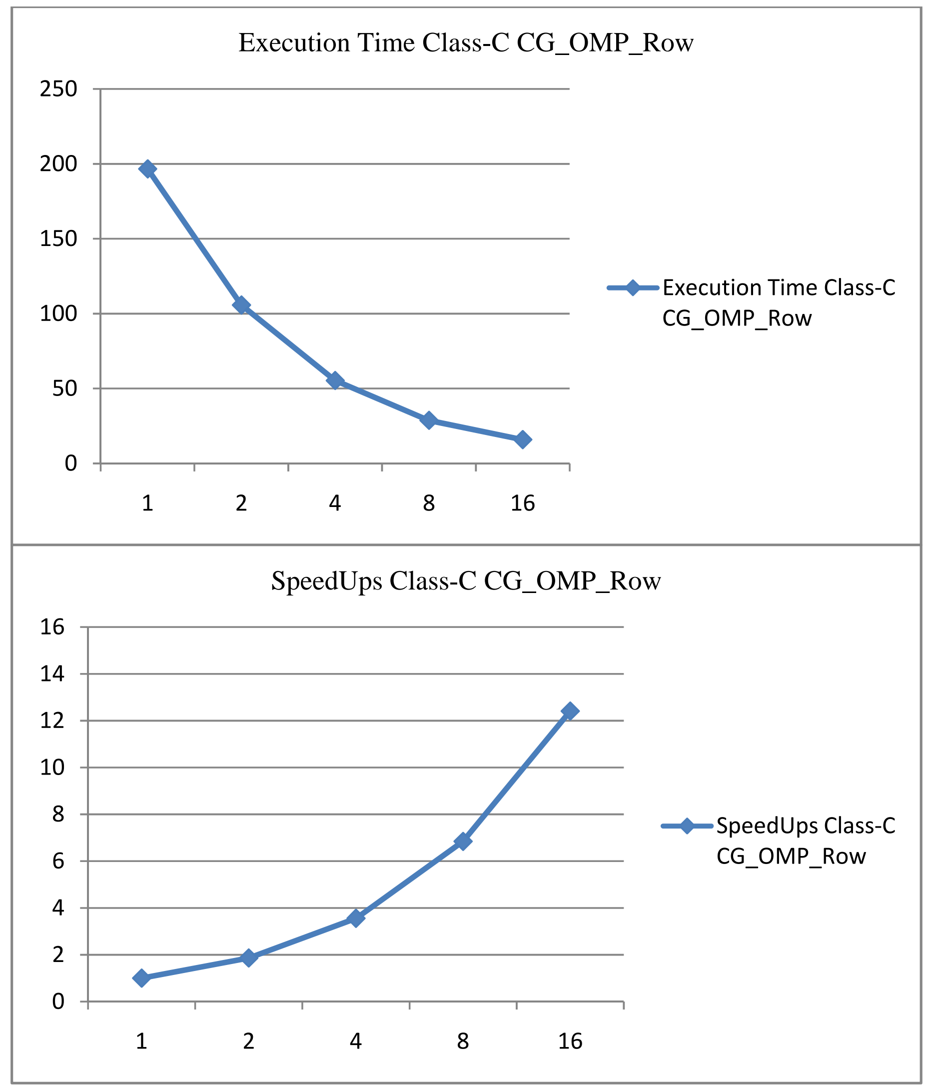
The benchmark was conducted on Raad SMP machine at NUST-SEECS, for details of the machine see appendix B3. Dataset used came from NAS Parallel Benchmarks (NPB) (22).



Dataset: CLASS-A, size=14000, iteration=75, NNZ = 1853104



Dataset: CLASS-B, size=75000, iteration=75, NNZ = 13708072



Dataset: CLASS-C, size=150000, iteration=75, NNZ = 36121058

Discussion on results

Scalability was achieved in the tests but as anticipated when the large system is simulated the speed-ups are relatively less. Because the vector ***p*** gets larger the irregular way of accessing ***p*** results in many caches misses.

Bock Partition

This time the matrix ***A*** is partitioned into blocks. From the figure 37 it is easy to understand that in sparse MVM thread no. 0 & 2 need first portion of the vector and thread no. 1 & 3 need the other, in this way we have successfully reduced the access space of the vector. As the access space for the vector is decreased we anticipate decrease in cache misses.

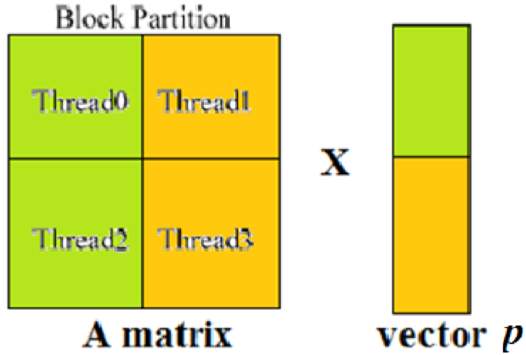
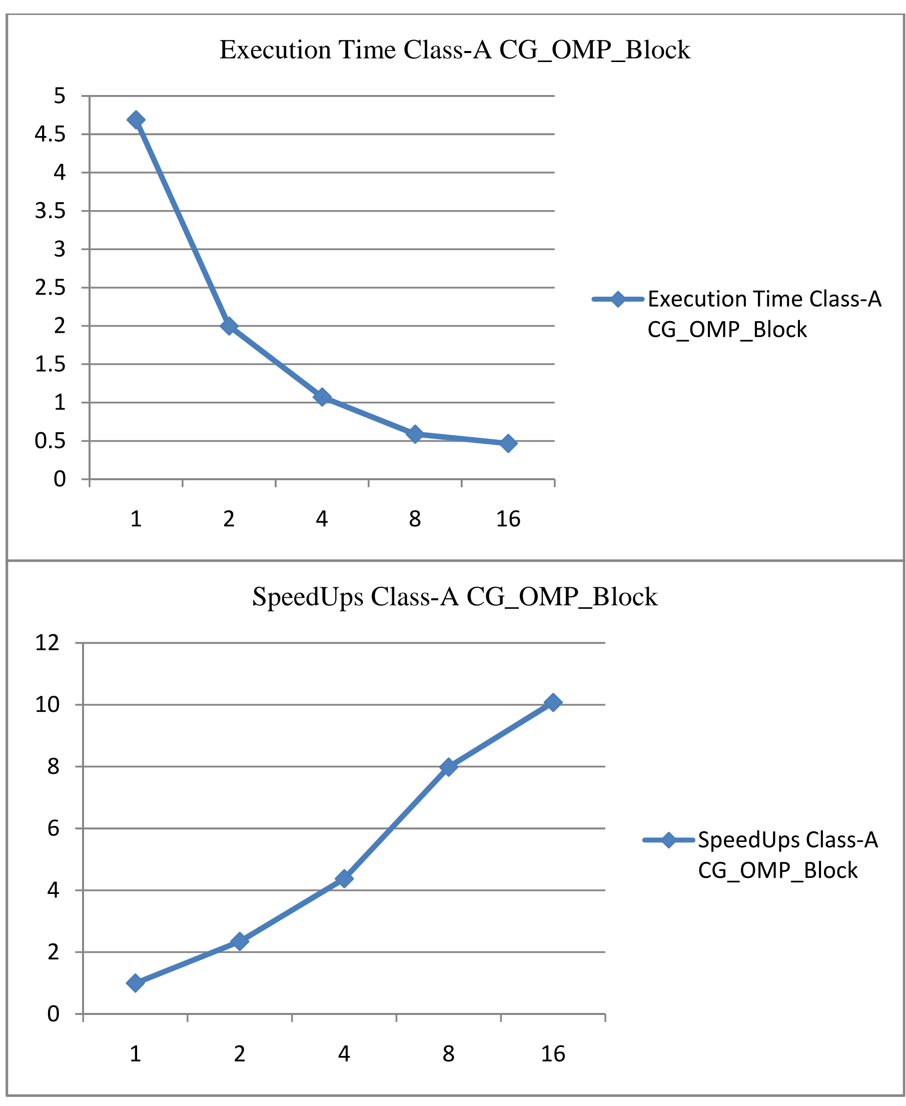


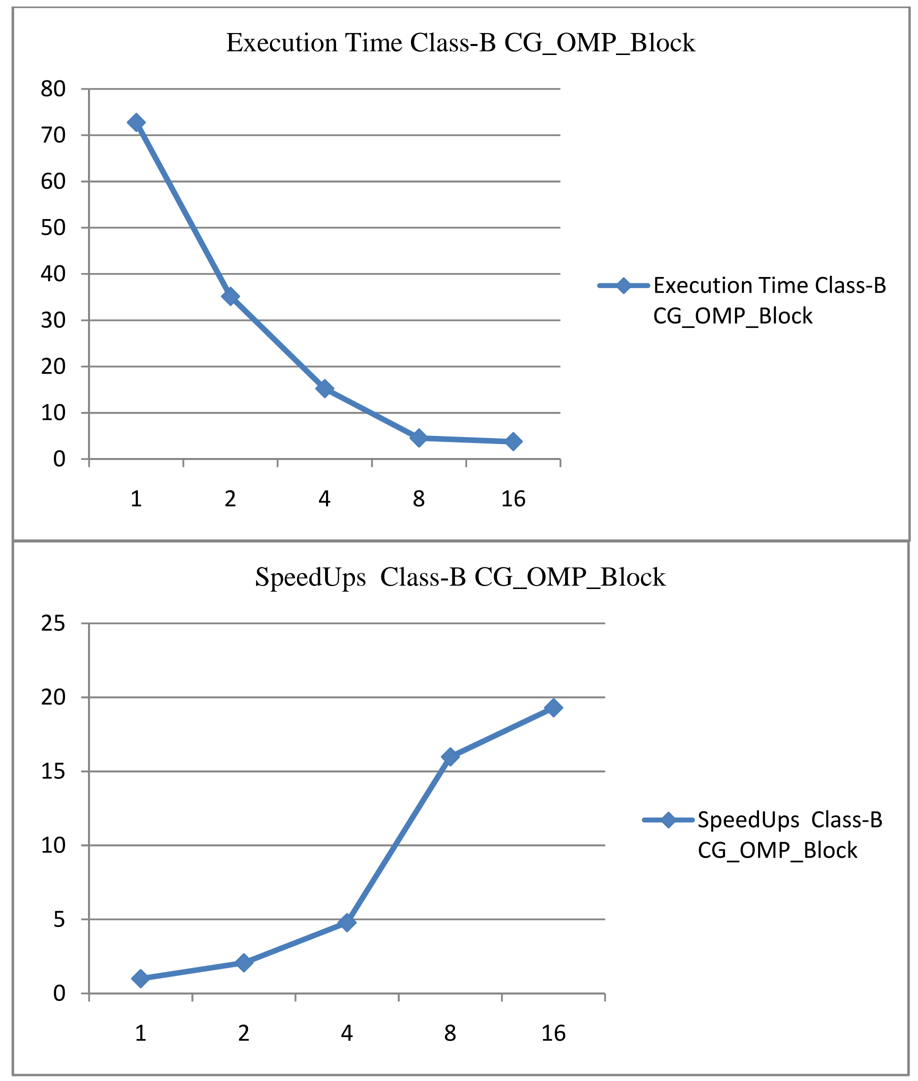
Figure 37: MVM with Block Partitioning of matrix A

