

**EXPERIMENTAL EVALUATION OF PARALLEL PROGRAM  
SCALABILITY ON XEON PHI SMP**

BY

**ALLAM ABDALGANI M.A FATAYER**

A Thesis Presented to the  
DEANSHIP OF GRADUATE STUDIES

**KING FAHD UNIVERSITY OF PETROLEUM & MINERALS**

DHAHRAN, SAUDI ARABIA

1963 ١٣٨٣

In Partial Fulfillment of the  
Requirements for the Degree of

**MASTER OF SCIENCE**

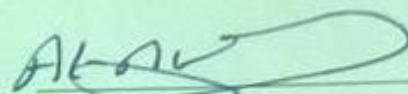
In

COMPUTER ENGINEERING

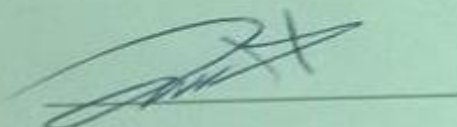
December, 2014

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS  
DHAHRAN- 31261, SAUDI ARABIA  
DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Allam Fatayer** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING.**



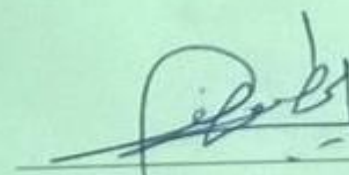
Dr. Mayez Al-Mouhamed  
(Advisor)



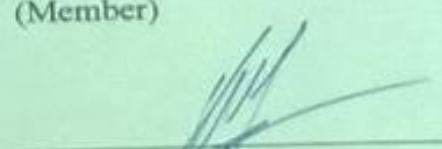
Dr. Ahmad Almulhem  
Department Chairman



Dr. Salam A. Zummo  
Dean of Graduate Studies



Dr. Wasfi Ghassan Al-Khatib  
(Member)



Dr. Muhamed Fawzi Mudawar  
(Member)

29/1/15  
Date

© Allam Fatayer

2014

## DEDICATION

*I dedicate this work to My Precious  
Parents,  
Wife, Sons, Sisters,  
And Brothers,  
Whose patience, continuous prayers  
and  
perseverance led to this  
accomplishment*

## ACKNOWLEDGMENTS

All praise and thanks are due to Almighty Allah, Most Gracious and Most Merciful, for his immense beneficence and blessings. He bestowed upon me health, knowledge and patience to complete this work. May peace and blessings be upon prophet Muhammad (PBUH), his family and his companions.

Thereafter, acknowledgement is due to KFUPM for the support extended towards my research through its remarkable facilities and for granting me the opportunity to pursue graduate studies

I acknowledge, with deep gratitude and appreciation, the inspiration, encouragement, valuable time and continuous guidance given to me by my thesis advisor, Dr. Mayez Al-Mouhamed. I am also grateful to my Committee members, Dr. Wasfi Ghassan Wasfi Al-Khatib and Dr. Muhamed Fawzi Mudawar for their constructive guidance and support and valuable suggestions and comments throughout the study.

Special thanks are due to my colleagues at the university, Dr. Ayaz ul Hassan Khan for his help in explaining STRASSEN, Amran Al-Aghbari and Mohammed Al-Asali in their important contributions in the coding used in BARNES HUT N-BODY simulation, and many others for their help and support.

Finally, I have tried my best to avoid any mistakes or inaccurate data and information in this study. I apologize for any mistake that was beyond my understanding and knowledge, and may ALLAH forgive me.

# TABLE OF CONTENTS

<b>ACKNOWLEDGMENTS</b> .....	<b>V</b>
<b>TABLE OF CONTENTS</b> .....	<b>VI</b>
<b>LIST OF TABLES</b> .....	<b>IX</b>
<b>LIST OF FIGURES</b> .....	<b>X</b>
<b>THESIS ABSTRACT</b> .....	<b>XII</b>
<b>ملخص الرسالة</b> .....	<b>XIII</b>
<b>CHAPTER 1 INTRODUCTION</b> .....	<b>1</b>
<b>1.1 High Performance Computing</b> .....	<b>1</b>
<b>1.2 Methodology</b> .....	<b>2</b>
<b>1.3 Contributions Summary</b> .....	<b>5</b>
<b>1.4 Thesis Skelton</b> .....	<b>7</b>
<b>CHAPTER 2 LITERATURE REVIEW</b> .....	<b>8</b>
<b>2.1 Many-core Processors</b> .....	<b>8</b>
<b>2.2 Challenges For Programming Many-core</b> .....	<b>11</b>
<b>2.3 Many-core Programming Model</b> .....	<b>12</b>
2.3.1 Explicit Threading .....	13
2.3.2 Message Passing Interface ( MPI ) .....	13
2.3.3 OpenMP .....	14
<b>2.4 Related work</b> .....	<b>16</b>
<b>CHAPTER 3 THE XEON PHI MANY INTEGRATED CORE</b> .....	<b>23</b>
<b>3.1 Introduction</b> .....	<b>23</b>
<b>3.2 Many Integrated Core Architecture</b> .....	<b>24</b>
<b>3.3 Cache Structure and Coherency Protocols</b> .....	<b>29</b>

<b>3.4</b>	<b>Software Stack Architecture View .....</b>	<b>32</b>
<b>3.5</b>	<b>Programming Model .....</b>	<b>33</b>
<b>3.6</b>	<b>Thread Execution Model .....</b>	<b>35</b>
3.6.1	Thread Affinity.....	38
3.6.2	Thread Affinity Type.....	40
<b>CHAPTER 4 STATIC PROBLEMS.....</b>		<b>42</b>
<b>4.1</b>	<b>Introduction.....</b>	<b>42</b>
<b>4.2</b>	<b>Matrix Multiplication .....</b>	<b>43</b>
4.2.1	Execution Model for MM .....	44
4.2.2	Strassen MM ( S-MM) .....	45
4.2.3	Conclusion .....	59
<b>4.3</b>	<b>Jacobi Solving Linear Equations.....</b>	<b>59</b>
4.3.1	Jacobi Execution Model.....	61
4.3.2	Experiment Results.....	67
4.3.3	Conclusion .....	71
<b>CHAPTER 5 SEMI-STATIC PROBLEM (N-BODY SIMULATION) .....</b>		<b>72</b>
<b>5.1</b>	<b>Introduction.....</b>	<b>72</b>
<b>5.2</b>	<b>BARNES-Hut (BH) Algorithm.....</b>	<b>74</b>
<b>5.3</b>	<b>Related Work .....</b>	<b>76</b>
<b>5.4</b>	<b>Effective parallelization using Cost Zone .....</b>	<b>80</b>
<b>5.5</b>	<b>Distribute Work Using Cost Zone .....</b>	<b>80</b>
<b>5.6</b>	<b>Reserve Locality Using Morton Order.....</b>	<b>82</b>
<b>5.7</b>	<b>Iterative Cost Zone Load Balancing (ICZB) Implementation .....</b>	<b>83</b>
5.7.1	N-body Implementation Steps .....	83
5.7.2	Load Bodies .....	85
5.7.3	Oct-Tree Iterative Creation Algorithm .....	85
5.7.4	Iterative Depth First Tree Traversal.....	87
5.7.5	Sorting Node and Body Arrays .....	88
5.7.6	Converting Oct-Tree Into Data Structure .....	89
5.7.7	Iterative Force Computation .....	90
5.7.8	Iterative Cost Zone Load Balancing (ICZB).....	91
5.7.9	Practical Challenges.....	92
<b>5.8</b>	<b>Implementation Correctness Checking.....</b>	<b>93</b>
<b>5.9</b>	<b>Body Generation and Dataset .....</b>	<b>94</b>

<b>5.10 ICZB Evaluation .....</b>	<b>96</b>
5.10.1 Algorithm Time Distribution.....	96
5.10.2 Speedup of STATIC and ICZB .....	98
5.10.3 Overhead of ICZB.....	99
5.10.4 Locality .....	100
5.10.5 Linearity and Effectiveness of Dynamic Load Balancing.....	101
<b>5.11 Conclusion .....</b>	<b>105</b>
 <b>APPENDIX A: STRASSEN MATRIX-MATRIX MULTIPLICATION CODE .....</b>	 <b>106</b>
 <b>APPENDIX B: JACOBI SOLVER.....</b>	 <b>118</b>
 <b>APPENDIX C: BARNES-HUT N-BODY SIMULATION.....</b>	 <b>139</b>
 <b>REFERENCES.....</b>	 <b>166</b>
 <b>VITAE.....</b>	 <b>169</b>



## LIST OF TABLES

Table 3-1 Extended MESI cache coherency protocol states [29] .....	31
Table 3-2 Global owned locally shared cache coherency protocol state [29] .....	32
Table 5-1 Sequential Barnes-Hut Algorithm .....	76
Table 5-2 Hotspot analysis of the Algorithm Steps .....	97

# LIST OF FIGURES

Figure 1-1	Methodology .....	4
Figure 2-1	Master thread forks a team of threads as needed. Parallelism is added until a desired performance is achieved [23] .....	14
Figure 2-2	OpenMP language extension. (From www.openmp.com) .....	15
Figure 3-1	General layout of the MIC coprocessor. For simplicity, only 8 of the total cores and 4 of the total 8 GDDR5 memory controller shown [27].....	24
Figure 3-2	Basic building block of the MIC .....	26
Figure 3-3	Core architecture for MIC .....	27
Figure 3-4	MIC pipeline data path .....	29
Figure 3-5	MIC Software stack Architecture.....	33
Figure 3-6	MIC Programming Model.....	34
Figure 3-7	Multi-threading Architectural Support in MIC Core [27].....	36
Figure 3-8	MIC Card Thread Context across 60 cores .....	39
Figure 3-9	Scatter Affinity for 6 thread .....	40
Figure 3-10	Compact Affinity for 6 threads .....	41
Figure 3-11	Balanced Affinity for 9 threads.....	41
Figure 4-1	Naive Matrix multiplication code .....	43
Figure 4-2	STRASSEN MM recursion into N level .....	47
Figure 4-3	Pseudo code of Reorder Implementation .....	51
Figure 4-4	Strassen, Submatrix indices.....	51
Figure 4-5	Strassen with MKL, 4 core, MIC .....	53
Figure 4-6	STRASSEN with MKL, 16 core, MIC .....	53
Figure 4-7	STRASSEN with MKL, 32 core, MIC .....	54
Figure 4-8	Strassen with MKL, 60 core, MIC .....	54
Figure 4-9	Speed Up of Strassen relative to 1 core, 8 core.....	55
Figure 4-10	Speed Up of Strassen relative to 1 core, 16 core.....	56
Figure 4-11	Speed Up of Strassen relative to 1 core, 32 core.....	56
Figure 4-12	Speed Up of Strassen relative to 1 core, 60 core.....	57
Figure 4-13	Execution time of MKL on smaller matrix size and different number Cores.....	58
Figure 4-14	JACOBI sequential code implementation .....	60
Figure 4-15	Jacobi Data Layout Representation.....	61
Figure 4-16	Direct Jacobi parallelization Code .....	62
Figure 4-17	Synchronous Jacobi implementation, using work sharing constructs and single construct to optimize overhead.....	63
Figure 4-18	Relaxed Synchronization (RS) execution flow chart .....	65
Figure 4-19	Percentage time spend in synchronization for 100 iteration .....	67
Figure 4-20	Jacobi experiment result for SJ, AJ, RJ for matrix size 1920 .....	68
Figure 4-21	Jacobi experiment result for SJ,AJ,RJ, matrix size 3840.....	69
Figure 4-22	Jacobi experiment result for SJ,AJ,RJ, matrix size 7680.....	69

Figure 4-23 Jacobi experiment result for SJ,AJ,RJ, matrix size 15360.....	70
Figure 4-24 Jacobi experiment result for SJ,AJ,RJ, matrix size 30720.....	70
Figure 5-1 Adaptive Quad Tree of BH for 2D Simulation .....	75
Figure 5-2 Barnes-Hut approximation in computing force for far bodies .....	75
Figure 5-3 Oct-Tree for 3D Barnes-Hut Simulation.....	76
Figure 5-4 Cost Zone Demonstration of work distribution for 8 threads .....	81
Figure 5-5 Morton Order representation, left for 2D, Right for 3D [50].....	82
Figure 5-6 N-Body with Barnes Hut Execution Steps.....	84
Figure 5-7 Oct-tree iterative creation algorithm .....	86
Figure 5-8 illustration of the chosen Morton order in my implementation. The numbers represents the order of selecting cubes. ....	87
Figure 5-9 Tree traversal algorithm. ....	88
Figure 5-10 An example shows the depth-first traversal order. The nodes are sorted in the array according to this traversal. The leaves which represent the bodies also sorted in the array according to this order. Each node also store an index of the next node in the tree	89
Figure 5-11 King Model galaxy, which contains $10^6$ bodies. Plotted using TeraPlot Visualizer .....	94
Figure 5-12 Galaxies generation procedure.....	95
Figure 5-13 Bodies distribution for a data set of 1M.....	95
Figure 5-14 Percentage Execution Time For N-body for Each Step .....	97
Figure 5-15 Speedup of STATIC and ICZB vs. Problem Size ( 1M,2M,3M,4M).....	98
Figure 5-16 Data Read and Write for Static vs. ICZB for 4M bodies .....	100
Figure 5-17 Data Read and Write for Static vs. ICZB for 5M bodies .....	100
Figure 5-18 L2 cache misses for static and ICZB for 4M bodies .....	101
Figure 5-19 L2 cache misses for Static and ICZB for 5M bodies .....	101
Figure 5-20 Linearity of ICZB (1M,2M,3M,4M,5M) 240 thread .....	102
Figure 5-21 Percentage of average relative work deviation of ICZB, 5M, 240 .....	103
Figure 5-22 Percentage of Average Relative Time Deviation, ICZB, 5M .....	104
Figure 5-23 Speedup of ICZB, problem size 5M, 240 .....	104

## THESIS ABSTRACT

<b><u>Name</u></b>	:	ALLAM ABDALGANI M.A FATAYER
<b><u>Title</u></b>	:	EXPERIMENTAL EVALUATION OF PARALLEL PROGRAM SCALABILITY ON XEON PHI SMP
<b><u>Degree</u></b>	:	Master of Science
<b><u>Major Field</u></b>	:	Computer Engineering
<b><u>Date</u></b>	:	December, 2014

As the time of Moore's Law and expanding CPU clock rates nears its halting point the condense of chip and hardware design has moved to expanding the number of cores present on the chip. These increase can be most clearly seen in the rise of the Many Integrated Core processors (MIC). Programming for these chips delivers another set of difficulties and concerns In this context, I present an experimental evaluation of parallel program scalability on the MIC Shared Memory Multiprocessor (SMP) using OpenMP programing paradigm. I address two classes of applications 1) Static and 2) Semi static. For first class I select a set of applications from the class of Basic Linear Algebra and numerical algorithms (Matrix-Matrix Multiplication (MM) and JACOBI SOLVER). Particularly, I analysis, optimize and implement these applications. For MM I used the STRASSEN matrix multiplication algorithm. The basic Strassen-MM (S-MM) algorithm time complexity of is  $O(N^{2.807})$  instead of  $O(N^3)$  of standard MM algorithm. my optimizations are based on a reordering approach to reduce the storage, use of a depth first walk (DFW), and invocation of the MKL optimized library for matrix-matrix multiplications. The results of MM using STRASSEN outperform Math Kernel Library (MKL) within large matrix size with percentage from 8% to 24%. For JS, I noticed that it does not scale well because of the excessive synchronization overhead, which must be implemented across all the working threads. To improve JS scalability, I explored (1) Synchronous Jacobi (SJ), (2) Asynchronous Jacobi (AJ), and Relaxed Jacobi (RJ). In SJ I used explicate barrier synchronization. In AJ a non-exact solution is computed because completing threads start the next iterations using current data, which is a mixing of new and old. AJ slows down the convergence rate. In RJ, completing threads at iteration K start the next iteration (k+1) using newly computed data. RJ provides overlap between two iterations at the cost of managing the availability of currently available intermediate results. Experiments show that SJ synchronization time takes 50% from the execution time on matrix size 4096. For exact solutions, my evaluation shows a performance gain of 24.4%, 32.6%, 38.9%, and 57.16% for RJ over SJ for matrices of size 3840, 7680, 15360 and 30720, respectively using 60 cores. For the second class, I select a semi static classical problem (N-Body simulation). In this application, an approximated solution using BARNES-HUT algorithm (BH) is implemented. BH uses an oct-tree, in which each node stores the aggregate mass of all of its children nodes (sub-tree) at their center of mass. Another problem is that the thread load moderately changes from one iteration to another due to body motion in space. A Dynamic Load Balancing (DLB) combined with data locality approach is used to improve Scalability, I call it Iterative Cost Zone Load Balancing (ICZB). My implementation on MIC shows that the execution time and aggregate load scales linearly with the problem size when using 60 cores for problem sizes within the range of 1 million to 4 million. In addition, my DLB-BH provides an increased speedup of 42% and 36% on problem size 1 million and 4 million respectively, as compared to traditional static BH. DLB is recommended as a compiler strategy as one optimization strategy for semi-static applications.

## ملخص الرسالة

الاسم الكامل: علام عبد الغني "محمد عادل" فطائر

عنوان الرسالة: تقييم تجريبي لتدرجية البرامج الموازية في المعالجات عديدة النواة التماثلية

التخصص: هندسة الحاسب الآلي

تاريخ الدرجة العلمية:

قانون مور اصبح يتجه الى نهايته، ويظهر ذلك جليا في ظهور المعالجات متعددة الأنوية. البرمجة لهذه الرقائق تظهر تحديات ومشاكل جديدة في وجه المبرمجين. في هذا السياق، نقوم على تقييم تجريبي لتدرجية البرامج الموازية في المعالجات عديدة النواة ذات التماثلية باستخدام نموذج برمجة OpenMP. قمنا باختيار فئتين من التطبيقات. الفئة الأولى التطبيقات ذات الحمل الثابت والفئة الثانية التطبيقات ذات الحمل المتغير. في الفئة الأولى قمنا باختيار ضرب المصفوفات والتي تصنف من مكتبة علم الجبر الأساسي، وايضا تم اختيار تطبيق من التحليل العددي في حل المعادلات الخطية يطلق عليه اسم JACOBI. في حقيقة الأمر قمنا بتحليل وتنفيذ وتحسين هذه البرامج. في ضرب المصفوفات قمنا باستخدام خوارزمية STRASSEN والتي لها حساب تعقيد اقل من حساب التعقيد لعملية ضرب المصفوفات الأساسية. التحسين لدينا يعتمد على اعادة ترتيب المصفوفات البينية لتقليل الحجم المطلوب، واستخدام المشي الأولي للمصادر، بالإضافة الى استدعاء المكتبة MKL للمصفوفات ذات الحجم الصغير. النتائج اثبتت ان طريقتنا استطاعة ان تتغلب على استخدام المكتبة MKL لوحدها في احجام المصفوفات الكبيرة بنسبه تتراوح من 8% الى 24%. في تطبيق JACOBI لوحظ عدم تدرجية ادائه بسبب الاحتياج الكبير للمزامنة في اثناء التنفيذ و خصوصا في التكرار ما بين جميع ال Thread العاملة. لتحسين العمل تم استكشاف ثلاث انواع من التطبيق (1 المتزامن 2) الغير متزامن (3) المتزامن المسترخي. في المتزامن تم استخدام مزامنه واضحه للعيان. في الغير متزامن تم حذف المزامنة وفي الحالة الأخيرة تم اعادة كتابة المزامنة وذلك من خلال السماح بالتداخل ما بين عمليات التكرار باستخدام النتائج الجزئية من كل Thread. يجدر الإشارة ان الغير متزامن يستخدم النتائج الحالية والتي يمكن ان تكون خليط من النتائج السابقة والحديثة والتي تقلل من سرعة التقاء الخوارزمية بالحل. النتائج اظهرت ان المزامنة تأخذ 50% من وقت التنفيذ في حالة المصفوفة بحجم 4096. الغير متزامن يعطي افضل النتائج بسبب حذف المزامنة ولكن في حالة قبول الحل التقريبي. في حالة الحل الدقيق فان التزامن المسترخي اظهر تحسن بالإداء على التزامن بمقدار 57.16% , 32.6% , 32.6% , 24.4% في الأحجام 3840 7680 15360 30720 باستخدام 60 نواة. في الفئة الثانية تم اختيار مشكلة كلاسيكية تحاكي حركة الأجسام في الفراغ تدعي (N-body). تم تنفيذ حل تقريبي للمشكلة باستخدام خوارزمية Barn-Hut. تعتمد هذه الخوارزمية على بنية الشجرة الثمانية لتمثيل توزيع الأجسام بالفراغ. حيث يتم تخزين البيانات التراكمية للكتلة المركزية في كل عقدة، للشجرات الجزئية التي اسفلها. وايضا من تحديات الخوارزمية هو تغير توزيع الحمل على ال Thread عند الانتقال من خطوة الى الثانية وذلك بسبب حركة الأجسام في الفراغ. للتحسين تم تطوير توزيع للحمل بشكل ديناميكي بالإضافة الى زيادة محلية للبيانات. النتائج اظهرت ان الحمل التراكمي يتناسب بشكل خطي مع وقت التنفيذ باستخدام احجام مختلفة تتراوح من 1 مليون الى 4 مليون. بالإضافة الى ذلك فان هناك تحسين بالتسريع للعملية بمقدار 42% و 36% في المشاكل بحجم 1 مليون و 4 مليون مقارنة بالطريقة الثابتة لتوزيع الأحمال. إن هذه الطريقة نوصي باستخدامها كاستراتيجية في تحسين المترجمات عند القيام بترجمة للمشاكل ذات الحمل المتغير

# CHAPTER 1

## INTRODUCTION

### 1.1 High Performance Computing

Traditionally, scientists employ both experimental and theoretical approaches to solve problems in the fields of science and engineering. With the advent of computer machinery, scientists have been able to transform a given problem into an algorithm, analyze and understand the problem through computing and simulations. Hence, the use of High Performance Computing (HPC) in simulation has now become popular and an important part of the exploratory process that many people believe that the scientific model has been extended to include simulation as an additional proportion [1]. In addition, computing systems have been playing a critical role in scientific computing, and hardware advances have allowed scientists to investigate problems in more details and with higher complexity than what the past eras of hardware could achieve.

Currently, the HPC industry is at a real changing point in its processor architecture because of a decades-long trend of exponentially expanding clock frequencies. The conventional single-core processor architectures are no more ready to exploit of the integrated circuit (IC) technology advances due to some basic issues, such as, power consumption, heat dispersal, and memory wall. Computer architects are searching for

different approaches to use the transistor plan. By incorporating a number of simple processors/cores on a single die, it is considered that this many-core chip technology has higher power-efficiency, improved heat dissipation, better memory latency tolerance, and numerous different profits[2]. Projections and early models indicate that tens, hundreds, if not thousands of general-purpose and/or special-purpose cores will be included on a single chip within a brief period of time. Many researchers suppose that the many-core architectures are going to become the mainstream for parallel computing later on. However, unlike previous hardware evolutions, this shift in the hardware roadmap will have an effect on the scientific computing by posing uncommon difficulties in the management of parallelism, locality, scalability, synchronization, load balancing, energy and fault-tolerance. It is an open question whether the current parallel programming approaches will keep on to scale to future computing systems built with many-core processors.

## **1.2 Methodology**

Scientific phenomena governed by partial differential equations (PDEs) can range from solid mechanics to fluid mechanics and electrodynamics, including any of the possible couplings. The solution of these equations can be approximated with the aid of computers by a discretization (and possibly linearization) and the subsequent numerical solution of the resulting sparse set of linear equations. This work is concerned with the fast solution of a set of scientific applications that chosen from Basic Linear Algebra, Numerical algorithm and N-body classical problem. Although these applications are the simplest model problem for, e.g., fluid flow simulation, they are still very useful as a building

block for the “physics-based” preconditioning of very complex scientific applications governed by coupled systems of PDEs. The regularly expanding interest of reality in the simulation of the complex scientific and engineering three-dimensional (3D) problems faced these days ends up with the solution of very large linear systems with several hundreds and even thousands of millions of equations/unknowns.

The solution to these systems in a moderate time needs a large amount of computational resources provided by current MIC machines. It is therefore vital to design parallel algorithms able to take advantage of their underlying architecture. So, a set of scientific applications were chosen to study the scalability and performance of MIC coprocessor using OpenMP programming paradigm. In addition, optimization techniques are proposed and implemented to reduce the execution time. The main objective of my work is to increase performance for a set of applications by decreasing the execution, Flow chart in Figure 1-1 outlines my methodology.

First, the sequential code was implemented and tested. In addition, the performance metrics are recorded. After that, the parallel code is implemented by inserting the proper OpenMP construct.. In this step the parallel program errors are solved and elevated from the applications.

Secondly, application profiling for parallel programs are done to understand constrains and scalability problems of the applications. Then depending on that an experimental programming, debugging and profiling are repeated to propose an optimization. So, optimizations are a handy way.



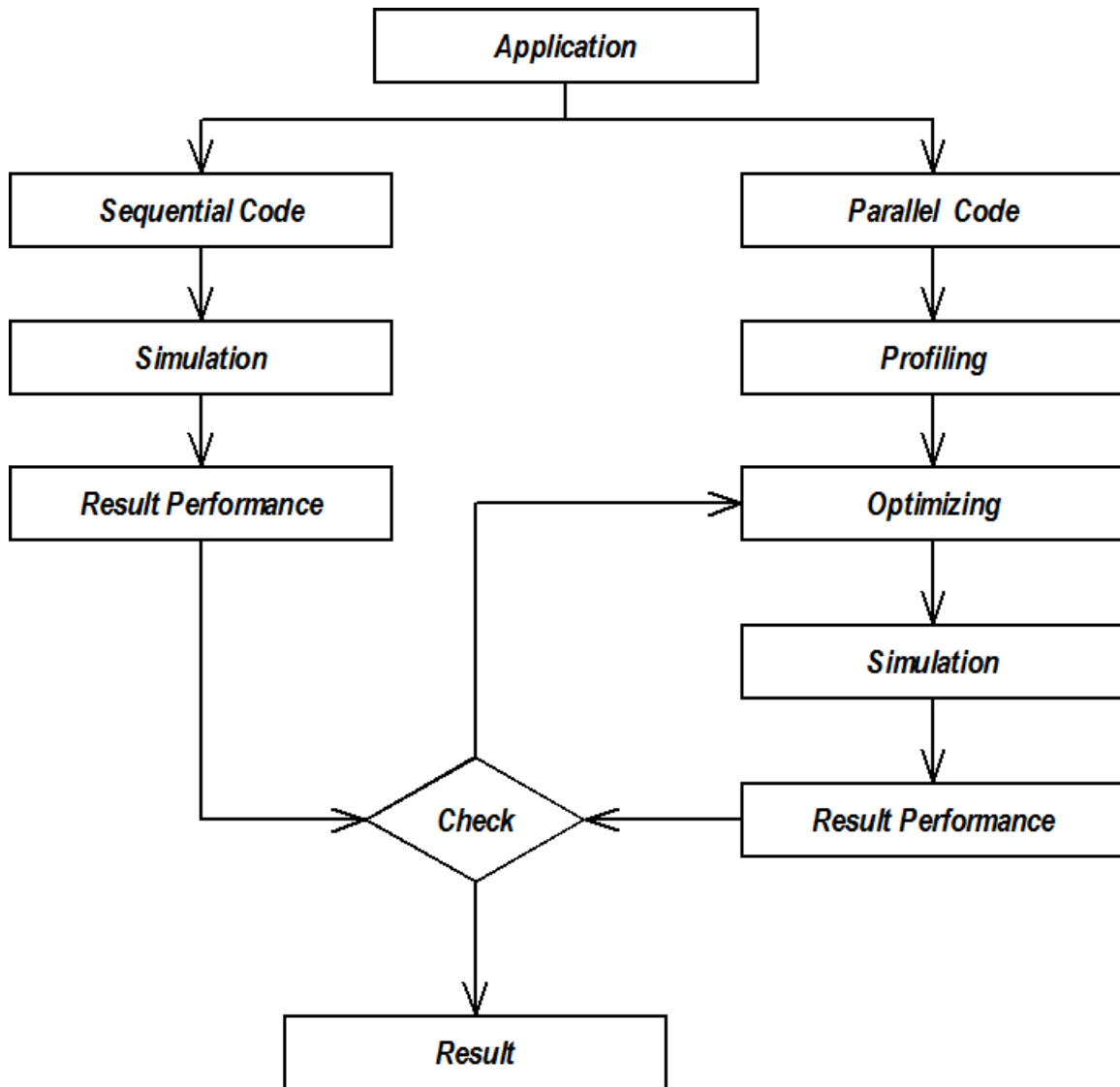


Figure 1-1 Methodology

### 1.3 Contributions Summary

As the time of Moore's Law and expanding CPU clock rates nears its halting point the condense of chip and hardware design has moved to expanding the number of cores present on the chip. These increase can be most clearly seen in the rise of the Many Integrated Core processors (MIC). Programming for these chips delivers another set of difficulties and concerns In this context, I present an experimental evaluation of parallel program scalability on the MIC Shared Memory Multiprocessor (SMP) using OpenMP programing paradigm

I address two classes of applications, static problems (basic linear Algebra and numerical algorithms) where the load is fixed after partitioning, and semi static problem (N-body Simulation) where the load change moderately after partitioning. For first class I used the STRASSEN Matrix Multiplication (SMM) and Jacobi Solver (JS) of a system of linear equations for which the load is static across the iterations. The basic STRASSEN-MM (S-MM) algorithm having time complexity of  $O(n^{2.807})$  instead of  $O(n^3)$  of standard MM algorithm. My optimizations are based on a reordering approach to reduce the storage, use of a depth first walk (DFW), and invocation of the MKL optimized library for matrix-matrix multiplications. In DFW, all available machine parallelism is used in each depth expansion. Using a few recursions, my approach is useful to reduce execution time over that of the MKL library using the traditional MM algorithm. The profitability of my approach over MKL increases with the matrix sizes

In iterative JS, the threads need to read a vector that was computed by all the working threads before starting the next iteration. It is noticed that due to the above data layout JS does not scale well because of the excessive synchronization overhead, which must be implemented across all the working threads. To improve JS scalability, I explored (1) Synchronous Jacobi (SJ), (2) Asynchronous Jacobi (AJ), and Relaxed Jacobi (RJ). In SJ I used explicate barrier synchronization. In AJ a non-exact solution is computed because completing threads start the next iterations using current data, which is a mixing of new and old. AJ slows down the convergence rate. In RJ, completing threads at iteration  $K$  start the next iteration  $(k+1)$  using newly computed data. RJ provides overlap between two iterations at the cost of managing the availability of currently available intermediate results.