Benchmark results

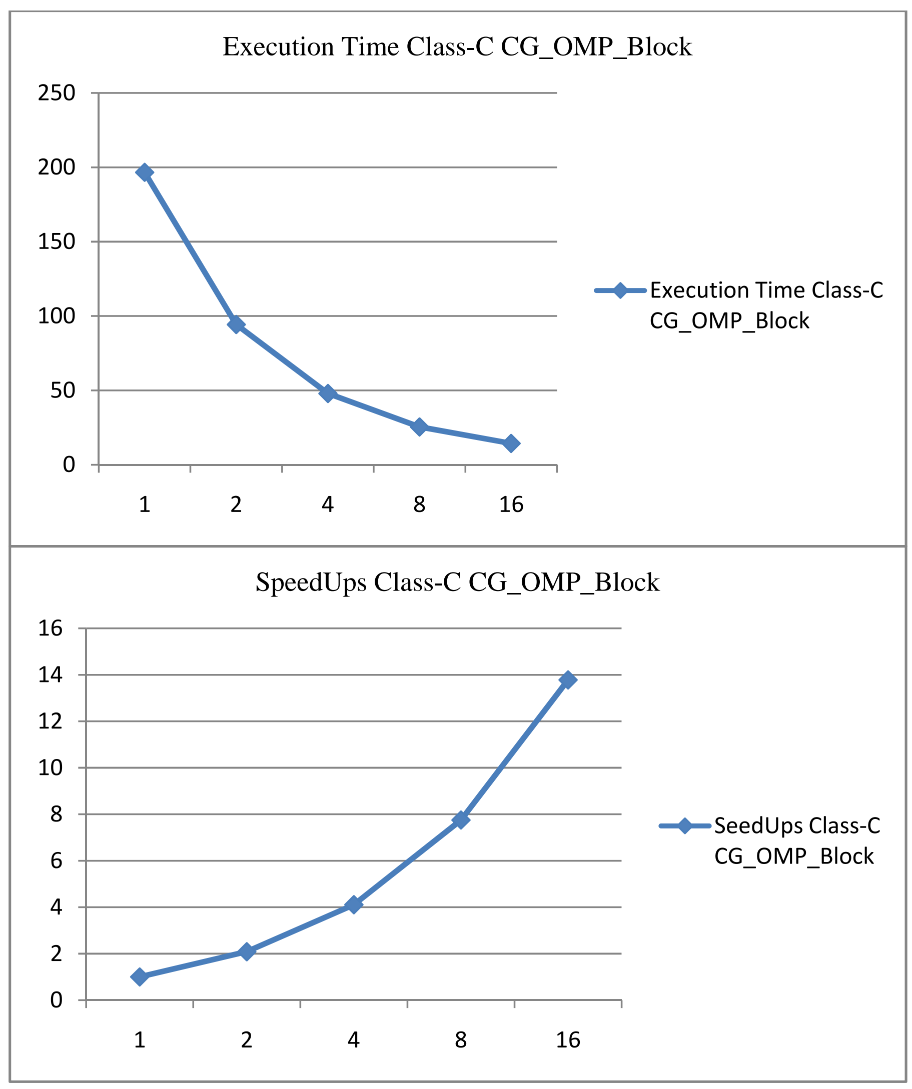
The benchmark was conducted on Raad SMP machine at NUST-SEECS, for details of the machine see appendix B3. Dataset used came from NAS Parallel Benchmarks (NPB) (22).



Dataset: CLASS-A, size=14000, iteration=75, NNZ = 1853104



Dataset: CLASS-B, size=75000, iteration=75, NNZ = 13708072



Dataset: CLASS-C, size=150000, iteration=75, NNZ = 36121058

Discussion on results

With block partitioning better speedups were obtained with dataset Class-C which is the largest. With Class-B super linear speedups were witnessed. On the smallest dataset Class-A performance was affected a little bit because the overhead of accumulation of results from threads after MVM which was done locally.

Oil Reservoir Simulation on GPUs

The CG solver was implemented in CUDA for nvidia GPUs. In the GPU there is limited memory and the access to global memory is expensive, if the sparse matrix is stored in CSR format it will take more space and more access as compared to DIA (Diagonal Sparse Matrix) format. The matrices as discussed in chapter 2 the matrices from Oil Reservoir Simulation have a structure, for 1D simulation for matrix is tridiagonal, for 2D its pentadiagonal and for 3D its heptadiagonal. This symmetry can be exploited by only storing the nonzero values of matrix in an array and in the offset between diagonals in a separate array. See Figure 38.

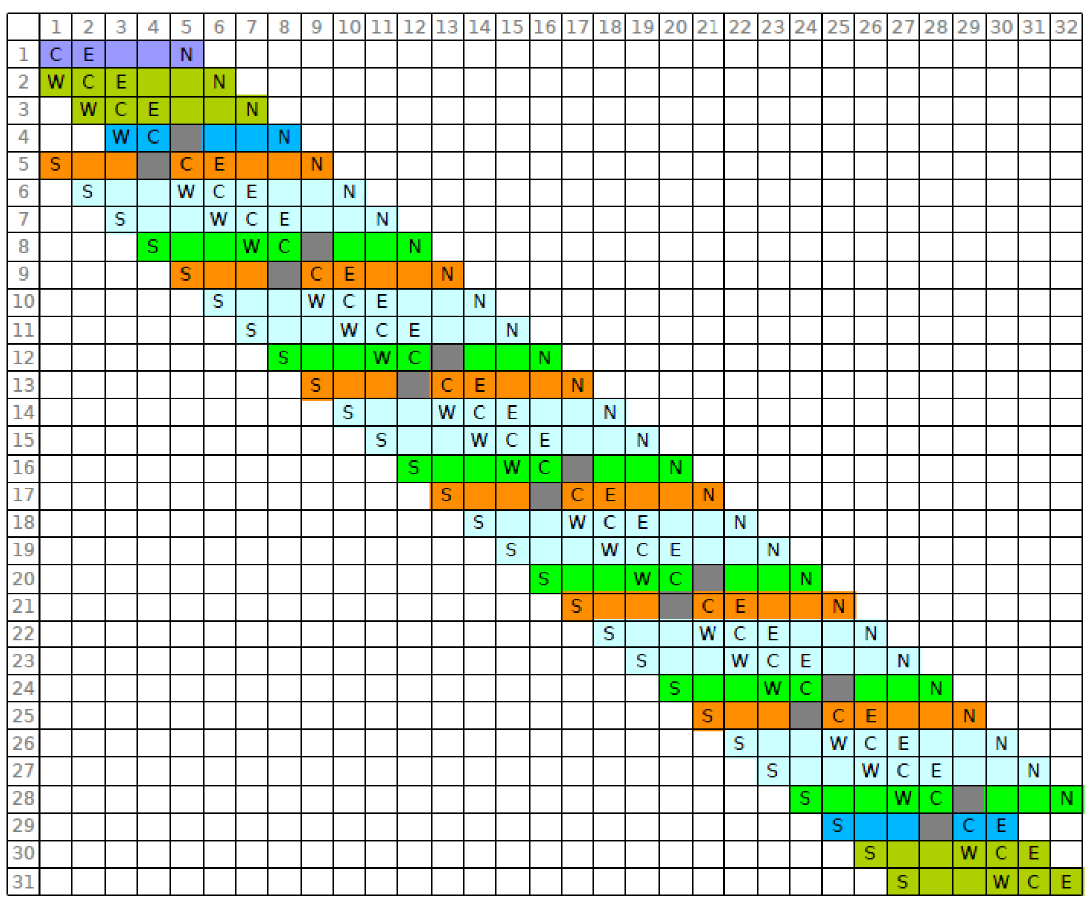
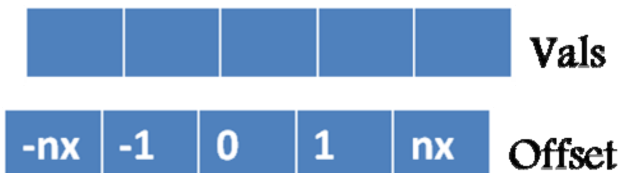


Figure 38: 32x32 Matrix representing a 2D reservoir in 1 Phase. Here offset from central diagonal is nx = 4

DIA format representation



CUDA versions of CG

In the CUDA version the host thread (CPU) handles the main CG iteration and calls compute kernels multiple times. Computer kernels include MVM kernel, VVM and Reduction kernel, VV addition kernel.

Bellow is explaination of different kernels.