For the second class the  $N$ -body simulation is considered as a model of semi static computations. A brute force approach for computing the gravitational forces for  $N$  bodies is on the  $O(N^2)$ . The BH approximation enables treating a group of bodies as one if these are far enough from a given body. This drops the computational complexity to  $O(N \log N)$  when using BH. BH uses an oct-tree, in which each node stores the aggregate mass of all of its children nodes (sub-tree) at their center of mass. Another problem is that the thread load moderately changes from one iteration to another due to body motion in space. Therefore, a Static problem partitioning strategy for BH (S-BH) is likely to suffer from accumulated load unbalance. It is well known that dynamic load balancing (DLB) improves BH scalability. However, DLB is complex because of the need to measure the Dynamic Load (DL) and adopt an adequate data structure to minimize runtime overheads. In the beginning of iteration  $K$ , the body slowly motion enables estimating the

DL for  $K+1$  as being the aggregate load measured by all the threads in iteration  $K$ . Thus DLB is implemented by evenly partitioning the DL over the threads so that to preserve the data locality to the best possible. I implemented DLB-BH using an efficient data structure to ease load redistribution together with oct-tree implementation.

## **1.4 Thesis Skelton**

The rest of the thesis is organized as follows. Chapter 2 presents state of the art for many-core processors, challenges in programming many-core, programming paradigm and related work. Chapter 3 shows a brief description and analysis of MIC architecture. Particularly, I will focus on the hardware point of view and combine it with programming paradigms that support. Chapter 4 describes my optimization method in static problems (S-MM, JACOBI). For the S-MM I will present my execution time optimization technique over the standard MM. On the other hand, for JS I will explain my relaxed synchronization technique. Chapter 5 reports the N-body simulation problem. First, I will explain my dynamic load balancing schema for optimization and scalability on MIC. Second, locality technique applied in N-body Simulation to increase speedup.

## CHAPTER 2

### LITERATURE REVIEW

#### 2.1 Many-core Processors

An evolution happened in Central Processing Units (CPUs) when shifted from single-core to multi-core/many-core, “Gordon Moore predicted that the transistor density of semiconductor chips would double approximately every 18 to 24 months”[3], which is known as Moore’s law. The computers would not only have more transistors but also faster transistors according to Moore’s law prediction. The traditional single-core processor frequency has followed it for 40 years. This made it relatively easy to optimize the performance of the conventional programs, including the scientific computer applications. Most users relied on the expanding capabilities and speed of uniprocessors to get performance improvement. However, this frequency increase could no longer be sustained because of the following problems.

- The absolute important problem is the increasing power density, which is an unsolvable problem for classical uniprocessor designs. The quantity of transistors per chip has extraordinarily expanded lately, each of these transistors devours power and produces heat
- Memory speeds don’t scale well as processor speeds. These diverging rates intimate that a memory wall problem will happen, in which memory accesses take

over code performance. These wasted clock cycles can cancel the benefits of frequency increases in the uniprocessor [4, 5].

- Innovations in IC technology allow the hardware feature size to keep dropping. As feature size drops, interconnect delay often overrides gate delay and becomes absolute serious performance problem to be solved in future IC design and can eventually cancel the speed of transistors [6].
- Uniprocessor is designed to exploit the instruction level parallelism (ILP) in program. While exploiting ILP was the primary goal of processor designs for a long time, the higher level parallelisms, i.e., thread-level parallelism (TLP) and data-level parallelism (DLP), occurring naturally in a large number of applications cannot be exploited with the ILP model.

Due of the limits described above, the era of taking advantage of Moore's law on the conventional uniprocessor designs appeared to be arriving to an end. Since 2005, the computing industry changed path when all major manufacturers, such as INTEL, IBM, SUN, and AMD, turned to multi-core designs, where a number of simple cores are integrated on a single die. The many-core architectures are believed to be able to take advantage of Moore's law by doubling the number of cores per die with every semiconductor process generation starting with a uniprocessor. There are numerous advantages to building many-core processors through smaller and simpler cores[2, 7, 8]:

- Decreasing the frequency drops down the power consumption significantly. So, designers can provide an efficient way to achieve performance by running multiple cores with lower clock rate.

- Configure and shutdown small core is easy, which allows a finer-grained ability to control the overall power efficiency. Many-core architectures partition resources, including memory, into individual small parts, and thus attenuate the effect of the interconnect delay and reduce grappling on the shared main memory. Each core uses a cache to reduce contention on the shared main memory and increase overall performance.
- Many-core chip designers support TLP, which is expected to be exploited in future programs and multiprocessor-aware operating systems and environments.
- Design and functionally verify is easy for a core. In particular, it is more acceptable to being tested with formal verification techniques than complex architectures.
- Performance and power characteristics of smaller core are easier to predict within existing electronic design systems.

Since the multi-core has been released in commercial servers. A new trend in industry and academia is rise by incorporating larger number of cores (tens or hundreds) into a single chip. Two main types of many-core processors are released from industry community. It is characterized on how main memory consistency applied combined with the local cache of the core.

- Full Memory System Hierarchy (FMSH): Cache coherency protocols are responsible to keep the main memory coherence with the local caches of each

core. Inconsistent data problem arises, when a core in the system maintains caches of a common memory resource, . Moreover, if the main memory has a copy of a memory block from a previous read and the local cache of the cores changes that memory block, the main memory could be left with an invalid cache of memory without any notification of the change. Cache coherence is responsible to manage such conflicts and keep consistency between cache and memory. An examples like Many integrated Core (MIC) from INTEL, Tile64 from TILERA and POWER7 from IBM [9].

- Flat Memory System (FMS): The consistency problem of the main memory does not exist. Because the caches in these systems are a read only caches and they are used by write through. The programmer handles movement operation of the data from the cache to the main memory and vice versa. An example of this type is the General Processing Unit (GPU) like KEPLER 20 from NVIDIA and FIRESTREEM from ATI.

## 2.2 Challenges For Programming Many-core

A high theoretic performance provides by many-core, this increases of performance cannot be controlled as simply as what I did with single-core processors. Most of software developers were very used to the idea of getting increased performance by upgrading machines with a faster processor [10]. Unfortunately, automatic improvement will not be possible when one upgrades to a many-core processors. Although, a many-core processor can run multiple programs at the same time, it does not complete a given



program in less time, or finish a larger program in a given amount of time, without changes. The problem is that majority of the programs are written in sequential programming languages, and these programs must be maintained and optimized to exploit possible performance gains enabled by underlying hardware. For the first time, many-core architectures requires that the software developers engage in parallel computing, which was reserved for the field of supercomputing. On the other hand, this shift in the hardware roadmap poses never known challenges to the software developers. For example, the programmer will be faced with the scalability problem of expressing, coordinating and exploiting multi-level parallelism provided by the many-core machines. The programmer will also be faced with the locality challenge of optimizing data movement in a highly non-uniform memory hierarchy. Where there are gaps between data accesses to core-local memory, card global memory, and intra-node off-chip memory, and communications with remote nodes. While to exploit architectural features and eventually obtain the desired performance is the ultimate goal for programmers of this many-core machines, no majority of opinion has been reached on how to do so. On the other hand, people have been doing parallel programming development for a period of time on vector machines, clusters, SMP. Many approaches have been proposed and utilized [\[11\]](#).

### **2.3 Many-core Programming Model**

The trends go into many-core coprocessors in the community of HPC. The needs for variant programming models are appear. There are three main programming models for many-core machines. These models vary in their complexity and scalability. So,

evaluations and experiments are done to compare their performance on different architectures.

### **2.3.1 Explicit Threading**

Explicit threading model includes POSIX threads (Pthreads), Sun Solaris threads, Windows threads, and other native threading Application Programming Interfaces (APIs). It is designed to express the natural concurrency that is present in most programs, and to improve the performance. This model usually offers an extensive set of routines to provide control over threading operations, such as create, manage, synchronize threads, etc. Software developers control the application by explicitly calls these routines. On the other hand, threads have to be individually managed, this model would be the more popular choice. By committing sufficient time and exertion, program developers may be able to parallelize the problem and achieve good performance. However, because explicit threading is an inherently low-level API that mostly requires multiple steps to perform simple threading tasks, it demands massive effort from the programmer's side. Also, this model does not offer fundamentals of Object Oriented Programming (OOP) such as encapsulation or modularity. Therefore, manually managing hundreds or thousands threads definitely would be an unpleasant experience for the majority of programmers. Due to this reason, researchers have been increasingly looking for other simpler alternatives.

### **2.3.2 Message Passing Interface ( MPI )**

Message Passing Interface (MPI) is the standard and portable system designed to function on a wide type of parallel machines. The syntax and semantics are defined in the core

library routines. It is useful for a wide range of programmers writing portable message-passing programs in FORTRAN or the C programming language. There are several varies and efficient implementations of MPI, including some that are free or from many hardware manufactures. These allow the development of a parallel software industry and encouraged development of portable and scalable large parallel applications[12].

### 2.3.3 OpenMP

OpenMP, a portable programming interface for shared memory Symmetric Multiprocessors (SMP). The code starts executing by a master thread. Then it forks a specified number of threads and a task is divided among them as illustrated in Figure 2-1. The threads then run **simultaneously**. The runtime environment manages allocating threads to different core[13]. OpenMP becomes a de facto standard for writing programs for SMP machines.

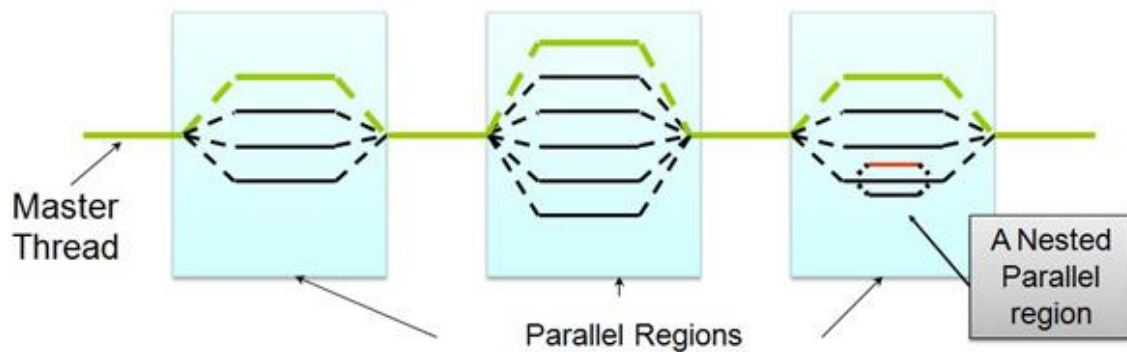


Figure 2-1 Master thread forks a team of threads as needed. Parallelism is added until a desired performance is achieved [23]

The code may contain many segments. The segment that is intended to run in parallel is marked through an OpenMP construct. This causes the compiler to parallelize it across the threads. After the execution of the parallelized code, the threads join back into the

master thread, which proceeds forward to the end of the program. By default, each thread executes the parallelized section of code independently. Parallel and work sharing constructs illustrated in Figure 2-2 can be used to divide the work among the threads. So that each thread executes its allocated part of the code. Task parallelism and data parallelism can be implemented using OpenMP.. The number of threads can be controlled by the runtime environment based on environment variables or in code using API routines.

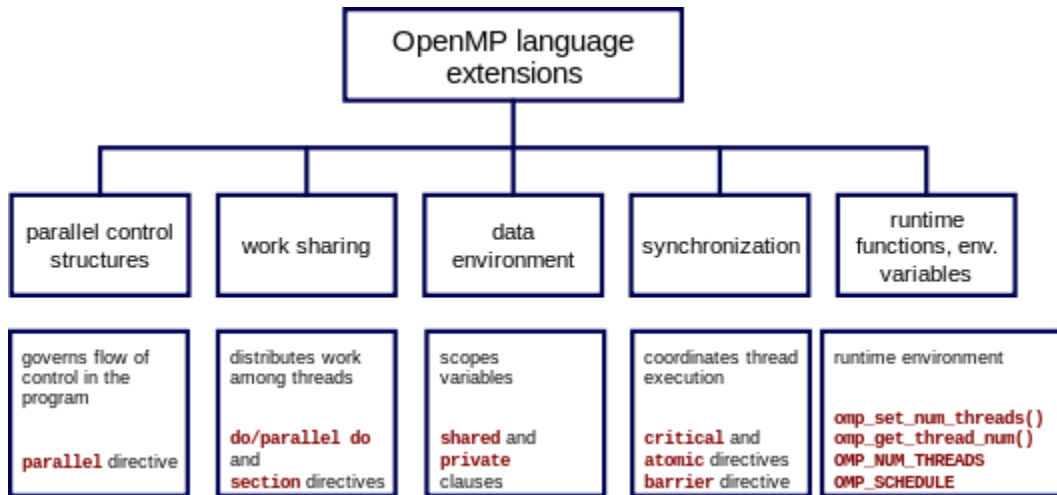


Figure 2-2 OpenMP language extension. (From [www.openmp.com](http://www.openmp.com))

Programming Many-core architectures faces significant hurdles using those models. It is Hurdles vary from one to another. One of the most difficult is addressing the programmability problems associated with code. For example, it is notoriously difficult to debug a parallel application, given the potential interleaving of the various threads of control in that application. Explicate threading allows the programmer explicitly to manage and control each thread. But, it will be tough to manage hundreds or even thousands of threads.