MVM kernel

Figure 39 show the code snippet of the MVM kernel, each thread is assigned a unique row which it multiplies with the vector. The column and row indices are calculated at runtime and the values are only fetched.

```
int row = blockDim.x * blockIdx.x + threadIdx.x ;
if(row < num_rows ){
    double dot = 0;

    for ( int n = 0; n < num_diags ; n ++){

        int col = row + offsets [n];
        double val = data [ row * num_diags + n ];
        if( col >= 0 && col < num_cols )
            dot += val * vector[ col ];

    }
   resultant_vector[ row ] = dot;
}
```

Figure 39: CUDA MVM kernel

VVM and Reduction kernel

As we know that the result of VVM is a scalar, that means threads in different thread blocks not only have to perform simple multiplication but also a complex synchronous reduction process. This reduction process is divided into two phases. In the first phase inter block reduction is performed in which threads in a block participate to reduce the portion of result which is stored in shared memory of that block, see figure 40 and figure 41.

```
__shared__ double sdata[512];
// each thread loads one element from both global arrays to share
// applying multiplication

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i < N){
     sdata[tid] = vec1[i] * vec2[i];

     __syncthreads();


     for (unsigned int s=blockDim.x/2; s>0; s>>=1) {

          if (tid < s) {
             sdata[tid] += sdata[tid + s];
          }
          __syncthreads();
     }

}
```

VVM

Intra Block reduction

Figure 40: VVM and Intra Block reduction

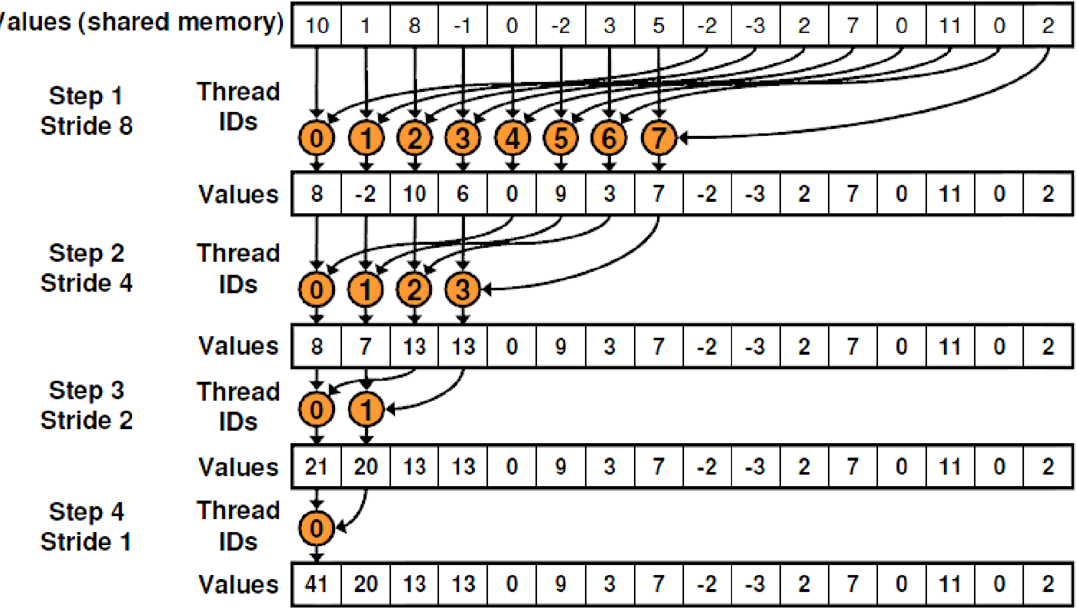
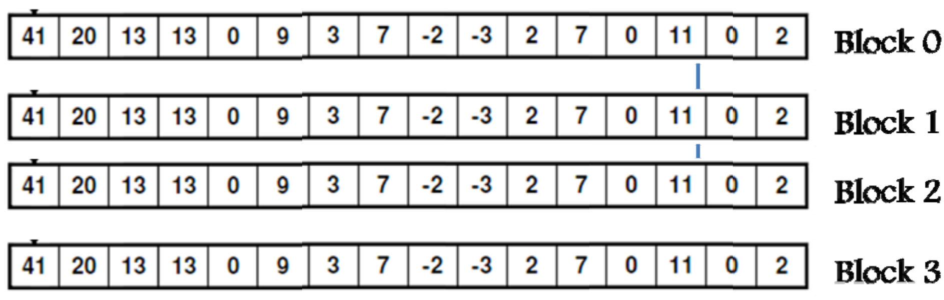


Figure 41: Graphical view of intra block reduction.

At the end 0th thread of the block writes the result into global memory this is from where second phase of the reduction starts. When all blocks are finished, insuring synchronous flow the block which writes in last performs the reduction globally. This global reduction is further illustrated in figure 42.

Global Reduction



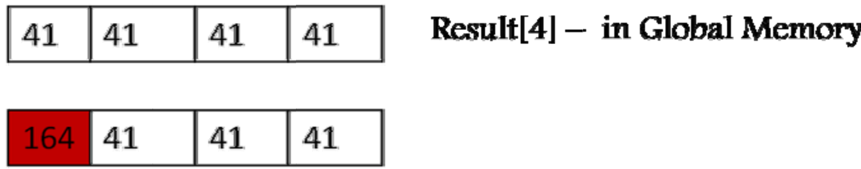


Figure 42: Global reduction, among 4 thread blocks

Note: normally all blocks do not have same value this is for demonstration purpose only.

VV Addition kernel

From figure 43 it can be seen that VV addition is a simple piece of code in which every thread is fetches the values from two vectors (vec1 and vec2) on the index with corresponds to its global index in the pool of threads. The values are added and the result is stored in the first vector (vec1). Note the scalar multiplication with vec2 this scalar can be alpha or beta in CG as it is called from the CPU in the naive approach.

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i < N_){

    vec1_[i] += vec1_[i] + scalar_ * vec2_[i];

}
```

Figure 43: VV Addition

Benchmark results

The benchmark was conducted on GPU test best at NUST-SEECS, see appendix B4 for details of the machine. An Oil Reservoir with 3072 x 4096 grid dimension (2D) was simulated with 5 time steps. Matrix dimension is 12582912 x 12582912, nnz = 62914560, nx = 3072, ny = 4096.

Benchmark included Total Time, Computation Time, Memory Copy Time, and Kernel Time; here the time is in seconds.

Total Time (sec)	Computation Time (sec)	Memory Time (sec)
2.830271000000000203	1.880077999999999916	0.170836999999999889

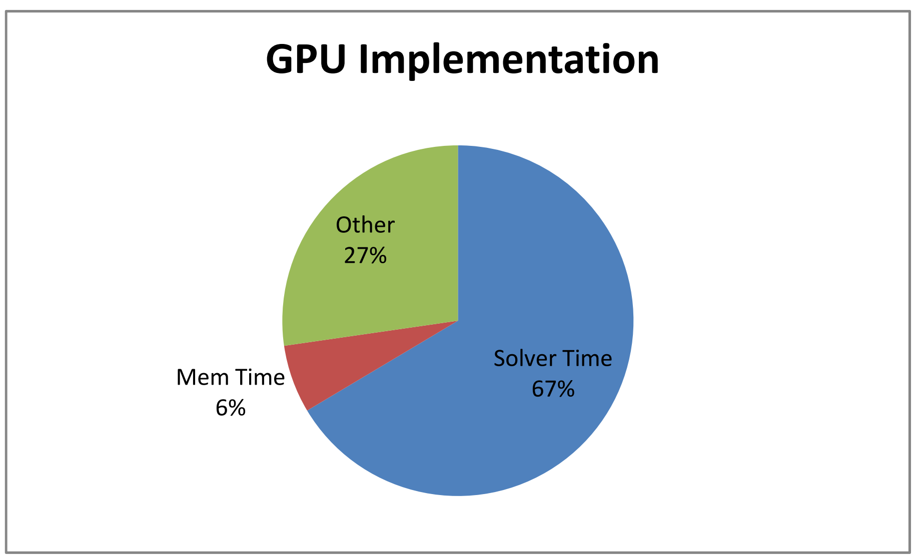


Figure 44: Percentage of time spent on different operations

Discussion on results

We achieved 12x speed-ups for the entire application, which included formation of PDEs to system of linear equations and their solution for multiple time steps. All operations were performed in double precision.

Chapter 6

CASE STUDY III: BLACKSCHOLES

This chapter includes all the relevant details of the design and analysis of the system, which comprises of main modules that will be implemented and optimized. We will also discuss the significance of the approach followed in parallelizing.

BLACKSCHOLES

The Blackscholes formula and its technique to calculate option prices for portfolio of options describes that Blackscholes is the simplest in nature. Its domain decomposition is simple due to the non-dependency of the option data which means that only option data is sufficient to calculate price for that particular option. So, inter-processors communication overhead is almost zero.

Blackscholes On Distribute Memory Clusters

We will be discussing Blackscholes parallelization on Distribute Memory Machines. We will also be evaluating these techniques later in this chapter.

Domain Decomposition

This is the most important and foremost step in any parallel application. The input data, which in our case is Option Data, is read from input file. This data has to be parallelized in such a way that the work distribution on all the processors should be balanced. The domain decomposition phase is very dynamic task and it strongly depends on the cluster's architecture underneath.

Design

Due to simplest data parallel nature of Blackscholes, we use simple row based partition approach. In this approach, we divide the total number of options by total number of processors to get a subset of options data set.

These chunks are then equally distributed among the processors by a collective call. Then each MPI process computes option prices for its own chunk of options. When all the processors are done with computation, they send back results and results are saved.

Basic Flow of Blackscholes in distribute memory is shown below

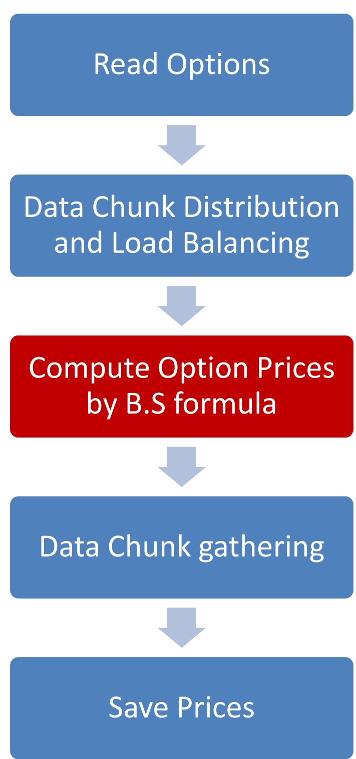


Figure 45: Basic Flow of Blackscholes in distribute memory

In this flow diagram of Blackscholes, the compute intensive part is highlighted by red colour. We are actually parallelizing this part.

Implementation

As discussed in design of Blackscholes that the analysis of the code shows that the option price calculation is more time consuming, so we need to parallelize this function.

A code snippet of serial version of the compute intensive function is given below:

```
calculateOptionPrice (… , N , …) {

    for(i 1:N)
          {
            /* Compute the Formula and get OptionPrice.... */
            /* check whether the 'Optiontype' is either 'Put' or
'call' */
            int option = (data[i].OptionType == 'P') ? 1 : 0;

            /* Compute Formula */
            optionPrices [i] = computeFormula (data[i].s,
                                                data[i].strike,
                                                data[i].r,
                                                data[i].v,
                                                data[i].t,
                                                option);
          }

}
```

The *computeFormula* function does all the computation. In serial approach, the main loop which iterates over N options. In MPI based parallel approach we divide this main for loop, and get smaller chunks of options data.

$$Options_chunk = total_num_options \,/\, total_num_processors$$

Then each process will be iterating over *Options_chunk* instead of N options.

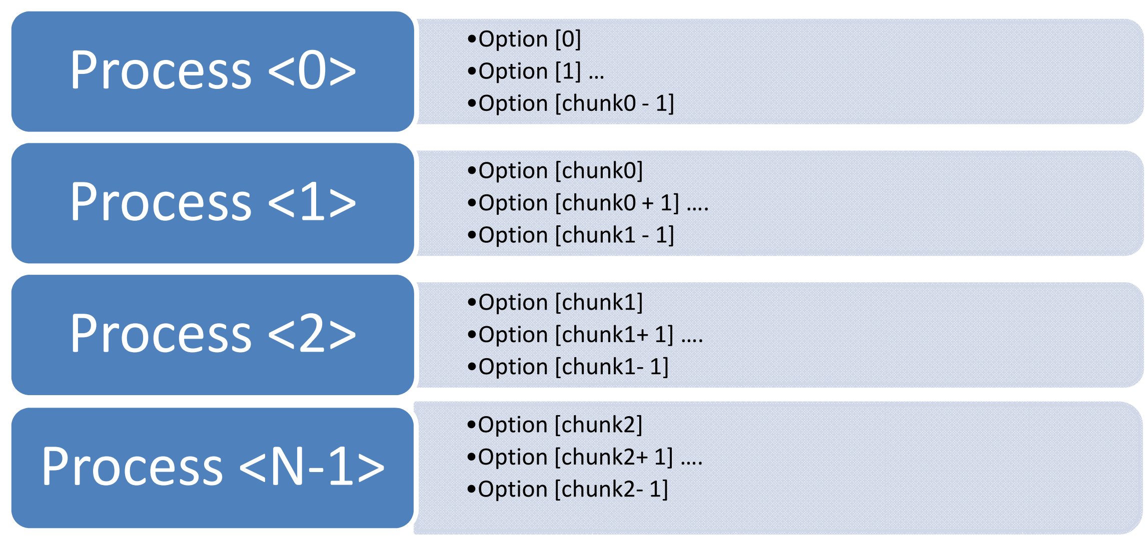
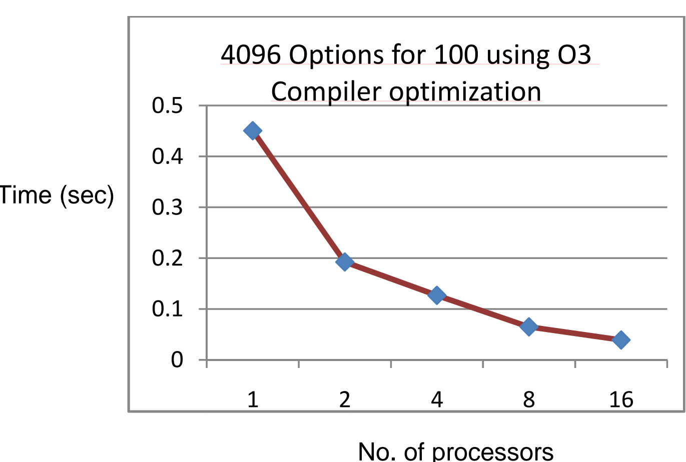


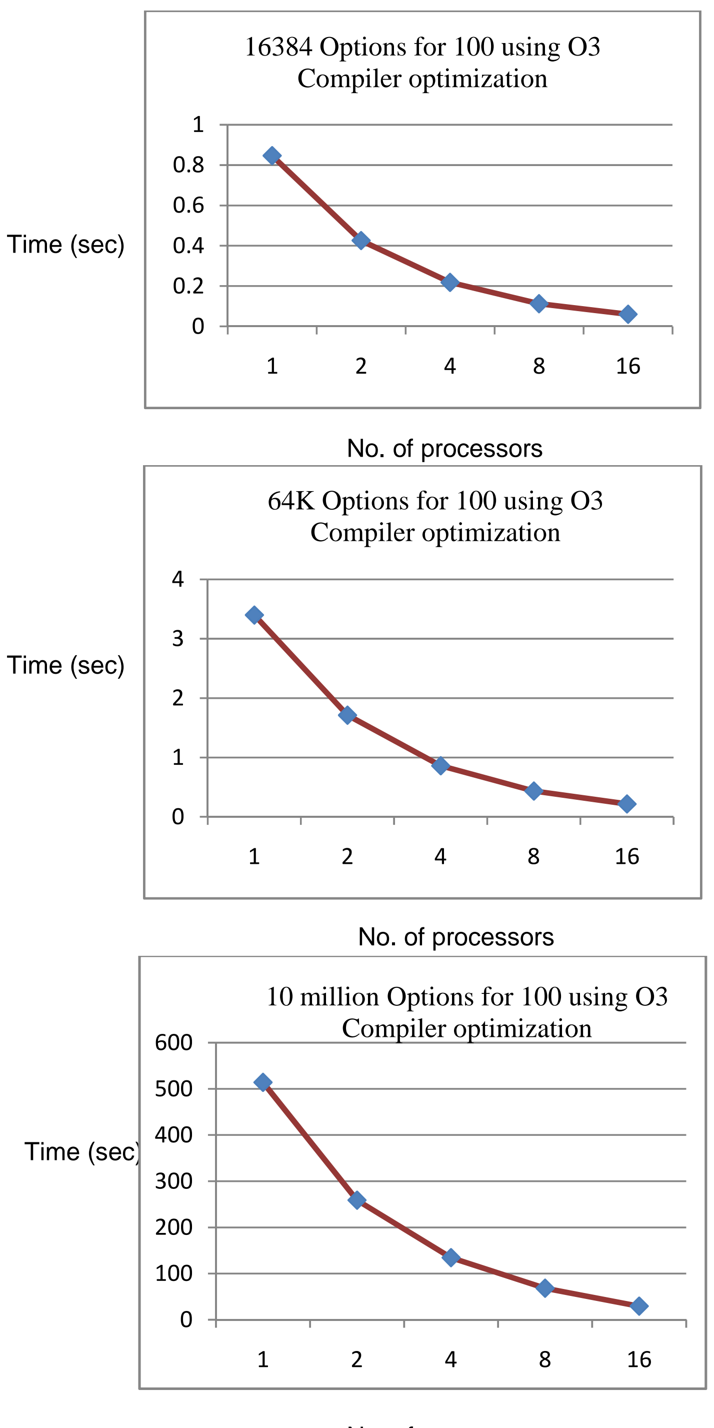
Figure 46: MPI based parallelization approach to Blackscholes

Benchmark results

The benchmarks were conducted on Barq cluster at NUST-SEECS, see appendix B2 for details of the machine.

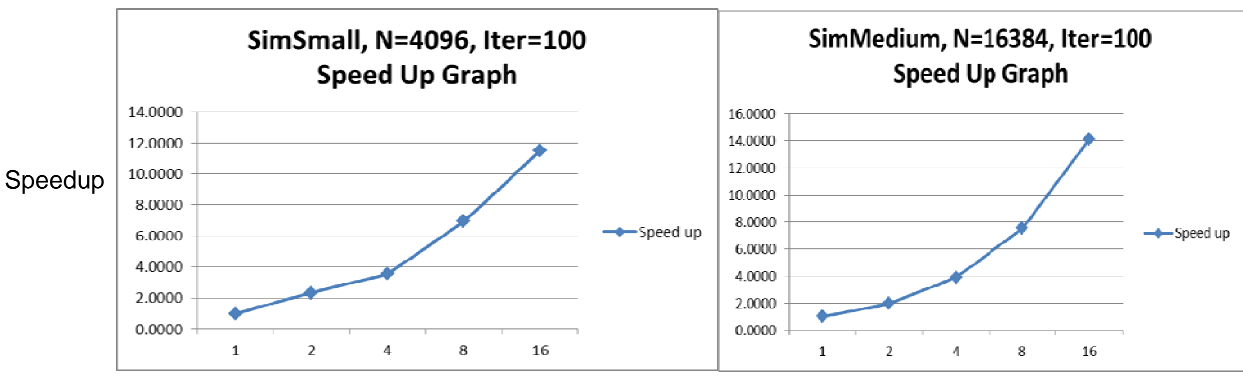
As discussed earlier that the Blackscholes application is data parallel in nature with almost zero communication cost. So, MPI parallelization approach with balanced partitioning of options can lead to good speedups. The data sets for Blackscholes were varying in nature. The maximum data set was 10 million options. When the experiments were performed, we obtained these results.

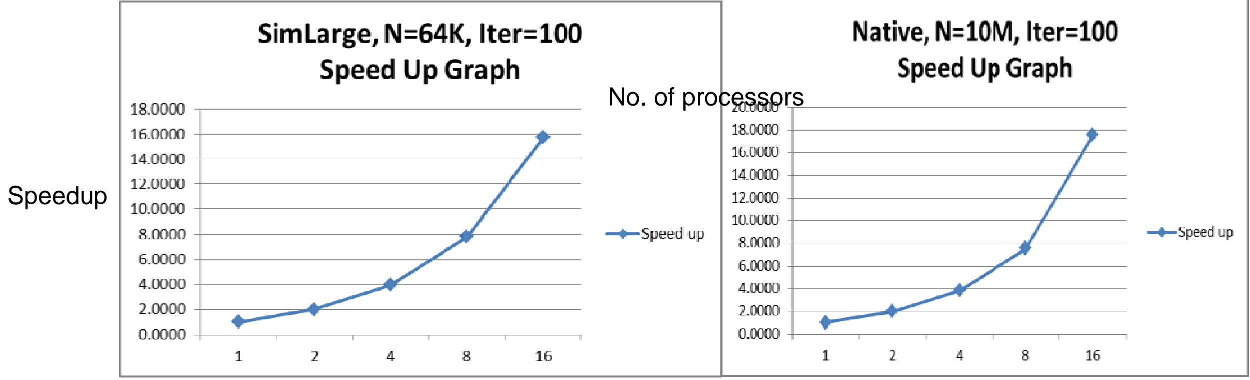




Blackscholes MPI Timing Results on Varying Data Sets

In these graphs, we see that the execution time has decreased as we increased number of processors. Now, we will see the speedups achieved by the parallelization.





No. of processors

Blackscholes MPI Speedup Results on Varying Data Sets

As we see in the above graphs, the speedups are shown. On maximum dataset at 16 processors, we see that the speedup is about 18. The reason is that in parallel version, cache locality has been improved.

Blackscholes On GPUs

GPU based approach is somewhat more complicated than MPI approach in which we simply divide the options. In CUDA based approach, we have to take much care of the efficient memory access and make sure that SIMD (Single Instruction Multiple Data) operations are being performed in order to best utilize CUDA threads.

Basic Flow of Blackscholes for GPU parallelization approach is shown below:

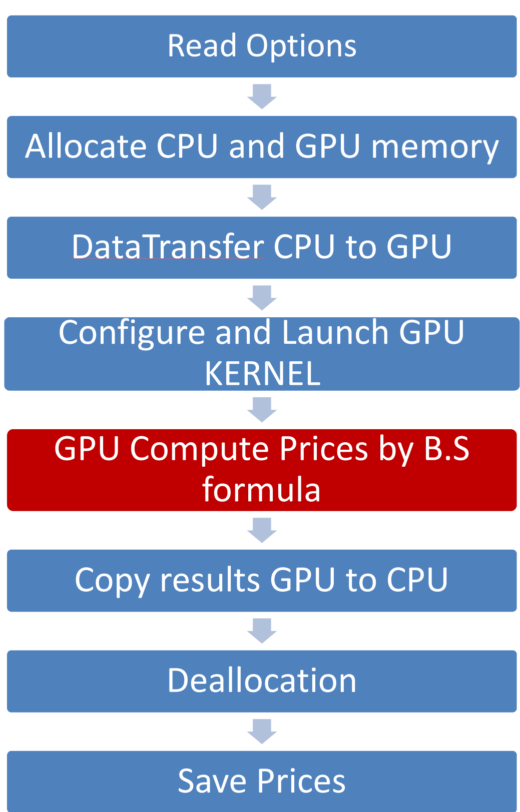


Figure 47: GPU based parallelization approach to Blackscholes

In this flow diagram of Blackscholes, the compute intensive part is highlighted by red colour. We are actually parallelizing this part.

Blackscholes CUDA Implementations

As discussed in design of Blackscholes that the analysis of the code shows that the option price calculation is more time consuming, so we need to parallelize this function.

In cuda implementation, we proposed two implementation which we will call as; *naïve* and *optimized* parallelization approach. In *naïve* approach, the data is in the form of a data structure named *OptionData*. We simply launch CUDA kernel, which will spawn cuda threads, and each thread will be responsible to calculate one option.

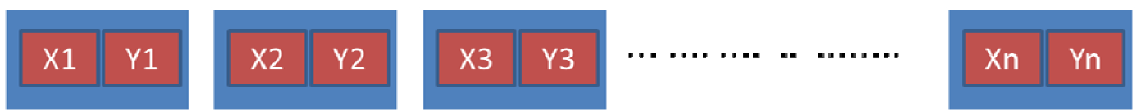


Figure 48: Options in the form of Option Data Structure (Naïve approach) assuming x, y option variables

A code snippet of above mentioned approach is shown below:

```
__global__ void BlackScholesGPU (float *d_s, float *d_strike,
            float *d_r, float *d_v, float *d_t, char*d_OptionType,
            float *OptionPrices, int numOptions) {
    const int      tid = blockDim.x * blockIdx.x + threadIdx.x;
    const int THREAD_N = blockDim.x * gridDim.x;

    for(int i = tid; i < numOptions; i += THREAD_N)
    {
      int option = (d_OptionType[i] == 'P') ? 1 : 0;
      OptionPrices [i] = BlackScholesGPU (d_s[i], d_strike[i],
                         d_r[i], d_v[i], d_t[i], option);
    }
}
```

GPU kernel calls `BlackScholesGPU` device function

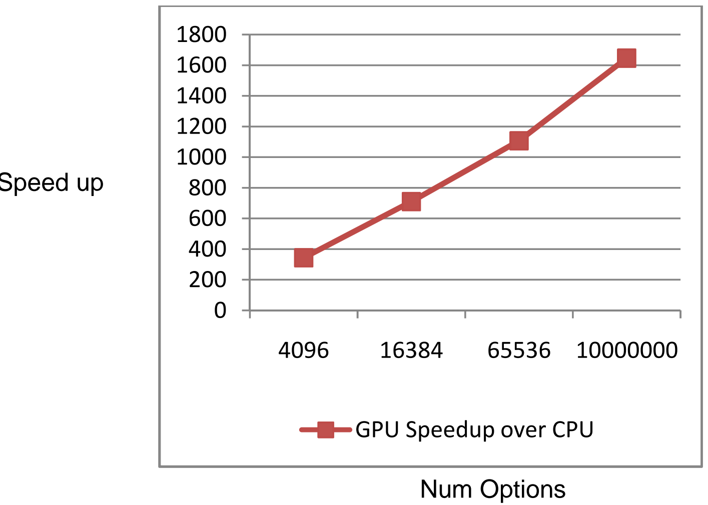
The `BlackScholesGPU` function does all the computation. In serial approach, the main loop which iterates over N options. In cuda based parallel approach we multiple threads operate on data in parallel.

Benchmark results

The benchmarks were conducted on CUDA Testbed at NUST-SEECS, see appendix B4 for details of the machine.

GPU performance evaluation – naïve approach

We will first evaluate our naïve implementation of Blackscholes. The data sets for Blackscholes were varying in nature. The maximum data set was 10 million options. When the experiments were performed, we obtained these results on our Cuda TestBed.



In these graphs, we see that the speed up is being increased as we go on increasing the number of options. The reason is that, on smaller data set, CPU also takes less time, although not lesser than GPU. But as the number of options increases the CPU time becomes greater and greater but GPU time does not increase too much.

GPU Naïve Approach Problem

In naïve approach (using structure of *OptionData*), when we analysed the code, we came to know that using *OptionData* structure would rise the major issue of poor memory accesses in parallelization on GPU which is called *uncoalesced* approach.

As we know that GPUs operate on SIMD (Single Instruction Multiple Data) mechanism. If data is in multiple of 4-bytes e.g. (integers or float) and present on consecutive memory locations, so that each thread should access them in parallel manner, it is called coalesced memory access. To understand the concept of Coalescing and Non-Coalescing consider the following figure.

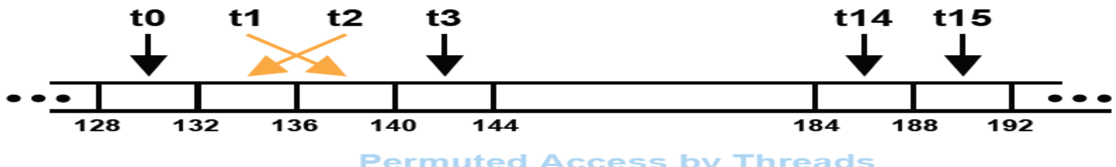


Figure 49: Memory Accesses in non-coalesced fashion

Figure 50: Memory Accesses in Perfectly Coalesced fashion

In our naïve implementation, where data is in the form of structure of options is the cause of non-coalescing. To understand the problem, consider the following figure:

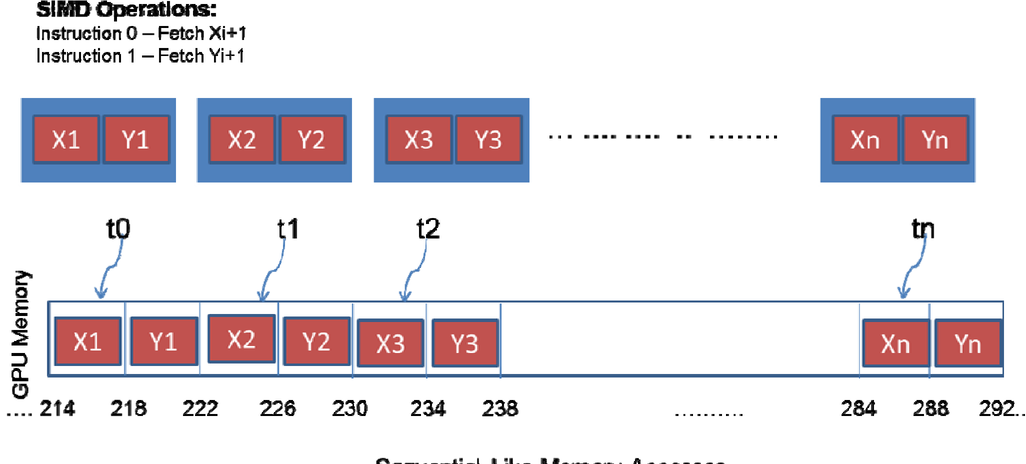


Figure 51: Blackscholes GPU naïve implementation non-colaescing Problem

Since the data is not in multiples of 4-bytes and also not on consecutive locations, so threads access the data in sequential manner e.g. t0 access first and then t1 and then so on.

GPU Optimized Approach And Results

To address the problem of uncoalesced accesses, we changed the *OptionData* structure approach which was the main cause of the problem. We proposed the solution, in which we used arrays of variable separately rather putting them into one structure and then creating array of that structure.

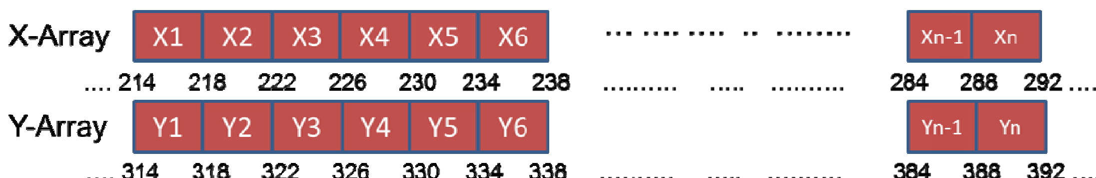


Figure 52: Blackscholes GPU Optimized implementation strategy

In this figure, we are assuming that the variables to calculate the option price are X and Y. We have discarded the structure based approach and created array of each individual variable. Now, when CUDA threads will schedule, the will