In OpenMP model the compiler takes the job to handle task assignment to the threads or team of threads. It is more productive environment. However, there is no guarantee to get high optimized code and difficult to address parallel programming problems like false sharing and data racing. Experiments show that application needs to be optimized using OpenMP.

## **2.4 Related work**

The demand of developing parallel applications increased. Since the hardware manufacture increases number of cores in the same die. This gives new challenges in scientific application to gain a speed up from the new machine. MIC launched at the end of 2012.

In [\[14\]](#) an early performance evaluation of the coprocessor. A focus on OpenMP programming paradigm and compared the coprocessor with another machine called BCS, which contains (16 socket, 128 core, Intel Xeon (NAHLAM), 64 GB). They did two main experiments. The first one to be evaluated memory bandwidth and openMP basic constructs overheads (parallel for, barrier, reduction) using STREAM benchmark and EPCC micro benchmark. The second one was for real compute scientific problem called Conjugate Gradient (CG) which was dependent basically on Sparse Matrix Vector Multiplication (SMXV). For the memory bandwidth they ran the experiments on the two machines with different affinity types. They found that the coprocessor has better performance of the BCS compared to 1 board with 8 processors. Also they computed the

overhead of the OpenMP constructs for native and offload on the coprocessor and the BCS. They found that the overhead for all the OpenMP constructs for the coprocessor was lower than the BCS which was a key performance issue for the coprocessor. Which means applications scale on large machines will scale on the coprocessor. Also, no significant difference between native and offload overhead on the coprocessor found.

For the real application kernel they apply roofline model to estimate the maximum performance in GFLOPS/s for each system. The roofline model is a visual graph that shows realistic expectations of performance and productivity of the multi-core and many core systems depending on its operational intensity for the algorithm. After that they run the experiments and compare their result to the approximation of the Roofline model. They achieve results closely near expectation of the model and better than the MKL library from Intel.

An optimized Geometric Multigrid (GM) problem for important multicore and many-core architecture have been done[15]. Conventional methods for solving linear system iteratively such as JACOBI, successive relaxation and Gause Seidle Red Black (GSRB) were applied. A compact multigrid solver benchmark that creates a global 3D domain partitioned into sub domains sized to proxy found in real MG application was constructed. Also, developing many optimization techniques on the KNC. GSRB is explicitly prefetched, because the compiler fails for prefetching this complex memory access pattern. To maximize performance of in-cache computations, SIMD intrinsic were applied to the GSRB kernel. They compute as if doing JACOBI and use masked stores to selectively update red or black values in memory. Moreover, large (2MB) TLB pages were used, and the starting address of each array was padded to avoid a deluge of conflict

misses when multithreaded cores perform variable coefficient stencils within near power-of-two grids. Similarly, a certain dimension was padded to a multiple of 64 bytes. To minimize the number of communicating neighbors, the KNC implementation leverages the shift algorithm in which communication proceeds in three phases corresponding to communication in  $i$ ,  $j$ , and  $k$ , where in each phase, sub domains only communicate with their two neighbors. These techniques were applied on the MIC using native mode and OpenMP programming paradigm.

A study on molecular dynamic and its performance on the CPU-MIC system is carried out [16, 17]. A development of three thread level parallelism schema is done. Task-level parallelism between CPU and MIC using offloads techniques, thread-level parallelism across multiple MIC cores, and data-level parallelism within each MIC core to exploit the SIMD unit effectively. Also, an applied memory latency hiding and prefetching techniques is done. To evaluate the proposed approach a comparison between CPU-MIC and CPU-GPU system is constructed. A gain of speedup 2.25 on the CPU-MIC system compared to CPU-GPU system. In addition, an evaluation of different machines with different SIMD width (128 bit, 256 bit, 512 bit) is carried out. Several optimization techniques applied on the SIMD using hand on analysis to exploit the unit. However, the experiments show that the compiler can't handle prefetching schema to gain full vectorization of the code automatically. For evaluation, the Sandia's miniMD benchmark is chosen, which is an MPI ranked implementation. Also three optimization techniques especially for the KNC include problem decomposition, PCIe bandwidth latency and code reuse.

MIC can also be used for image processing applications that need hundreds of TFLOPS like in [18]. A Synthetic Aperture Radar (SAR) via back projection has been studied. It is an image reconstruction mechanism used in the real applications of radar system and can be extended to other applications like medical imaging. The importance of this work by using an algorithmic optimization for mathematical operation such as square root, sine and cosine by converting them into a few multiplications and additions. The optimization resembles strength reduction, a well-known compiler optimization. While significant reduction relies on, and is thus constrained to, mathematical equivalence, their optimization exploits the method of approximation. Therefore, call of this model approximate strength reduction (ASR). Also, exploit hardware gather support of MIC for incessant eccentric memory accesses, thus improving the vectorization efficiency. The efficiency of gather access is further improved by exploiting geometric properties of back projection used in SAR imaging. data transfer is pipelined to hide latency. The Parallel resources in recent computation platforms must be exploited. The computation is carefully partitioned between Intel Xeon and MIC so that the benefits of MIC's high compute intensity can be maximized. Data movement is optimized for locality and vectorization, which is provided by architecture support for irregular memory access. The dimension space is divided into three levels: MPI, OpenMP and cache Blocking. The experiment results show that the application of ASR to the back projection stage achieves 2–4x speedups, while maintaining a similar level of accuracy. But, in the gather hardware optimizations they gain 1.4x speedup.

A hybrid evaluation is done in [19] using MPI-OpenMP programming paradigm. A demonstration of conservative spectral method for the Boltzmann equation originally



developed by Gamba and Tharkabhushaman, to parallelize the application. The iteration space is divided into the number of threads equally size to neglect the load balancing issue and it is applied directly using the OpenMP directive without any optimization techniques. For evaluation, MIC is used (Stepmede TACCs) and AMD OPTERON architecture. The experiments done using both OpenMP and MPI-OpenMP approaches . The results present a linear speedup when increasing number of cores.

Sparse Matrix with dense Vector (SpMV) has been studies in [20]. Before evaluating the kernel, an investigation to the new hardware capabilities using micro benchmark. Compute the read-bandwidth by sum kernel of large matrices. After that write bandwidth is evaluated. This helps in determining how to deal with the hardware and what is the limitation of this hardware for real application. after exploration of the machine compiled the kernel with different optimization options from the compiler options such as no compiler optimization -O0 , compiler optimization level -O2, -O3, vectorization and without vectorization. One of the interesting experiments is study the useful cache line density, which is a metric derived for the analysis. For each row, compute the ratio of the number of nonzero on that row to the number of elements in the cache lines of the input vector due to that row. The results outline that this is a performance metric comparing running application without vectorization and with vectorization options.

Colfax tests the MIC coprocessor with a basic N-body simulation, which is basic for a set of applications in computational astrophysics and biophysics. An implementation for non-optimized version of the problem is done, and it is run on the coprocessor natively. Experiments show that they get benefits from it directly. After that, digging into optimized code and try to understand what the bottleneck in the code is. A profiling using

VTune was done. The result shows that the most time is consumed from the Short Vector Math Library (SVML) provided by Intel. The analysis shows that the library by default supports denormal numbers accuracy. Turn this option off leads to better performance. Also, looking at the assembly language produced from the C code using the VTune profiler. They analyze if the compiler do its job correctly and optimize the code efficiently. Additionally, they found that the programmer must take care with the types and precision of variables, constants and functions.

C++ Parallel library construct is created in[21], which makes it easy to insert a function to every member of an array in parallel and dynamically distributing the work between the host CPUs and one or more coprocessor cards. A description of the associated runtime support and use a physical simulation called smoothed particle hydrodynamic example to demonstrate the library construct can be used to quickly create a C++ application that will significantly benefit from hybrid execution, simultaneously exploiting CPU cores and coprocessor cores. Experimental results show that one optimized source code is sufficient to make the host and the coprocessors run efficiently.

The work shows a new way of application development that has been made possible by the MIC coprocessors. The OFFLOAD\_FOR\_EACH function template allows the developers to quickly build new applications that target the architecture. Also, this gives the developer the ability to create one source code and efficiently using MIC architecture. From software engineer point of view this makes it easy for the programmer to debug, troubleshoot and optimize the application on this new machine.

Even most already parallel algorithms need some adaptation to run effectively on parallel architectures. Some research optimizes common parallel programming primitives on the different architecture and programming languages. The needs to identify the new parallel architecture become crucial.

## CHAPTER 3

### THE XEON PHI MANY INTEGRATED CORE

#### 3.1 Introduction

New architectures have evolved to satisfy the needs of compute power. Accelerators, such as Graphical Processing Units (GPUs) and Many Integrated Core (MIC) are two ways to fulfill the requirements [22]. MIC coprocessors offers all standard programming models that are available for Intel Architecture: OpenMP, POSIX threads or MPI [23]. The MIC coprocessor plugs into a standard PCIe slot and provides a standard shared memory architecture. For programmers of higher level programming languages like C/C++ or FORTRAN using well established parallelization paradigms like OpenMP, Threading Building Blocks (TBB) or Message Passing Interface (MPI), the coprocessor appears like a symmetric multiprocessor (SMP) on a single chip. Compared to accelerators this reduces the programming effort a lot, since no additional parallelization paradigm like CUDA or OpenCL needs to be applied [24]. However, supporting shared memory applications with only minimal changes does not necessarily mean that these applications perform as expected on MIC. In this chapter I will describe the most important features relevant to better understand the optimization techniques that will be applied in the next chapters. Most of the information based on information found in the Intel 64 and IA-32 Architectures Optimization Reference Manual, the Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual [25], Intel C++ compiler XE

13.1 user and reference guide [26] and Intel Xeon Phi Coprocessor System Software Developer's Guide [27].

### 3.2 Many Integrated Core Architecture

MIC coprocessor platform is based on the concepts of the Intel Architecture and that provides standard shared-memory architecture. Figure 3-1 shows the high level architecture of the MIC coprocessor die. It has more than 50 cores (this may varies depending on the version of the coprocessor and manufacture), offers full cache coherency across all cores. The cores connected by a high performance two ways directional ring interconnect ring. In addition, there are 8 memory controllers supporting up to 16 GDDR5 expected to deliver up to 5.5 GT/s. Each memory controller supports two channels per memory controller. This provides a theoretical bandwidth up to 352 GB/s delivered to the coprocessor.

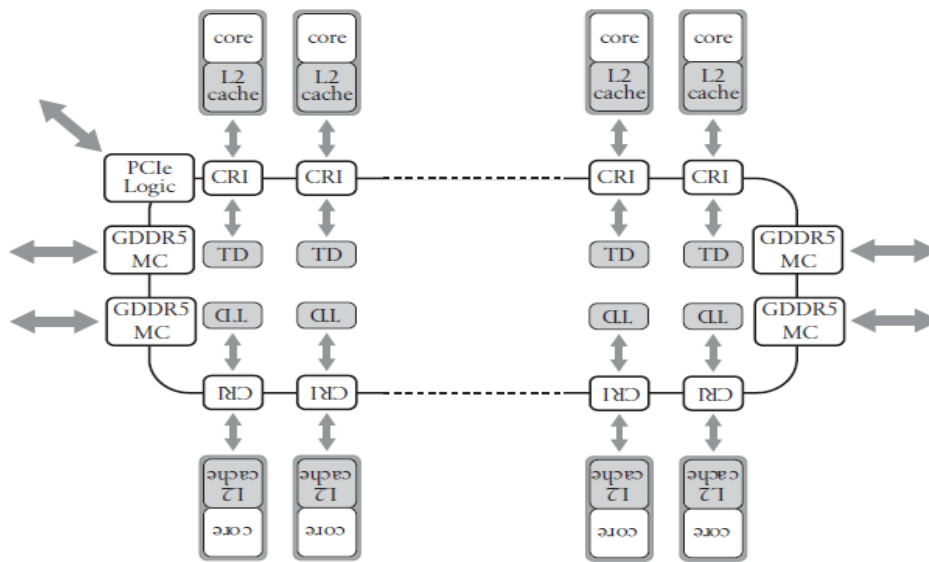


Figure 3-1 General layout of the MIC coprocessor. For simplicity, only 8 of the total cores and 4 of the total 8 GDDR5 memory controller shown [27]

The two ways directional ring has three types of rings in each direction. A data block ring (64 bytes wide), an address ring (send/write commands and memory addresses) and an acknowledgment ring (flow control and coherency messages). There are a set of tag directories connected to the ring and mapping address to the tag directories is based on hash functions over memory addresses, leading to an equal distribution around the ring. The memory controllers also connected to the ring, providing access to the GDDR5 memory.

Figure 3-2 illustrates basic building block of the coprocessor. At the right it shows the GBoxes memory controller that access external memory for read and writes. Every controller has 2 channels with 32 bit wide bus.

The GBoxes contains three types; interfaces to the ring interconnect (FBOX), request scheduler(MBOX) and the physical layer that interfaces with the GDDR devices (PBOX). The MBOX contains two CMCs (or Channel Memory Controllers) that are completely independent from each other. The MBOX provides the connection between agents in the system and the DRAM I/O block. It is connected to the PBOX and to the FBOX. Each CMC operates independently from the other CMCs in the system. At the left, it shows the SBOX controller, it is generation 2 PCI express client logic. This is the system interface to the host machine support x8, x16 PCI configuration. Also, at the left there is the DBox controller which is a debug display engine. At the middle of the Figure it shows how the basic block of each core and how it is connected to each other. In each core there is a Core Ring Interface (CRI) which includes interface to the core and ring interconnect, the L2 cache, the tag directory (TD), and asynchronous processor interrupt controller (APIC) which receives interrupts to redirect the core for response.