operate in SIMD manner, which will result in efficient memory accesses which we call Coalescing.

SIMD Operations:
Instruction 0 – Fetch Xi

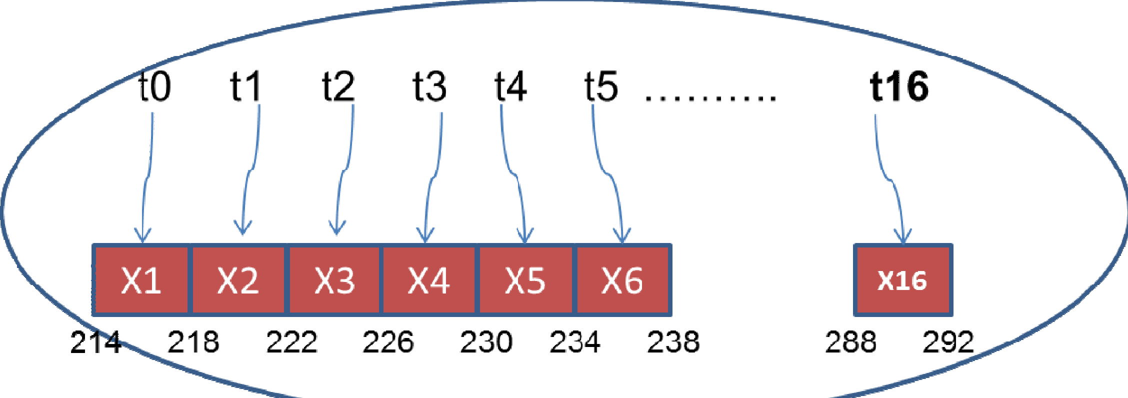
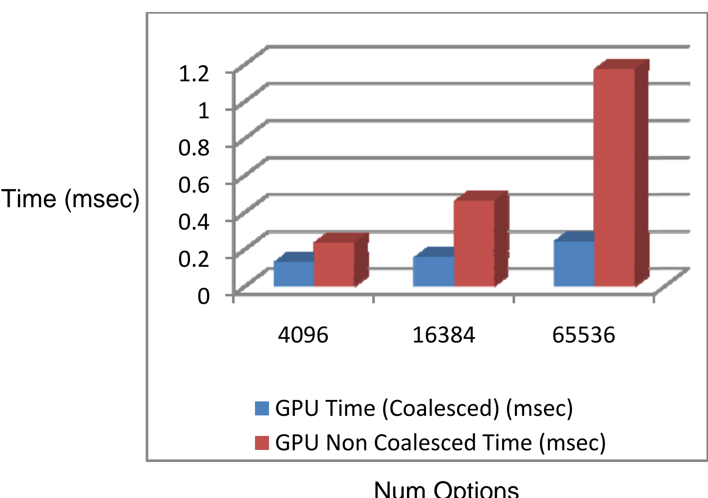
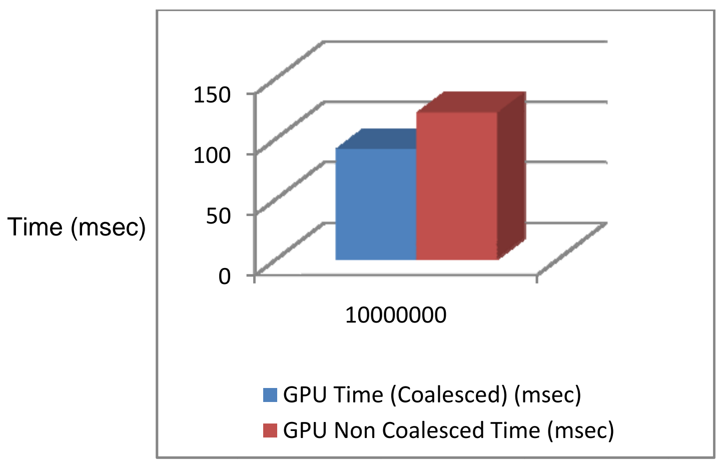


Figure 53: Blackscholes GPU Optimized Coalesced Memory Access

Here we see that cuda threads operate in parallel fashion which executing the instruction. We can see the difference between naïve approach which was causing uncoalescing and optimized approach with coalesced memory accesses in the graph below:





Blackscholes GPU naïve and Optimized approach comparison

Chapter 7

VISUALIZATION

FLUIDANIMATE

Properties of the demo:

1. Gravity Effect and Collision Demo
2. Num particles = 35K
3. External Acceleration is applied TOWARDS the gravity

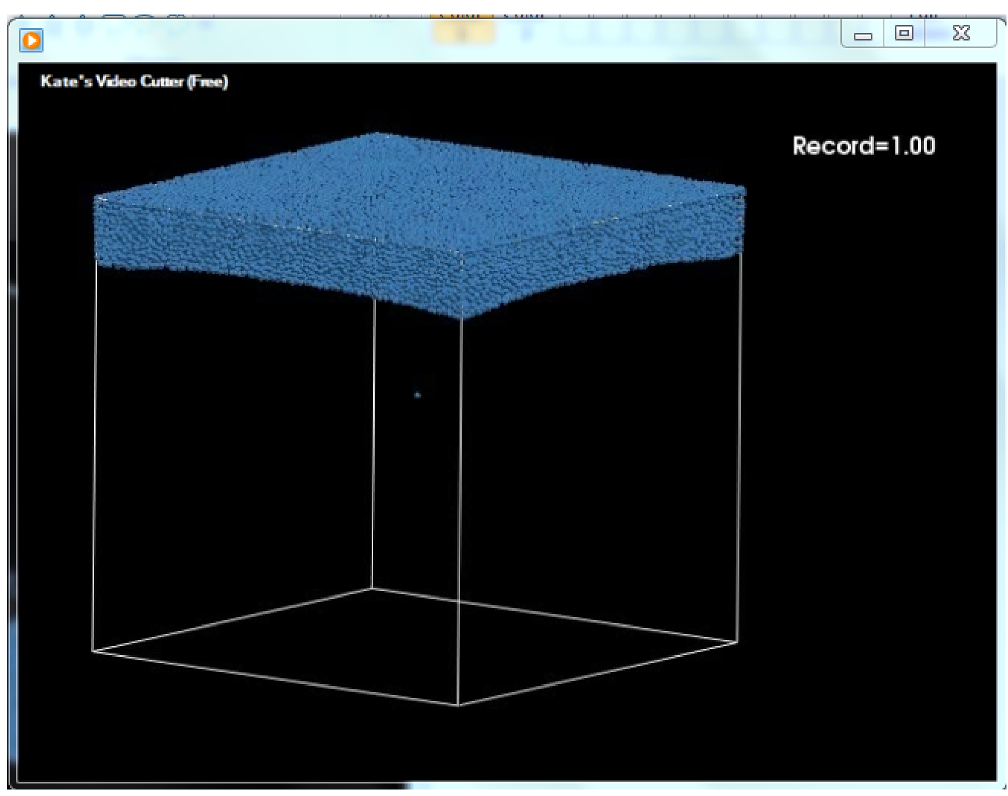


Figure 54: Initial stage, Fluidanimate demo

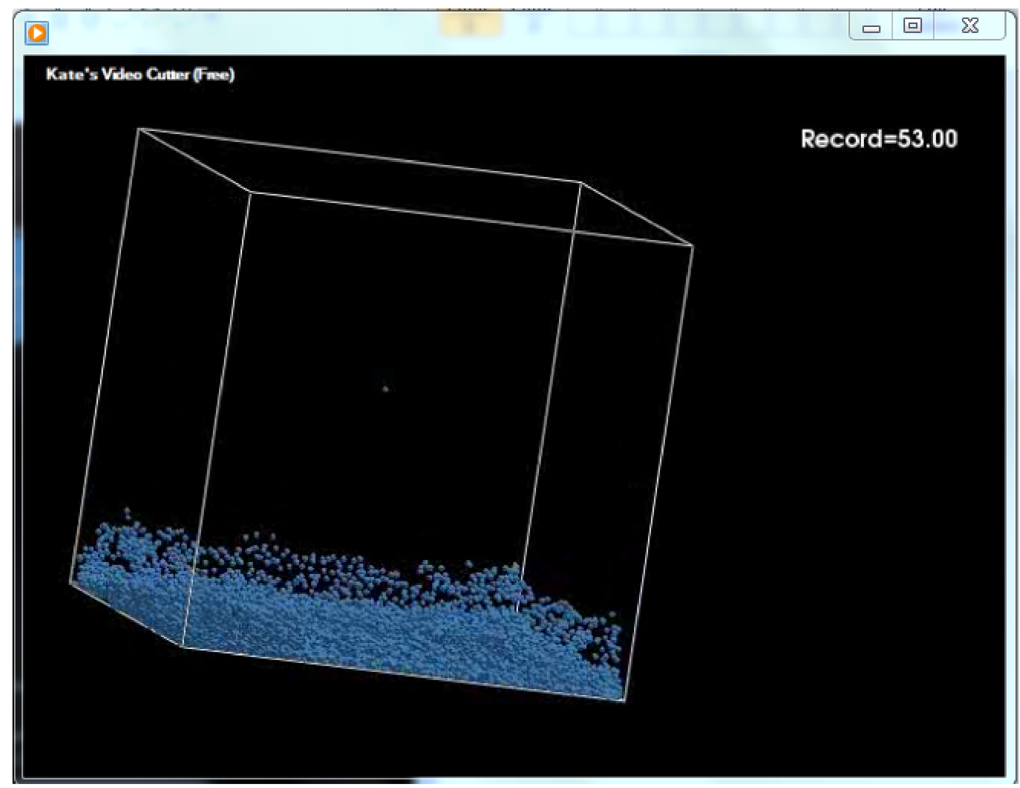


Figure 55: After 53 frames, Fluidanimate demo

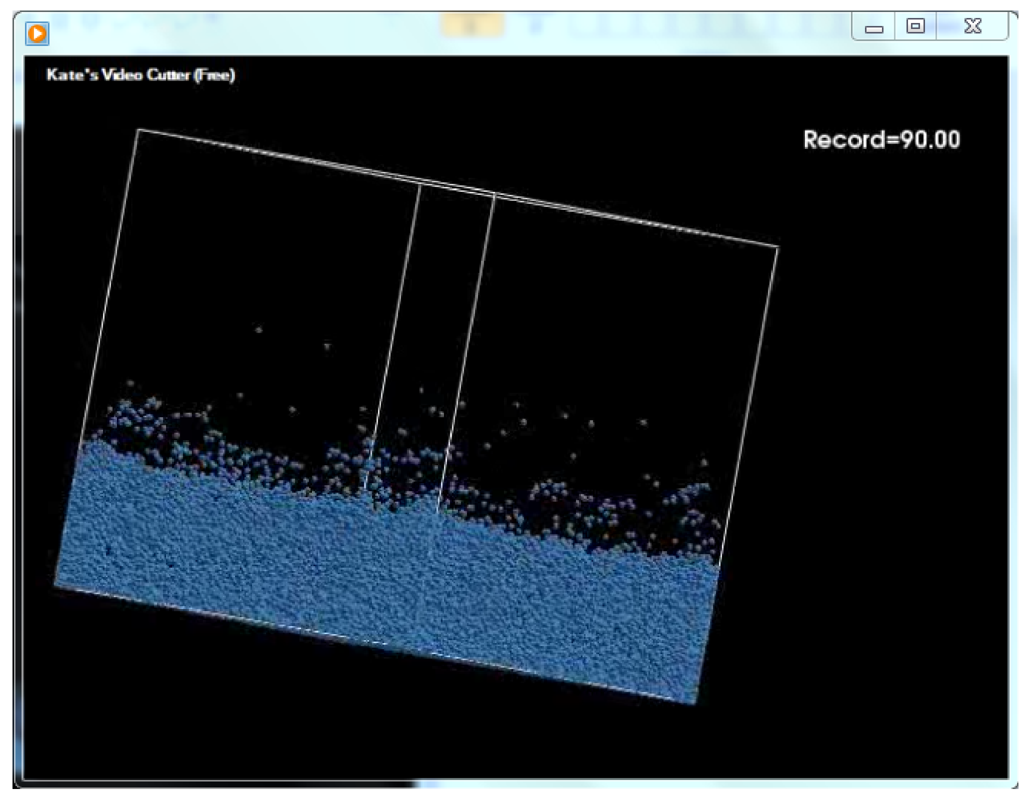


Figure 56: After 90 frames, Fluidanimate demo

This and more demos are uploaded on YouTube, refer to the links below.

FYP Demo Video – 01 **Gravity Effect and Collision Demo**
Num particles = 35K
External Acceleration is applied TOWARDS the gravity.
http://www.youtube.com/watch?v=2djzOTsTZRQ

FYP Demo Video – 02 **Gravity Effect and Collision Demo**
Num particles = 35K
External Acceleration is applied TOWARDS the gravity.
http://www.youtube.com/watch?v=wOcqX0dCjJQ

FYP Demo Video – 03 **Fluid Initial Compression of empty space Demo**
Num particles = 100K
External Acceleration is applied towards the gravity.
http://www.youtube.com/watch?v=7yZvWBDhUkE

OIL RESERVOIR SIMULATION

Properties of the demo:

Available on YouTube: http://www.youtube.com/watch?v=J6J0TO0Q_MQ

1. A reservoir was discretized into 50 x 8 grid points.
2. 4 oil producing wells were installed on grid point (10, 0), (11, 0), (20, 0) and (21, 0) with 600, 400, 600, and 400 STB/D production rate.
3. On the basis of flow of fluids in the reservoir PDEs (Partial Differential Equations) were formed.
4. The PDEs were converted into Linear Equations.
5. The system of Linear Equation was solved using Parallel CG for Pressure of the reservoir.
6. The simulation ran for 400 time steps where Delta T (change in Time) for 5 days.

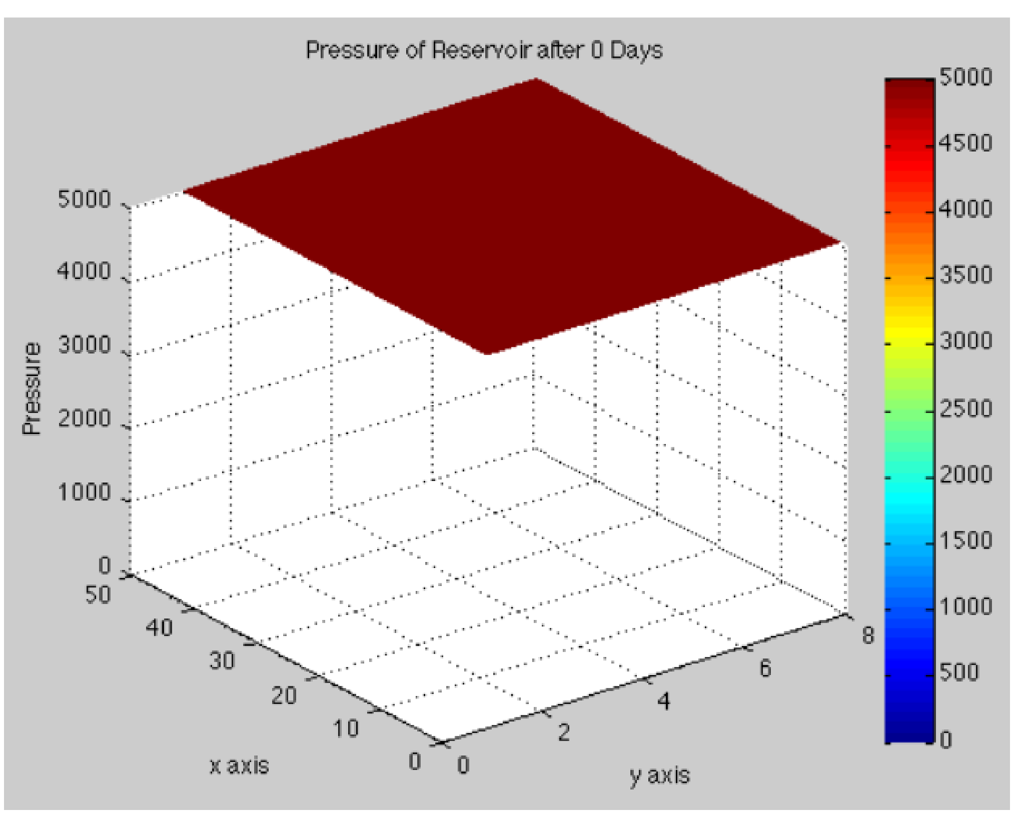


Figure 57: Initial stage, Oil Reservoir Simulation demo

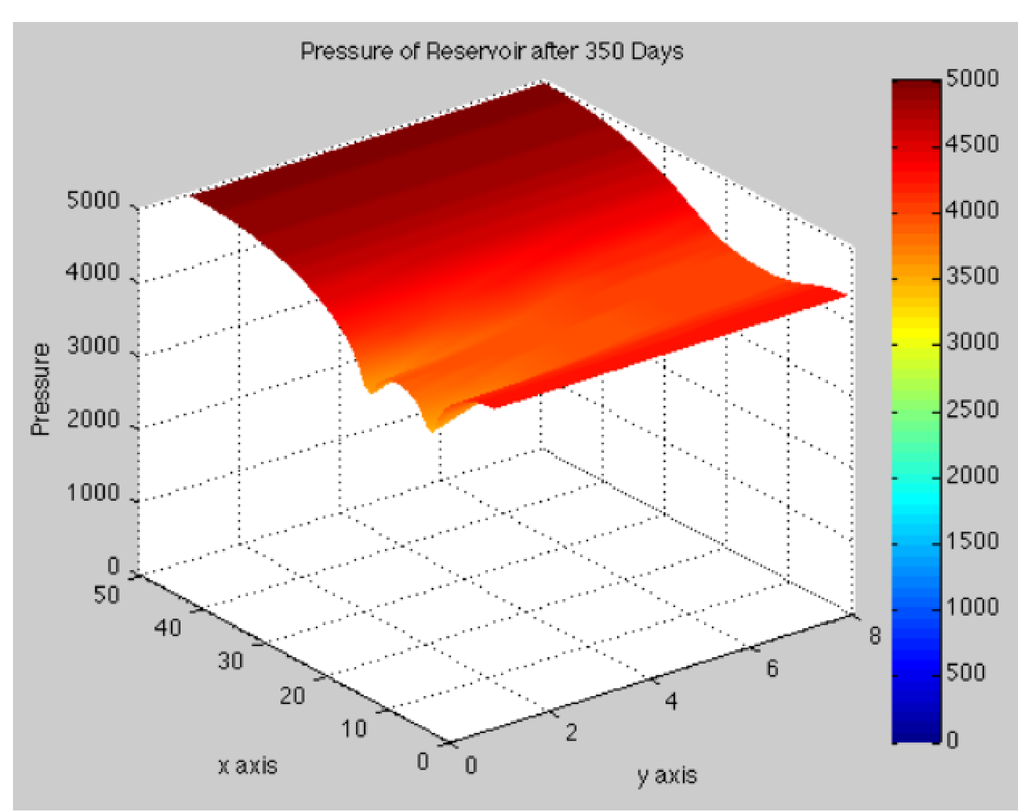


Figure 58: After 350 days, Oil Reservoir Simulation demo

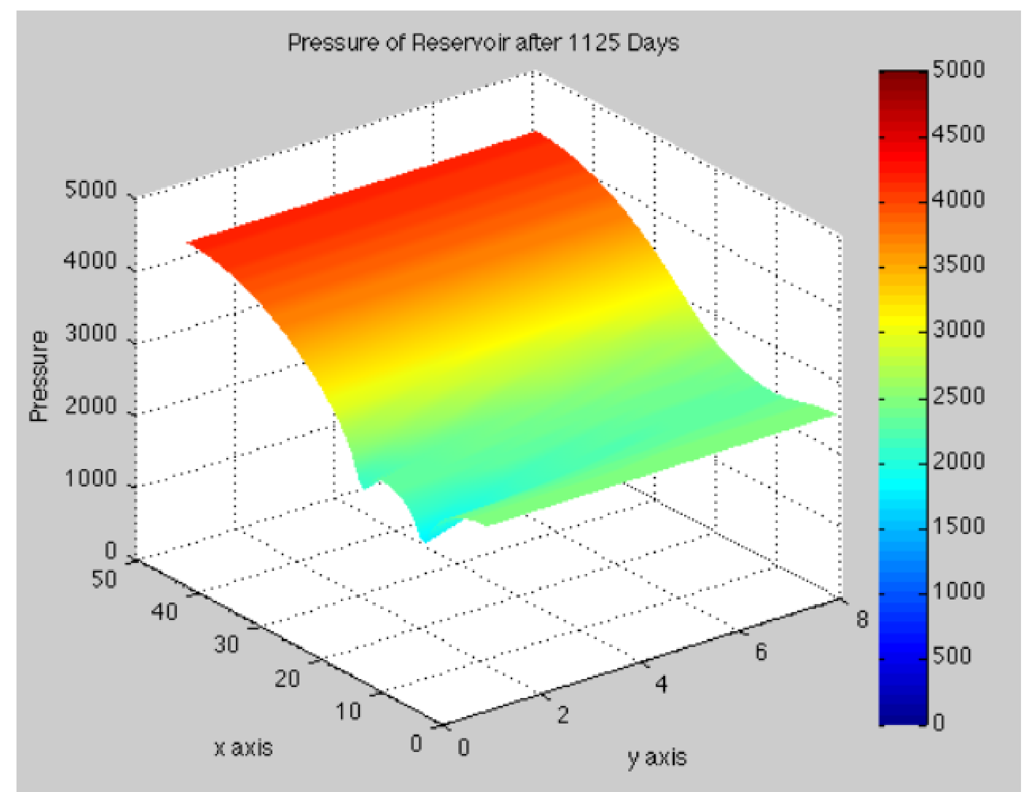


Figure 59: After 1125 days, Oil Reservoir Simulation demo

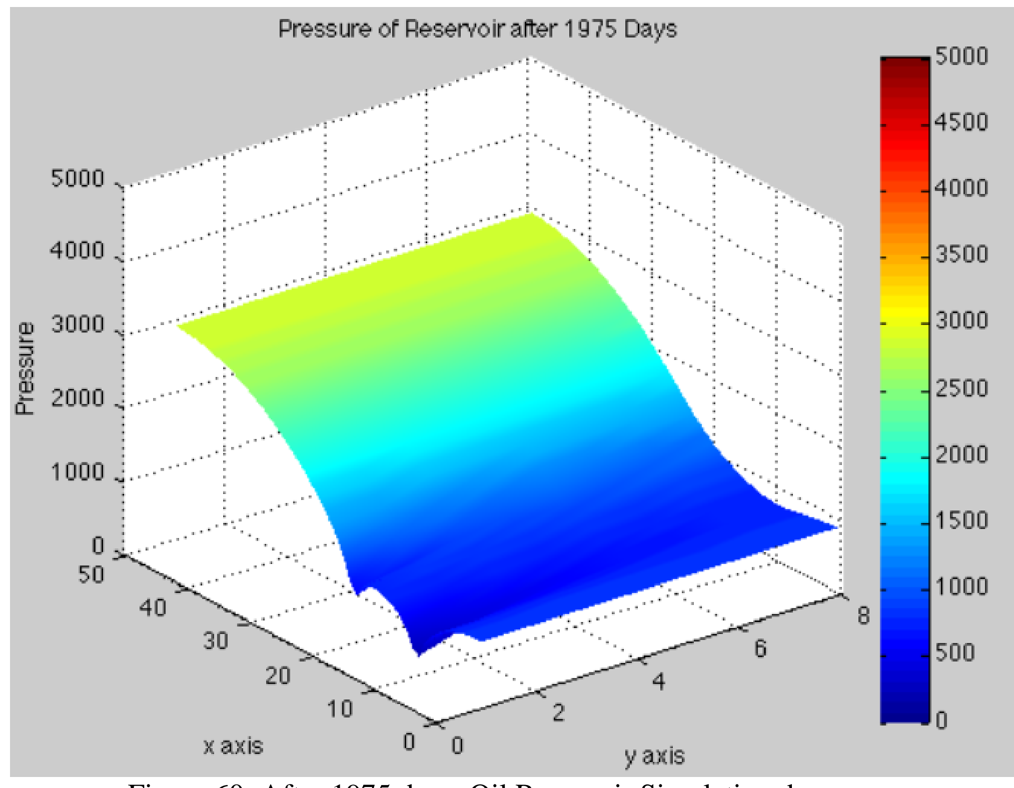


Figure 60: After 1975 days, Oil Reservoir Simulation demo

Chapter 8

CONCLUSION & FUTURE WORK

The aim of our proposed project was to implement some widely used scientific simulation on state of the art High Performance Computing Architectures. The three simulations were chosen as case study to achieve this goal. We implemented these applications on Distributed Memory Architectures, Shared Memory Architectures and Many Core GPUs. We designed and implemented these simulations in an optimized way and proved the research hypothesis; the algorithm design approach we followed can lead to best optimized implementation. We proposed different possible implementation and discussed their pros and cons and finally we