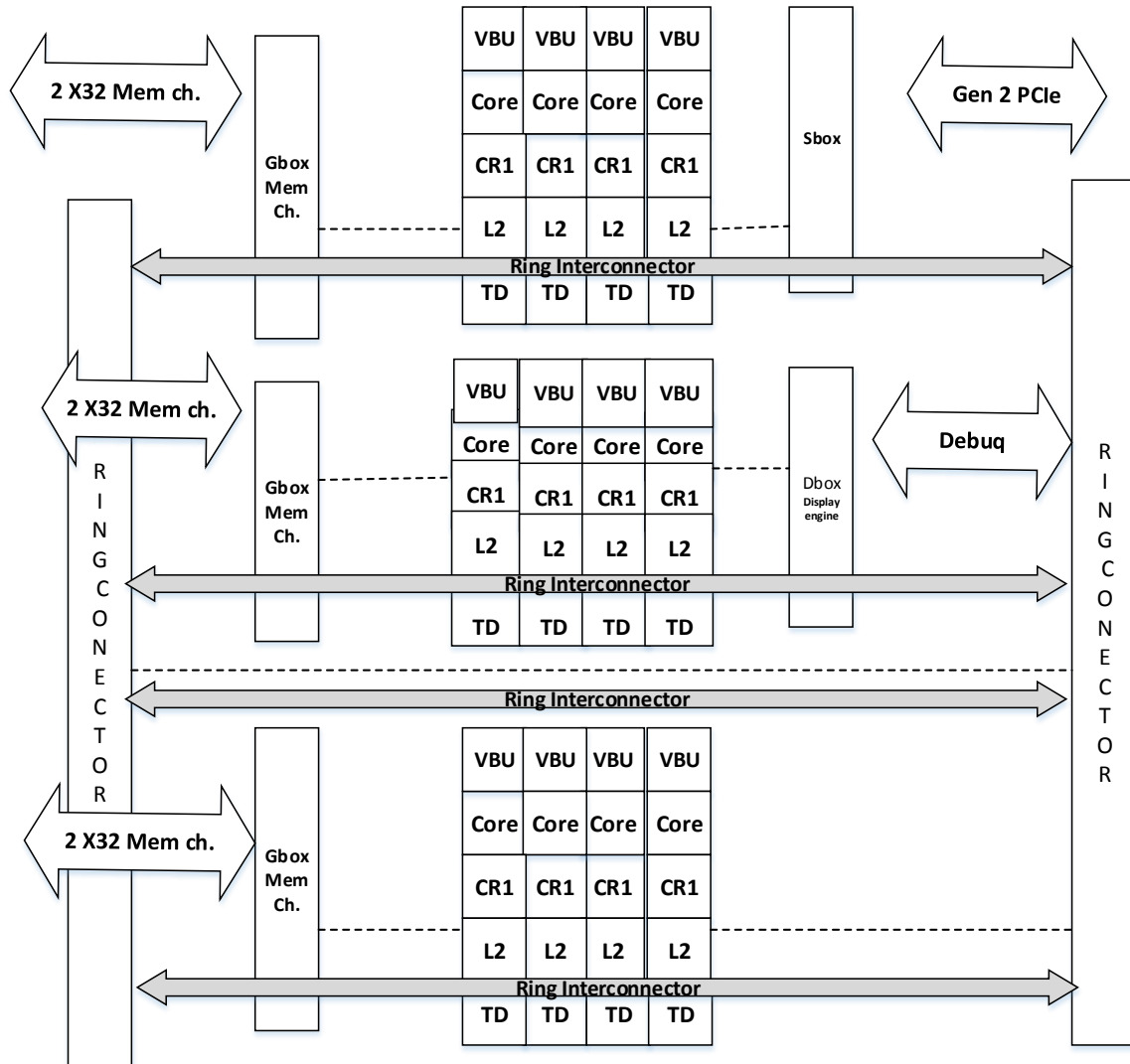


Figure 3-2 Basic building block of the MIC

Figure 3-3 shows the high level architecture of the MIC core. Every core offers four-way simultaneous multi-threading (SMT) and 512-bit wide Single Instruction Multiple Data (SIMD ) Vector Processing Unit (VPU), which corresponds to eight double-precision (DP) or sixteen single precision (SP) floating point numbers. Each core in the architecture has a 32kB L1 data cache, a 32kB L1 instruction cache, and a 512kB L2 cache that cumulatively produce shared cache among the cores. The architecture of a core is based on the pentium architecture, but the design has been updated to 64 bit architecture. Each

core includes two basic units scalar units (SU) at the left and the Vector Processing Unit at the right (VPU). Both of them share the same instruction decode unit in each core but every unit has its own register type. Also the local private L1 cache data and instruction cache is shared between the two units. In addition, L2 cache with the other cores in the same coprocessor through the CRI is shared.

MIC has the ability to execute Intel Instruction set Architecture (ISA) in addition to the MIC ISA. It is a 5 stages dual pipeline; the main pipeline U-pipe and the V-pipe. The core can execute 2 instructions per clock cycle like Pentium one on the U-pipe and the other on the V-pipe. The U-pipe executes any instruction include the vector instructions. But the V-pipe can't execute all instructions types [28].

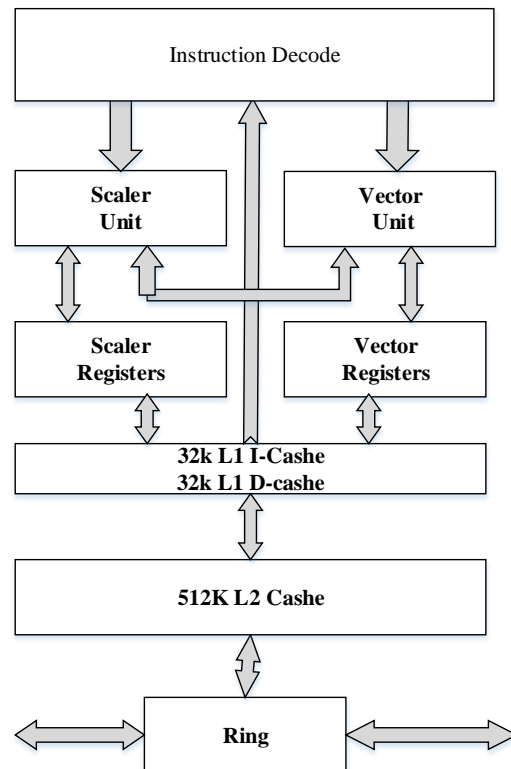


Figure 3-3 Core architecture for MIC



One of the differences over the Pentium is the 64 instruction extensions. Integer register files, data paths, and major busses were widened from 32 bits to 64 bits. Integer registers were increased from 8 to 16. Changes were made to the instruction decoder to decode new opcodes. A four-level page table and RIP-relative addressing have been added. Extending in new directions from the 64-bit enhanced base line. The cores have hardware multithreading support that can reduce the impact of latencies to keep the execution units busy. Each 64-bit in-order short pipeline core supports four hardware threads.

Figure 3-4 shows pipeline data path of the core [25]. At any given clock cycle, the two instructions each core can issue from any single context can be: 1 vector operation using the pipe0 and 1 scalar operation using pipe 1 (or prefetch operation to load data to the cache before processing it ), 1 vector operation and 1 (special) vector operation, or 2 scalar operations. Another component is the VPU associated with each core. This is primarily a sixteen-element wide SIMD engine, operating on 512-bit vector registers with Fused Multiply Add (FMA).

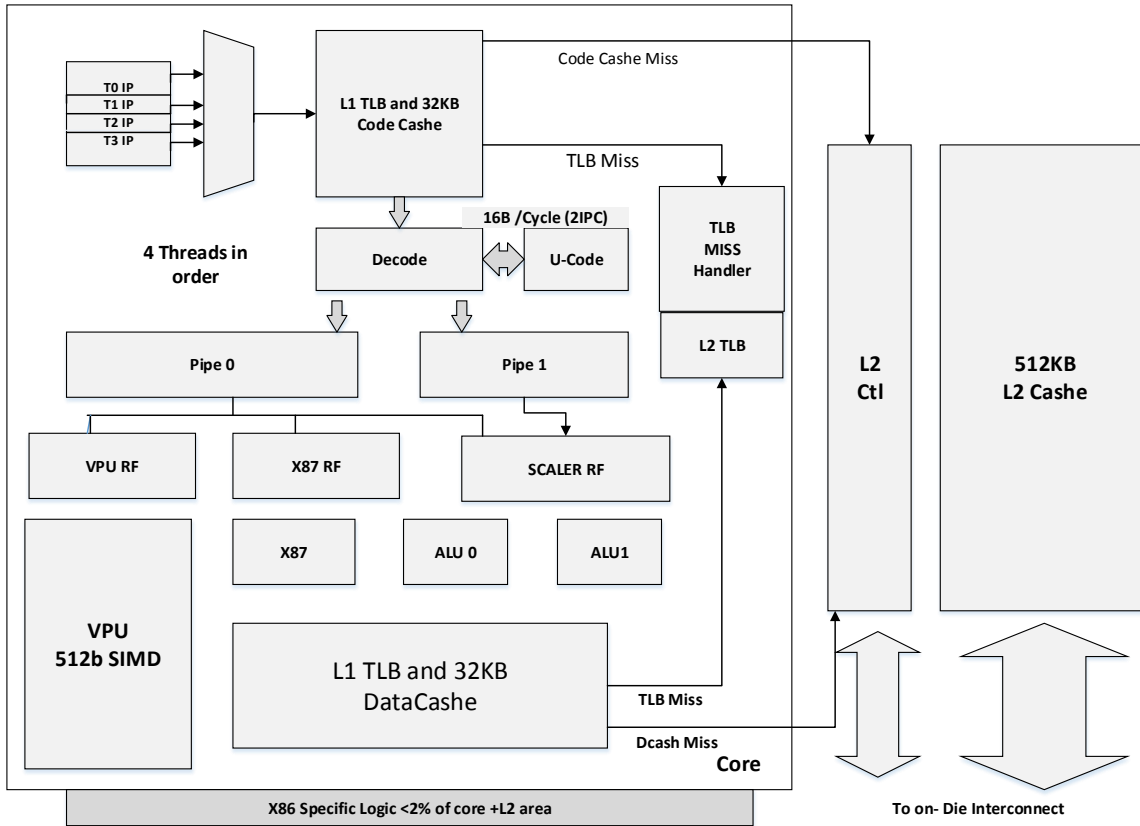


Figure 3-4 MIC pipeline data path

Due to these vectorization capabilities and the large number of cores, the coprocessor can deliver 1 TFLOPS of DP theoretical performance. VPU supporting a new instruction set called Intel Initial Many-Core Instructions (Intel IMCI) [23]. The Intel IMCI includes, among other instructions, the fused multiply-add, reciprocal, square root, power and exponent operations, commonly used in physical modeling and statistical analysis.

### 3.3 Cache Structure and Coherency Protocols

MIC has two levels of caches in each core. The Level One (L1) cache has 32 KB L1 instruction cache and 32 KB L1 data cache. Associativity is 8-way, with a 64 byte cache line. Bank width is 8 bytes. Data return can be out-of order. The access time has 3-cycle

latency. The level two (L2) unified cache has 64 bytes per way with 8-way associativity, 1024 sets, 2 banks, 32GB (35 bits) of cacheable address range and a raw latency of 11 clocks. The expected access time is approximately 80 cycles [25]. The L2 cache has a streaming hardware pre fetcher that can selectively pre fetch code, read, and Read-For-Ownership (RFO) cache lines into the L2 cache. There are 16 streams that can bring in up to a 4-KB page of data. Once a stream direction is detected, the pre fetcher can issue up to 4 multiple pre fetch requests. The replacement policy for both the L1 and L2 caches is based on a pseudo Least Recently Used (LRU) algorithm.

The L2 cache is attached to the core-ring interface block. This block also includes the Tag Directory (TD), and the Ring Stop (RS) which connects to the inter-processor core network. Within these sub-blocks are the transaction protocol engine which is an interface to the RS and is equivalent to a front side bus unit. The RS handles all traffic coming on and off the ring. The TDs, which are physically distributed, filter and forward requests to appropriate agents on the ring. It is also, responsible for starting communications with the GDDR5 memory through the on-die memory controllers.

To keep caches coherent through the cores, the coprocessor implements variations of cache coherency protocols include MESI, Extended MESI and Globally Owned Locally Shared (GOLS) [29].

MIC cache coherency protocol is a directory protocol based on MESI that uses GOLS to simulate an owned state, thus allowing the share of a modified line. Table 3-1 and Table 3-2 illustrate MESI extended protocol state and definition of each state. The aim is to avoid write backs to memory when another core wants to read a modified cache line.

So, the shared state of this protocol does not mean that the line has not been modified. Each core's cache uses the MESI state of the lines that it contains and the Distributed Tag Directories (DTDs) will contain the global GOLS coherency state of each line. Lines are assigned to each DTD regarding the line address instead of the core that is containing or requesting the line.

**Table 3-1 Extended MESI cache coherency protocol states [29]**

Cache State		State definition	State definition related to memory
<b>M</b>	Modified	Only this core owns the line(dirty)	It has been modified regarding memory
<b>E</b>	Exclusive	Only this core owns the line(clean)	It has not been modified regarding memory
<b>S</b>	Shared	Several cores can have the line	It may or may not have been modified regarding memory
<b>I</b>	Invalid	The core does not own the line	It has not been used

When a cache miss occur the core will request the line to the correspondent DTD. This DTD will answer depending on the GOLS state of the line and will request memory or the core which owns the line to answer with the data. If another core has the line, it will notify the DTD and send the data to the requester core, which will also notify to the DTD that it has received the data. Then, the DTD will update the line state. Any eviction will also have to request the DTD for allowance before effectively evicting the line.

**Table 3-2 Global owned locally shared cache coherency protocol state [29]**

Cache State		State definition	State definition related to memory
<b>GOLS</b>	Globally Owned Locally Shared	Several cores can have the line	It have been modified regarding memory
<b>GE/GM</b>	Globally Exclusive/Modified	Only this core owns the line	It may or may not have been modified regarding memory (the core will have the line in M or E)
<b>GS</b>	Globally Shared	Several cores can have the line	It has not been modified regarding memory
<b>GI</b>	Globally Invalid	No core holds the line	It has not been used

### 3.4 Software Stack Architecture View

The MIC coprocessor software architecture is outlined in Figure 3-5. There are essentially four layers in the software stack appear at the right of the figure: tool runtimes, user-level offload libraries, a low-level communication layer that's split between user-level libraries and kernel drivers (Comms), and the operating system. There is a host-side and co-processor-side component for each. Everything below the offload runtimes is part of the Intel Many Core Platform Software Stack (MPSS).

The software stack of MIC which is shown on the left-hand side is based on a modified Linux kernel. The operating system on the MIC coprocessor is in fact an embedded Linux environment which called micro OS ( $\mu$ OS). It provides basic functionality such as

process creation, scheduling, or memory management. Multiple options are available for communication between the host and the card. The card driver provides virtual network interfaces, so it is possible to use the TCP/IP network stack. This is for management and compatibility with existing applications. On the other hand, it cannot provide maximum performance, since the network stack was designed for a different purpose than communication over PCI Express.

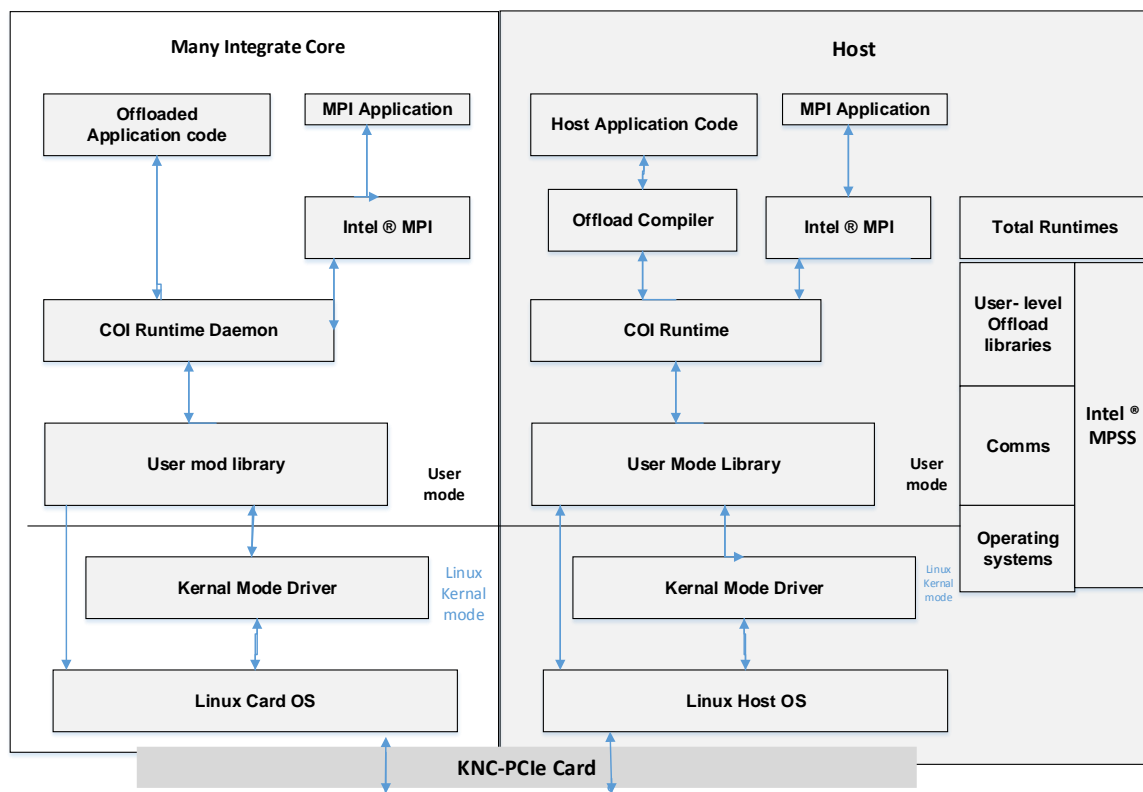


Figure 3-5 MIC Software stack Architecture

### 3.5 Programming Model

MIC supports the majority of compute paradigm for what is available right now. [24]. There are two main approaches; the “Offload” in this case the program is viewed as running on host and offloading select work to the coprocessor. In “Native” approach the

program runs natively on coprocessors which may communicate with other MIC card or programs by various methods.

Several execution models can be derived from these two approaches. It depends on the programming paradigm. Figure 3-6 shows it in case of MPI. Figure 3-6 (a) shows offload model where the communication of the processes take place between the hosts' processors. But, the coprocessor capabilities used through the offload library between the host and coprocessor in each subsystem. On the other hand, Figure 3-6 (b) shows the coprocessor only model (Native) where the communication between the processes is done between the coprocessors that is in different subsystem. Also, Figure 3-6 (c) shows the third model where the execution of the MPI process and the related MPI communications and Message passing is supported inside the coprocessor, inside the host node and between the coprocessor. In this case I can assume MPI nodes inside the coprocessor itself and apply communications between them.

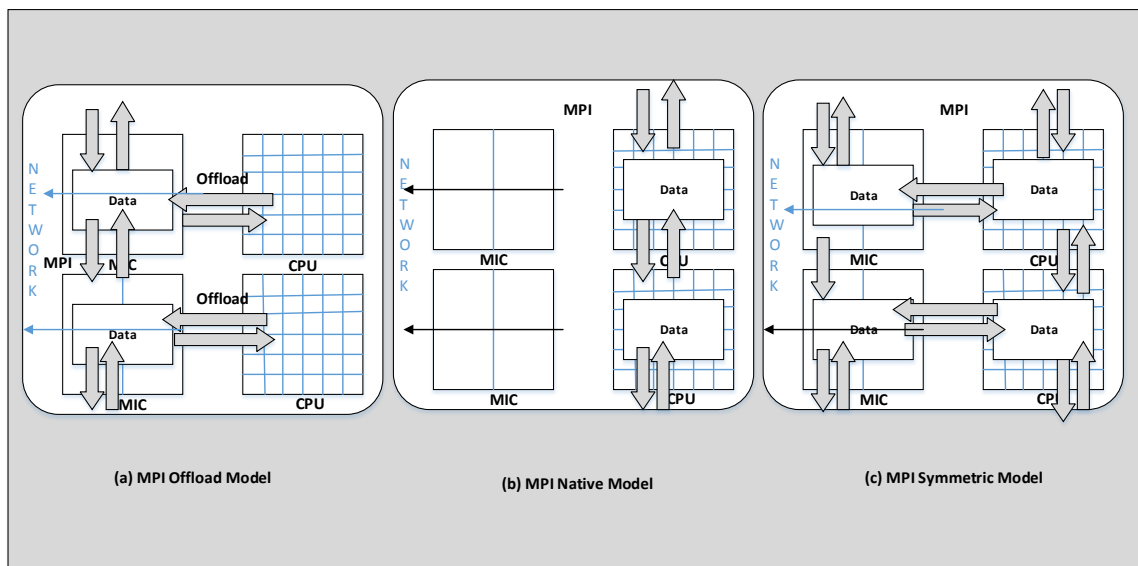


Figure 3-6 MIC Programming Model

Other case is using OpenMP programming paradigm. It is a fork-join model supported through two execution models. However, the “Native” model where the program runs in the coprocessor only and all the work done on the coprocessor. The other one is the offload model, where the program runs in host and offloads (send) compute intensive code by associated data to the device as specified by the programmer via pragmas in the source code. Also, it supports hybrid model of execution using MPI and OpenMP inside the coprocessor or between different coprocessors. This is useful in cluster environment.

### **3.6 Thread Execution Model**

MIC utilizes hyper-threading (HT) or simultaneous multithreading (SMT) on each core as a key to masking the latencies inherent in an in-order micro architecture. This should not be conflicted with hyper threading on Xeon processors that exists primarily to more fully feed a dynamic execution engine [30]. In HPC workloads, very often hyper-threading may be ignored or even turned off without degrading effects on performance.

MIC offers four hardware threads per core with sufficient hardware components, floating-point hardware components and memory capabilities. Figure 3-7 at the left shows 4 in order threads numbered (T0, T1, T2, T3) that is scheduled at a multiplexer. The four threads used for hiding the latencies and keep the two main units (Scalar, VPU) in each core busy. From the figure there are 3 stages for handling a thread. It starts from the Previous Picker Function (PPF) that selects the thread from the L1 instruction cache. It includes for each thread context branch target prediction, branch recovery address, and the next segmentation of Instruction Pointer (IP) and old latched address for the specific thread context. After that, there is a prefetching buffer in the next stage of the picker



function. It includes the instructions that is ready to be picked from the thread picker buffer and to send to the third stage which is the decode stage. The thread picker issues 2 instructions in every cycle from the same thread context. However, as I described before every core can executes in each cycle 2 instructions (U-pipe, V-pipe).

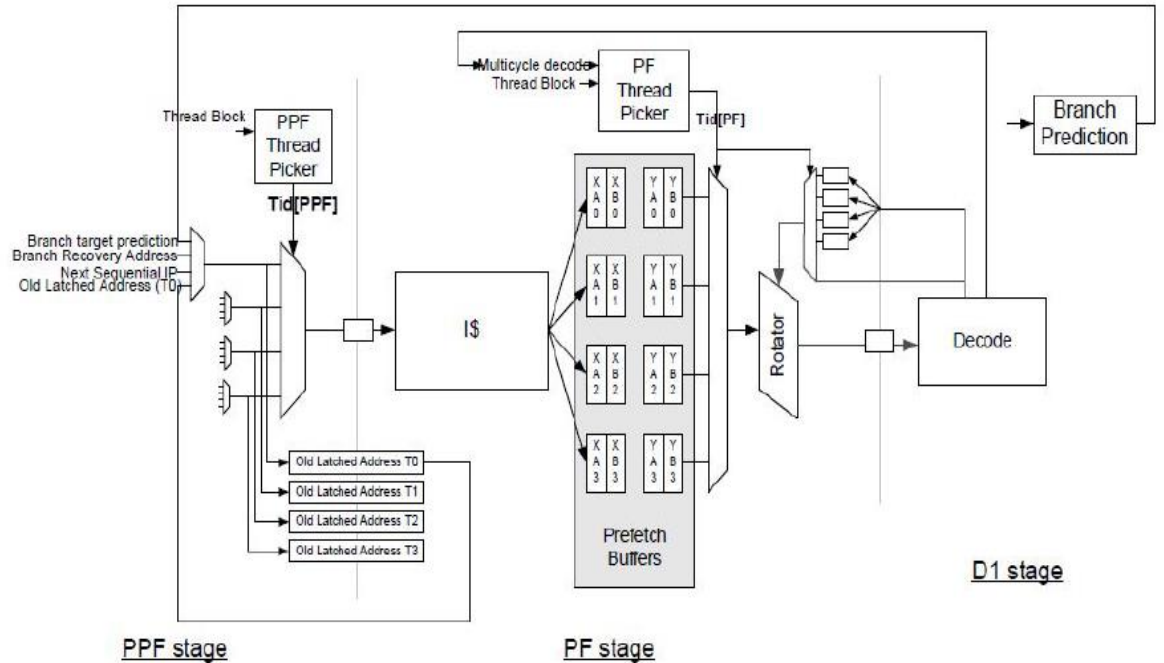


Figure 3-7 Multi-threading Architectural Support in MIC Core [27]

The PF works in a round-robin style, issuing instructions during any one clock cycle from the same thread context only. For example, in cycle  $N$ , if the PF issues instruction(s) from Context 3, then in cycle  $N + 1$  the PF will try to issue instructions from Context 0, Context 1, or Context 2 – in that order. Hence, it is not possible to issue instructions from the same context (Context 3 in this example) in back-to-back cycles.

In one hand, this makes it generally impossible for a single thread per core to approach either limit. In general, applications need a minimum of three or four active threads per

core to access all of the resources offer. For this reason, the number of threads per core utilized should be a tunable variable in an application and be set based on experimental experience of the application. On the other hand, the theoretical flop rate presupposes that the workload can be decomposed into fused multiply accumulates. This will be true only in some very unusual situations, or in the innermost loop of some applications. But it's always good to remember that the total wall time of a real application depends always on many other lines of code that may not be able to being re factored in this way.

All four hardware threads per core share their local L2 cache but have high speed access to the caches associated with other cores. Any data used by a specific core will reserve space in that local L2 cache, and also it can be in multiple L2 caches around the chip. While MIC has a penalty for “cross-socket” sharing, which occurs after about 16 threads , It has a lower penalty across more than 200 threads. There is a benefit to having locality first organized around the threads being used on a core (up to 4) first, and then around all the threads across the coprocessor. So I can conclude that the way the threads spread across the cores will affect the performance. However, the thread consumes the data from L1 cache in case of miss if the data available in the cache L2 for another thread it will be delivered to it without needs to deliver from the memory. This is can be done by setting the `KMP_AFFINITY` variable for the compilation time when using OpenMP or `I_MPI_PIN_DOMAIN` with MPI.

MIC appears as conventional Shared Memory Multiprocessing (SMP). To keep memory consistence among all of the cores the system implements directory cache coherency protocol across the cores. This maintains the shared variable consistence in all of the cores.

The core will only be stalled when a load miss occurred. When a load miss occurred, the hardware context with the instruction triggering the miss will be suspended until the data are brought into the cache for processing. This allows the other hardware contexts in the core to continue execution. Both the L1 and L2 caches can also support up to about 38 outstanding requests per core (combined read and write). The system agent (containing the PCI Express agent and the DMA controller) may also generate 128 outstanding requests (read and write) for a total of  $(38 * \# \text{ of cores} + 128)$ . This allows software to prefetching data aggressively and avoids triggering a dependent stall condition in the cache. When all possible access routes to the cache are in use, new requests may cause a core stall until a slot becomes available.

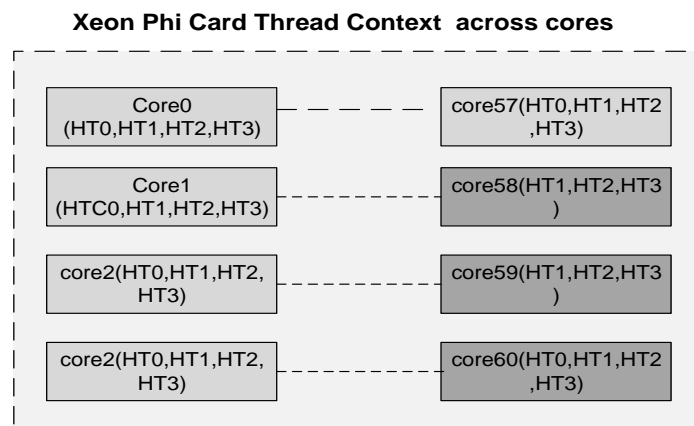
MIC doesn't support paging to an external device. It has only one DMA engine, so any communications (network file-system, MPI, sockets, ssh, and so forth) between the coprocessor and host can interfere with offload data transfers and affecting on the application performance.

### **3.6.1 Thread Affinity**

The Intel runtime library and Coprocessor OS; has the ability to bind OpenMP thread contexts to physical processing unit [\[26\]](#). The interface is controlled using the `KMP_AFFINITY` environment variable. It restricts execution of certain thread context to a subset of the physical processing units in a multiprocessor computer. Depending upon the topology of the machine, thread affinity can have a dramatic effect on the execution speed of a program.

There are three levels for interfaces the affinity to the processing unit (high level, mid-level, low level). The first level assigned implicitly by the operating system according to specific affinity type (scatter, balanced, compact) by setting an operating system variable. The second one is explicitly done by the programmer for a specific core id by setting a variable in the OS environment. The third one, the programmer can use API to assign thread explicitly to the core.

Figure 3-8 shows how the threads are spread across the cores. The logical processor number used by the coprocessor OS on MIC architecture is different from that on the host. There is one logical processor for each hardware thread context. Logical processor 0 is placed on the first hardware context of the highest numbered core. Logical processors from 1 up to the highest, minus three, are placed consecutively on core 0 context 0, core 0 context 1, core 0 context 2, core 0 context 3, core 1, context 0 and so on, with the last three logical processors being on the highest numbered core, with hardware thread contexts 1, 2, and 3.