proposed some best possible techniques to overcome the flaws of existing non-scalable parallelization approaches. We discussed the results of each parallelization technique, implemented and suggested some improvement in existing algorithms which could lead to significant performance increase. Finally, we demonstrated our application by providing visualization to these applications. Although the visualizations were not rendered on real time but the basic purpose of visualization was to demonstrate the concept behind these simulation.

We have implemented these scientific applications on different HPC architectures, in future the project can be taken a step further where nested parallelism can be used i.e. MPI + OpenMP and the applications can also be ported to hybrid architectures i.e. MPI + CUDA.

Chapter 7

References

1. *Computational Science: Ensuring America's Competitiveness.* s.l. : President's Information Technology.

2. [Online] http://code.google.com/p/stanford-cs193g-sp2010/wiki/GettingStartedWithCUDA.

3. parallel programming guide. *nvidia.com.* [Online] developer.download.nvidia.com.

4. [Online] https://computing.llnl.gov/tutorials/mpi/.

5. http://openmp.org. [Online] http://openmp.org/wp/about-openmp/.

6. http://www.nvidia.com. [Online] http://www.nvidia.com.

7. *CPU and GPU Co-processing for Sound - Master of Science in Computer Science Thesis.* s.l. : Norwegian University of Science and Technology, 2010.

8. *Simulation of Free surface flows with SPH.* **J.J Monaghan, M.C Thompson and K. Hourigan.** Lake Tahoe : ASME Symposium on Computational Methods in Fluid Dynamics, 1994.

9. *Nvidia's Particle-based Fluid Simulation by Simon Green.* s.l. : Nvidia, 2008.

10. **Ellero, Dr Marco.** *Particle Simulation Methods for Fluid Dynamics Lecture04 Monte Carlo Methods.* s.l. : Institute of Aerodynamics, Technical University Munich.

11. **Matthias Müller, Barbara.** *Particle-Based Fluid-Fluid Interaction.* .

12. **Turgay Ertekin, J.H. Abou-Kassem & G.R. King.** *Basic Applied Reservoir Simulation.* 2001. ISBN:978-1-55563-089-8.

13. *Development Of Reservoir Characterization Techniques And Production Models For Exploiting Naturally Fractured Reservoirs.* s.l. : The University of Oklahoma Office of Research Administration.

14. Iterative/Direct method. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Iterative_method.

15. *Implementing parallel conjugate gradient on the EARTH multithreaded architecture.* **Chen, Fei, Theobald, K.B. and Gao, G.R.** s.l. : Cluster Computing, IEEE International Conference on, 2004.

16. **Math, ragujevac J.** *PARALLEL ALGORITHM FOR SOLVING THE BLACK-SCHOLES EQUATION.* 2005.

17. free_black_scholes_model. [Online] http://www.optiontradingpedia.com/free_black_scholes_model.htm.

18. [Online] http://www.strw.leidenuniv.nl/~deul/practicum/html/parallel11.php?node=6611.

19. Blocking vs. Non-blocking Communication under. [Online] http://webcache.googleusercontent.com/search?q=cache:SlHxK1xSW-UJ:www.tu-chemnitz.de/sfb393/Files/PS/sfb98-18.ps.gz+MPI+blocking+communication+paper&cd=3&hl=en&ct=clnk&gl=pk&source=www.google.com.pk.

20. [Online] http://beige.ucs.indiana.edu/B673/node153.html.

21. **Seung-Jai Min, Ayon Basumallik, Rudolf Eigenmann.** *Optimizing OpenMP Programs on Software Distributed Shared.*

22. NAS Parallel Benchmarks. [Online] http://www.nas.nasa.gov/Resources/Software/npb.html.

23. *Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms.* **John G. Lewis, Robert A. van de Geijn.**