**Figure 3-8 MIC Card Thread Context across 60 cores**

### 3.6.2 Thread Affinity Type

MIC has three types of thread affinity. These types describe how the thread context is bind to the hardware thread along the cores; and in which order. Figure 3-9 shows how scatter affinity distribute 6 threads across 3 cores; The thread context are placed across the cores; until all cores have at least one thread, after that add the others in round robin fashion. Figure 3-10 shows how to assign 6 threads context to the hardware threads cores using the compact Affinity. It assigns the thread contexts to the hardware contexts by filling the core with 4 threads one at a time. The last one is balanced affinity. Figure 3-11 shows assigning 9 threads in using balanced type. It is only available on the MIC coprocessor; in this type threads placed on separate cores until all cores have at least one thread, similar to the scatter type. However, when the runtime must use multiple hardware thread contexts on the same core, the balanced type ensures that the thread numbers are close to each other, which scatter does not do.

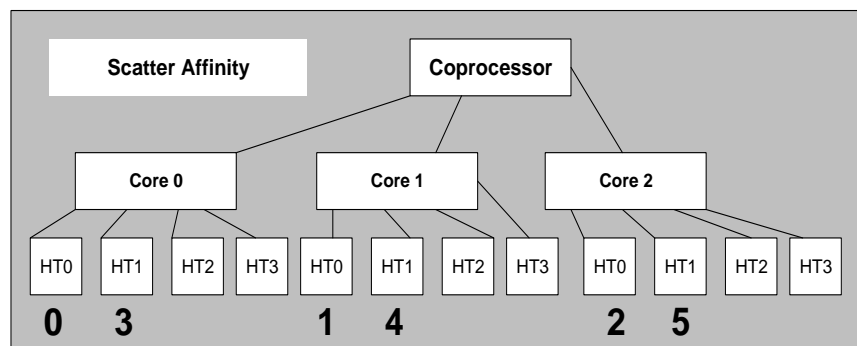


Figure 3-9 Scatter Affinity for 6 thread

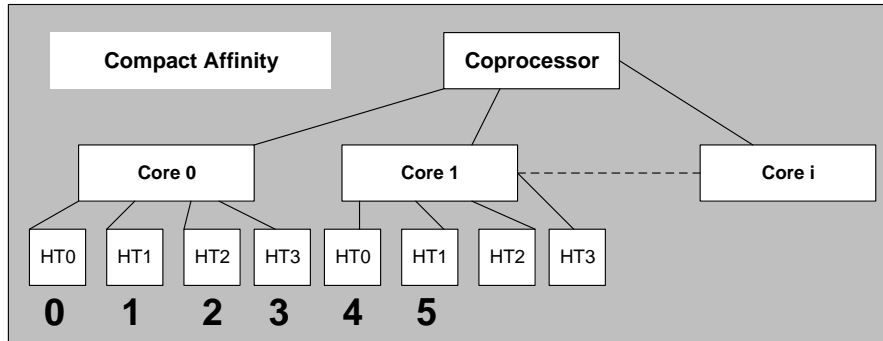


Figure 3-10 Compact Affinity for 6 threads

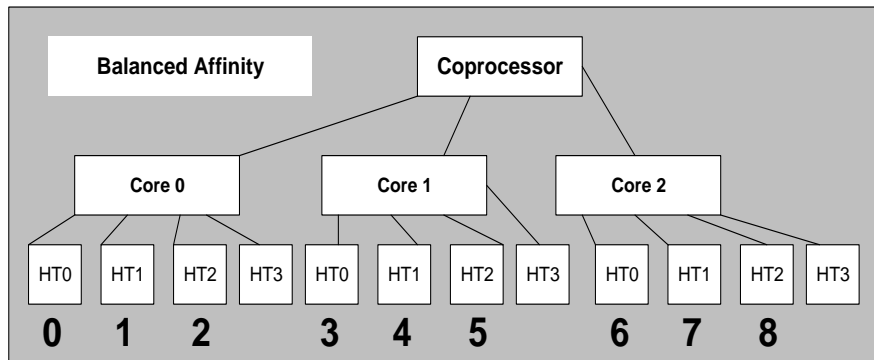


Figure 3-11 Balanced Affinity for 9 threads

From analytical point of view, it is normally beneficial to use cores before threads, so the compact affinity type is unlikely to yield the best results, because it leaves cores unused. The thread allocation under scatter is likely to be better than compact, because it uses cores before threads. However, scatter allocates threads such that threads with IDs in close numerical proximity are on different cores, and therefore do not share caches. Because threads with neighboring IDs often operate on closely related data, placing them on different cores is unlikely to be the best way to allocate them. The thread allocation under balanced is balanced over the cores and the threads allocated to a core are neighbors of each other. Therefore, cache utilization should be efficient if the threads access data that is near in store.

## CHAPTER 4

### STATIC PROBLEMS

#### 4.1 Introduction

In this chapter, I present my work for static problems. I choose from two categories Basic Linear Algebra and numerical algorithms. For first category, I used the Strassen Matrix Multiplication (SMM). Matrix-Matrix multiplication (MM) is massively parallel application with fixed data layout. The basic Strassen-MM (S-MM) algorithm having time complexity of  $O(N^{2.807})$  instead of  $O(N^3)$  of standard MM algorithm. My optimization is based on a reordering approach to reduce the storage, use of a depth first walk (DFW), and invocation of the Math Kernel Library (MKL) optimized library from Intel for smaller matrix-matrix multiplications. In DFW, all available machine parallelism is used in each depth expansion.

For the second category I used Jacobi Solver (JS) of a system of linear equations for which the load is static across the iterations. In iterative JS, the threads need to read a vector that was computed by all the working threads before starting the next iteration. It is noticed that due to the above data layout JS does not scale well because of the excessive synchronization overhead, which must be implemented across all the working threads. To improve JS scalability, I explored (1) Synchronous Jacobi (SJ), (2) Asynchronous Jacobi (AJ), and Relaxed Jacobi (RJ). In SJ I used explicate barrier synchronization. In AJ a non-exact solution is computed because completing threads start

the next iterations using current data, which is a mixing of new and old. AJ slows down the convergence rate. In RJ, completing threads at iteration K start the next iteration (k+1) using newly computed data. RJ provides overlap between two iterations at the cost of managing the availability of currently available intermediate results.

## 4.2 Matrix Multiplication

Matrix-matrix multiplication (MM) is a cornerstone of linear algebra algorithms; when multiplying matrices, the elements of the rows in the first matrix are multiplied with corresponding columns in the second matrix. I will use MM ( $C=A \times B$ ) where the size of the problem is  $N \times N$ . Figure 4-1 Naive Matrix multiplication outlined naïve matrix multiplication.

```
for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)
    {
        sum=0; // S1
        for (k=0; k<N; k++)
        {
            sum+=A[i][k]*B[k][j]; // S2
        }
        C[i][j]=sum; // S3
    }
}
```

Figure 4-1 Naive Matrix multiplication code

MM nested loop is a loop independent dependency (LID) since there is no data access between different iteration space and dependency occur only in the same iteration space for S2.



In MM, assumes both  $A$  and  $B$  are stored in row major order. traditional methods of matrix multiplication are not cache-friendly. In MM  $C = AxB$ , elements in matrix  $A$  are accessed in row major, but elements in Matrix  $B$  are accessed in column major order. Each access to  $B$  results in a cache miss since the consecutively accessed elements are not contiguously stored in memory. Elements of  $B$  are repeatedly accessed when computing different elements of  $C$ , but they do not remain in the cache for reuse as the cache capacity is small. Besides, only small portions of the fetched cache blocks are accessed before they get replaced due to conflicts. The net result is a large number of cache misses. The entire computation of MM involves  $2N^3$  arithmetic operations (counting additions and multiplications separately), but produces and consumes only  $3N^2$  data elements. As a whole, the computation shows honorable reuse of data. In general, an entire matrix will not fit in the cache. The work must therefore be broken into small chunks of computation, each of which uses a small enough piece of the data. In standard MM I can compute the number of references to the memory as the following equation.

$$\text{Memory reference} = N^3(\text{read each column of } B, N \text{ times}) + N^2(\text{read each row of } A \text{ once}) + 2N^2(\text{read and write each element of } C \text{ once}) = 3N^2 + N^3$$

If I compare it to access of elements required from the memory ( $3N^2$ ). So I can notice that a lot of overhead and miss reuse. As a conclusion, in this implementation, the algorithm needs to be optimized to get better performance and scalability.

#### 4.2.1 Execution Model for MM

Naïve matrix multiplication has no dependency as I describe later, the loop iterations can be executed independently of each other. So parallelizing the naïve code is

straightforward. Insert the `#pragma omp` for before the outermost loop (i loop). It is beneficial to insert the `pragma` at the outermost loop, since this gives the most performance gain. In the parallelized loop, variables  $A$ ,  $B$ ,  $C$  and  $N$  are shared among the threads, while variables  $i$ ,  $j$ , and  $k$  are private to each thread.

An early experiments are done, to increase locality and reusing of the cache. They are tiling and blocking. A comparison of the results have been done with highly optimized library MKL. It shows that MKL outperforms them on MIC.

To optimize execution time of MM on MIC STRASSEN algorithm is applied. I have developed a number of implementations. The optimized version is presented in the next section.

#### 4.2.2 Strassen MM (S-MM)

Volker Strassen published the Strassen algorithm in 1969 [31] based on a divide and conquer strategy. Let  $A$ ,  $B$  be two square matrices over a ring  $R$ . The objective is to calculate the matrix product  $C$  as follows:

$$C=AB \quad A,B,C \in R^{2^n \times 2^n}$$

If the matrices  $A$ ,  $B$  are not of type  $2^n \times 2^n$ , the missing rows and columns will be filled with zeros.  $A$ ,  $B$ , and  $C$  will be partitioned into equally sized block matrices such that

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

with

$$A_{i,j}, B_{i,j}, C_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$$

then

$$\begin{aligned} C_{1,1} &= A_{1,1} B_{1,1} + A_{1,2} B_{2,1} \\ C_{1,2} &= A_{1,1} B_{1,2} + A_{1,2} B_{2,2} \\ C_{2,1} &= A_{2,1} B_{1,1} + A_{2,2} B_{2,1} \\ C_{2,2} &= A_{2,1} B_{1,2} + A_{2,2} B_{2,2} \end{aligned}$$

In the above construction still 8 multiplications are needed to calculate  $C_{i,j}$  matrices. In order to reduce the number of multiplications, the following new matrices have to be defined.

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

Now, using only the above 7 multiplications,  $C_{i,j}$  can be express in terms  $M_k$  as follows:

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ C_{1,2} &= M_3 + M_5 \\ C_{2,1} &= M_2 + M_4 \\ C_{2,2} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

This matrices partition process can be done recursively until the sub matrices degenerate into numbers. Figure 4-2 represents level 1 and level 2 of STRASSEN algorithm that goes into Level N.

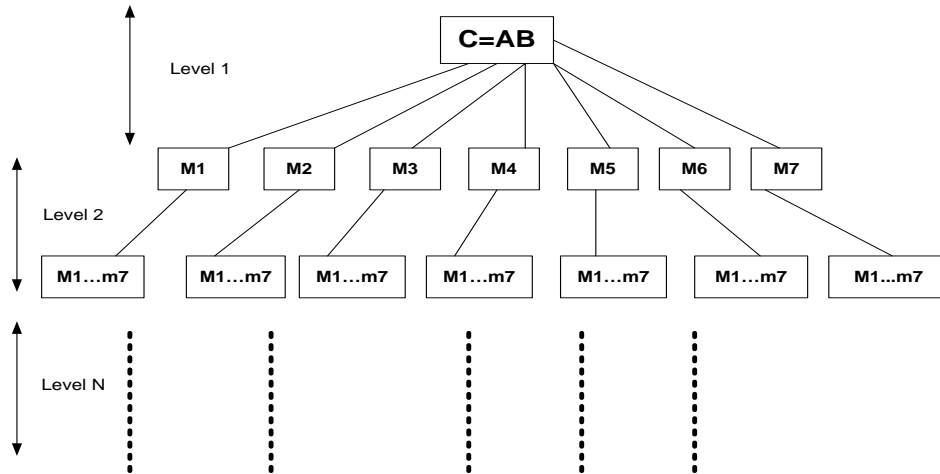


Figure 4-2 STRASSEN MM recursion into N level

The complexity of S-MM algorithm in terms of arithmetic operations (additions and multiplications) can be expressed as follows:

$$f(n) = 7f(n-1) + 14^n$$

where  $f(n)$  denotes the number of additions performed at each level  $l$  of the algorithm.

$$g(n) = (7 + O(1))^n$$

where  $g(n)$  denotes the number of multiplications performed at each level.

Thus, the asymptotic complexity for multiplying matrices of size  $N = 2^n$  using the STRASSEN algorithm is  $O([7 + o(1)]^n) = O(N^{\log_2 7 + o(1)}) \approx O(N^{2.8074})$ . The reduction in the number of operations however comes at the price of a somewhat reduced numerical stability, and the algorithm also requires significantly more memory compared to the standard algorithm. Both initial matrices must have their dimensions expanded to the next power of 2, which results in storing up to four times as many elements, and the seven auxiliary matrices each contain a quarter of the elements in the expanded ones. The arithmetic complexity of the algorithm is:

$$t_m(n) = n^{\log_2 7} \quad t_a(n) = 6n^{\log_2 7} - 6n^2$$

Where  $t_m(n)$  and  $t_a(n)$  respectively denote the number of multiplications and the number of additions. The execution model of STRASSEN can be summarized for recursive implementation as follows:

- Divide matrix  $C$  into  $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$
- Compute matrix  $M1, M2, \dots, M7$ .
- Compute matrices  $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$
- Any multiplication I check if the size of matrices greater than a threshold value , call the previous steps again recursively.
- If matrix size less than the threshold call normal MM.

#### 4.2.2.1 Implementation on MIC

Optimization of parallel applications under new many-core architectures is challenging even for regular applications. However, in modern architectures the arithmetic operations take approximately the same number of cycles. Therefore, the performance of strassen comes from the lower complexity of addition operation compared to MM multiplication. Therefore, successful strategies inherited from previous generations of parallel or serial architectures just return incremental gains in performance and further optimization and tuning are required [32].

The Original implementation of S-MM suffers from memory usage [33], and it is not practical due to the size of memory that it needs for huge matrices. I have implemented a reorder algorithm of S-MM to reserve memory allocation[34] [35].

In this implementation two intermediate matrices (T1,T2) have been reserved in each level of recursion, of size  $(N/2L)$ . Where L is the level of recursion and N is dimension of the matrix  $N \times N$ .