24. *Evaluation of Cache Coherence Protocols on Multi-Core Systems with Linear Workloads.* **Yong J. Jang, Won W. Ro.** s.l. : ISECS International Colloquium on Computing, Communication, Control, and Management, 2009 .

25. **Intel.** Sparse Matrix Storage Formats. *http://software.intel.com/en-us/.* [Online] [Cited: 16 July 2011.] http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/win/mkl/refman/appendices/mkl_appA_SMSF.html.

26. *Performance Evaluation of Multithreaded Sparse Matrix-Vector Multiplication using OpenMP.* **Shengfei Liu1, Yunquan Zhang, Xiangzheng Sun, RongRong Qiu.** s.l. : 11th IEEE International Conference on High Performance Computing and Communications, 2009.

27. *Perfomance Models for Blocked Sparse Matrix-Vector Multiplication kernels* . **Vasileios Karakasis, Georgios Goumas, Nectarios Koziris.**

28. [Online] http://hpc.seecs.nust.edu.pk/hardware.php.

29. [Online] http://zone.ni.com/devzone/cda/tut/p/id/6097.

30. [Online] http://webcache.googleusercontent.com/search?q=cache:SlHxK1xSW-UJ:www.tu-chemnitz.de/sfb393/Files/PS/sfb98-

18.ps.gz+MPI+blocking+communication+paper&cd=3&hl=en&ct=clnk&gl=pk&source=www.google.com.pk.

31. **Yun Zhang, Mihai Burcea, Victor Cheng, Ron Ho and Michael Voss.** *An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs.*

32. **Gjermundsen, Aleksander.** *CPU and GPU Co-processing for Sound - Master of Science in Computer Science Thesis.* s.l. : Norwegian University of Science and Technology, 2010.

Chapter 8

Appendix

A1

The Matrix Market (MM) exchange formats provide a simple mechanism to facilitate the exchange of matrix data. The file formats were designed and adopted for the Matrix Market, a NIST repository for test data for use in comparative studies of algorithms for ***numerical linear algebra.***

B1

- Xeon Intel Quad Core 5355
- 100 Computing node - 200 CPU
- 800 coreQuad Core 2.66 GHZ
- 8.320 TeraFlops of Computing Power

B2

It is a cluster comprising of nine nodes connected through a 24 port Gigabit Ethernet switch. Each node has 4 GB of main memory and contains an Intel Xeon Quad-Core Processor (28). The processor is clocked at 2.4 GHz with a rated FSB of 1066 MHz. The processor is a Multi-Chip Module (MCM) and is not a pure quad core design. Each two cores share an L2 cache of 4MB making a total of 8 MB cache for the entire chip. Each core has a 32 KB L1 Data and Instruction cache with 8-way set associatively. L2 cache is 16-way set associative and has 64-byte line size.

B3

It is a cluster comprising four nodes connected through Myrinet Optical Fibre Gigabit Ethernet and Fast Ethernet. The cluster contains 64 Ultra SPARC IV+

processors. Each node contains 16 processors (SMA). The cluster has total of 128GB memory installed on it. We performed our experiments on one node of the cluster having 16 processors with shared memory.

B4

Before we go into the detail discussion of performance, we will discuss our Deployment TestBed. We have conducted experiments on our test bed which we call CUDA TestBed. It contains Intel's Nehalem microarchitecture Quad-core processor with capability of launching eight hardware threads. It also has eight GB of system memory installed on it along with GTX 480 graphics card powered by nVidia. NVidia is considered a leading GPU manufacturer in the market. Its GTX 480 is considered a giant in GPU computing. It has 480 CUDA cores with processing clock of 1401 MHz.