My first experiment shows that the original implementation is unpractical on Intel Xeon Phi with 5.6 Gbyte of memory. I can achieve matrix size up to 3072 with 5 level of recursion. But in reorder implementations I achieve matrix size up to 10240. So, I will focus in the next sections on the reorder Implementation.

Figure 4-3 shows pseudo code of the implementation. It has three main operations addition, subtraction and multiplication. The algorithm called recursively into L level of recursion depending on the threshold value. For each of the operations mentioned you need to pass the new size of the matrix and the indices of sub matrices. Because in each level the size of matrix t is changed. Figure 4-4 shows how to pass the indices to each sub matrix. I have 4 sub matrices as shown. Each operation of the algorithm called with three matrices as operand applies the operation using the first two matrices and store the result in the third matrix operand. The small  $x$  denotes to which matrix I use.

CBLAS\_DGEMM from MKL has been used as the engine of the multiplication operation. It is a highly optimized library from Intel. However, experiments done on optimization of matrix multiplication using tiling and blocking have shown shows that CBLAS\_DGEMM outperforms them.

```

void strassenMultMatrix(double *x, double *y, double *z,
int size,int srow1 , int scoll1,int srow2,int scoll2,int srow3,int
scoll3,int DIM0,int DIM1,int DIM2
){
    double **t1, **t2;
    int newsize = size/2;
    if (size >= threshold) {
        t1 = (double*)
malloc(sizeof(double*)*newsize*newsize);
        t2 = (double*)
malloc(sizeof(double*)*newsize*newsize);
        addMatrices(a11,a22,t1, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        addMatrices(b11,b22,t2, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        strassenMultMatrix(t1,t2,c21, int size,int srow1 ,
int scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2); // Compute M1
        subMatrices(a21,a11,t1, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        addMatrices(b11,b12,t2, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        strassenMultMatrix(t1,t2,c22, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2); // Compute M6
        subMatrices(a12,a22,t1, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        addMatrices(b21,b22,t2, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        strassenMultMatrix(t1,t2,c11, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2); // Compute M7
        addMatrices(c11,c21,c11, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        addMatrices(c21,c22,c22, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        addMatrices(a21,a22,t1, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2);
        strassenMultMatrix(t1,b11,c21, int size,int srow1 , int
scoll1,int srow2,int scoll2,int srow3,int scoll3,int DIM0,int
DIM1,int DIM2); //Compute M2
        subMatrices(b12,b22,t2, int size,int srow1 , int

```

```

scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
strassenMultMatrix(a11,t2,c12, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2) // Compute M3
subMatrices(c22,c21,c22, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
    addMatrices(c22,c12,c22, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
subMatrices(b21,b11,t2, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
strassenMultMatrix(a22,t2,t1, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2); // Compute M4
addMatrices(c11,t1,c11, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
addMatrices(c21,t1,c21, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
addMatrices(a11,a12,t1, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
strassenMultMatrix(t1,b22,t2, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);subMatrices(c11,t2,c11,newsiz);
addMatrices(c12,t2,c12, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
    }
    else {
        normalMultMatrix(a,b,c, int size,int srow1 , int
scoll,int srow2,int scol2,int srow3,int scol3,int DIM0,int
DIM1,int DIM2);
    }
}
}

```

**Figure 4-3 Pseudo code of Reorder Implementation**

Sub matrix	Row index	Column Index
Sub <sub>1,1</sub>	Newsiz	Newsiz
Sub <sub>1,2</sub>	Newsiz	newsiz+scol <sub>x</sub>
Sub <sub>2,1</sub>	newsiz+srow <sub>x</sub>	Newsiz
Sub <sub>2,2</sub>	newsiz+srow <sub>x</sub>	newsiz+scol <sub>x</sub>

**Figure 4-4 Strassen, Submatrix indices**



### **4.2.2.2 Experiment Results**

I implemented several versions of S-MM. Including Single and double precision to evaluate error lower bound, upper bound and average error. Also the matrix has been implemented using two dimensional Arrays and one dimensional array. My result shows that on MIC the best implementation is using one dimensional array. Because *MALLOC* with two dimensional arrays didn't allocate the memory consecutively and this affects the performance on MIC. I also compute the execution time with 5 levels of recursion and compare it to a highly optimized library CBLAS\_DGEMM from MKL.

#### **4.2.2.2.1 Scalability**

To evaluate S-MM scalability on MIC; the execution time is reported. I compared the results with highly optimized matrix-matrix multiplication library developed by Intel which is named MKL. In addition, the speed up is computed using different matrix size and different number of cores; where each core was assigned 4 threads using compact affinity.

To Test the scalability, I ran the experiments with different matrix sizes and different number of threads. All the experiments were done by disabling the factorization unit and using the O2 level of optimization of the Intel compiler 2013. Also the Affinity is set to compact to increase the locality and sharing between the threads. In my experiments each core has 4 OpenMP threads. They are bind to the hardware threads depending on the OS scheduling criteria. Figure 4-5, Figure 4-6, Figure 4-7 Figure 4-8 show the experiments using different number of cores.

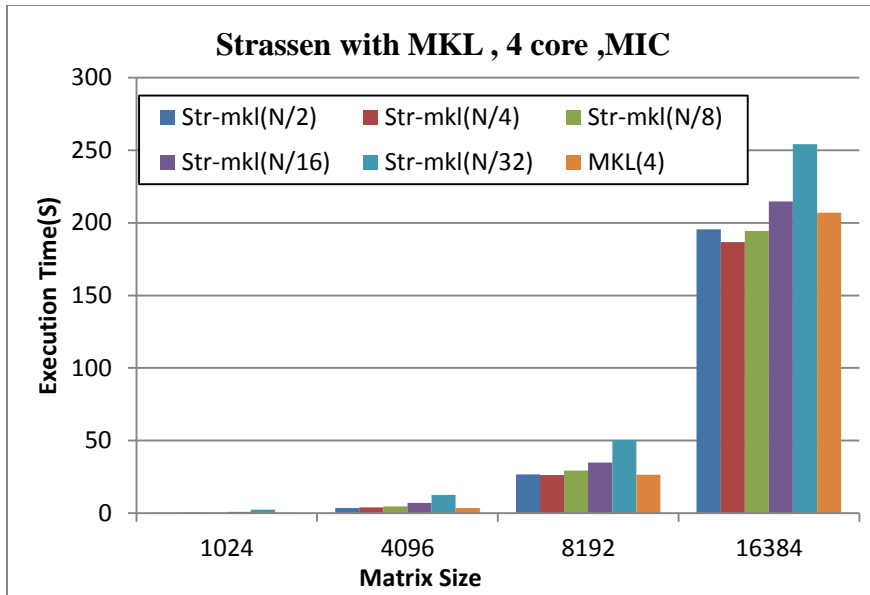


Figure 4-5 Strassen with MKL, 4 core, MIC

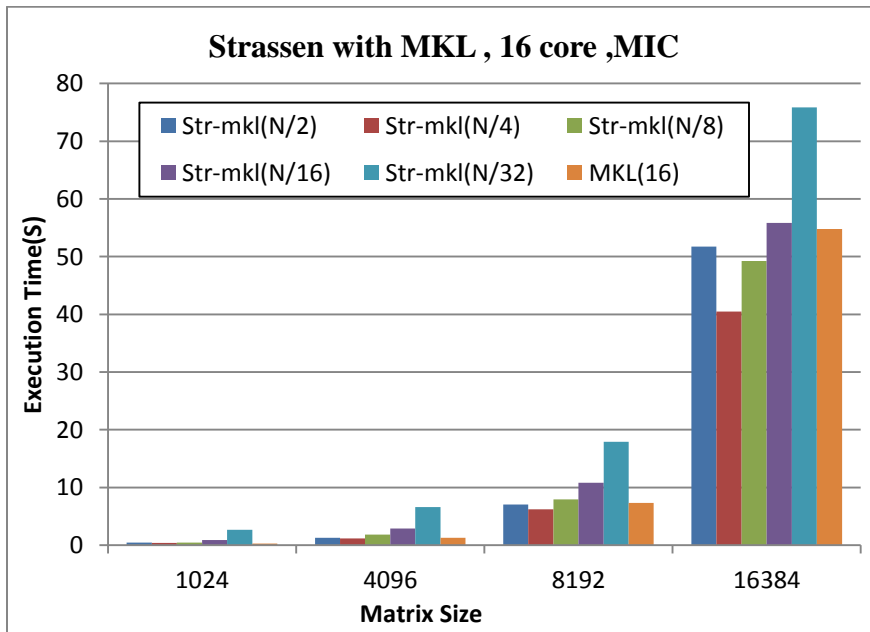


Figure 4-6 STRASSEN with MKL, 16 core, MIC

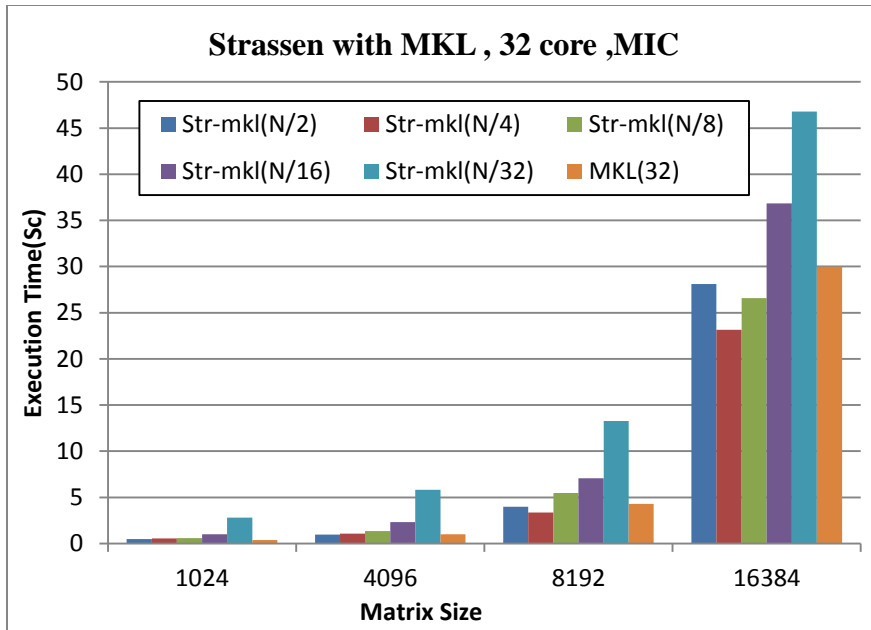


Figure 4-7 STRASSEN with MKL, 32 core, MIC

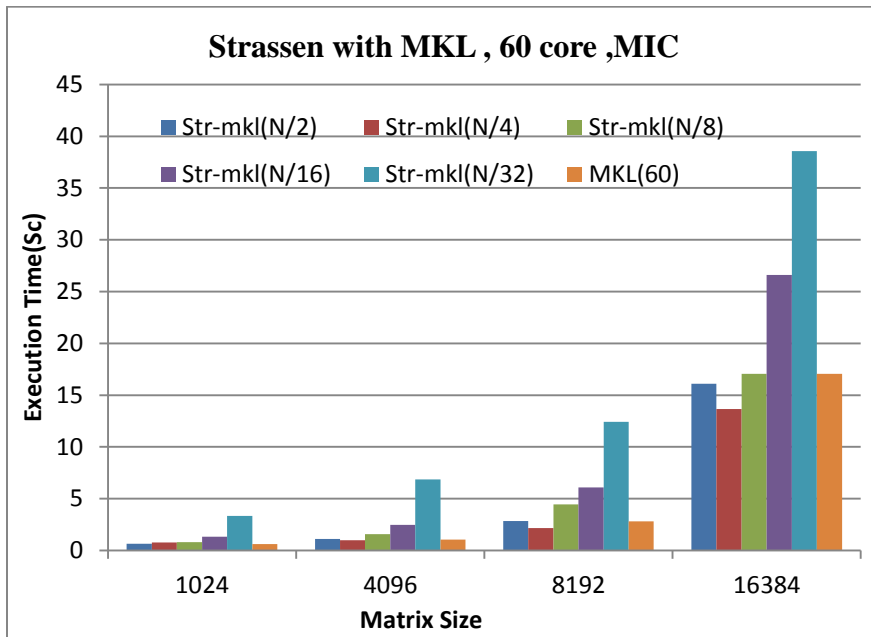


Figure 4-8 Strassen with MKL, 60 core, MIC

From the figures (Str-mkl) is the execution time of my strassen using CBLAS\_DGEMM as MM for smaller matrix size along different recursion level. Also, the results of MKL have been shown. Experiments show that increasing the level of recursion up to level two decreases the execution time using large matrix size. I obtain an execution time better than CBLAS\_DGEMM (MKL) 8% to 24% on matrix size 8192, 16384 respectively, when the number of cores greater than 32 core. I can conclude from the results that the number of cores used is an important factor combined with the size of the matrix and level of recursion.

Further interesting results can be obtained from the speed up. Figures below depict the same data, but as a speedup relative to one core.

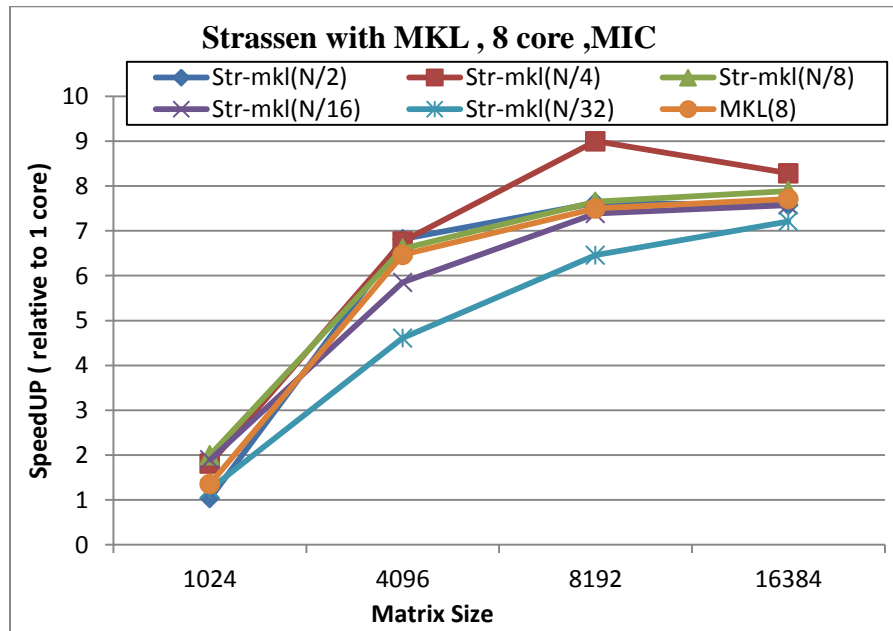


Figure 4-9 Speed Up of Strassen relative to 1 core, 8 core

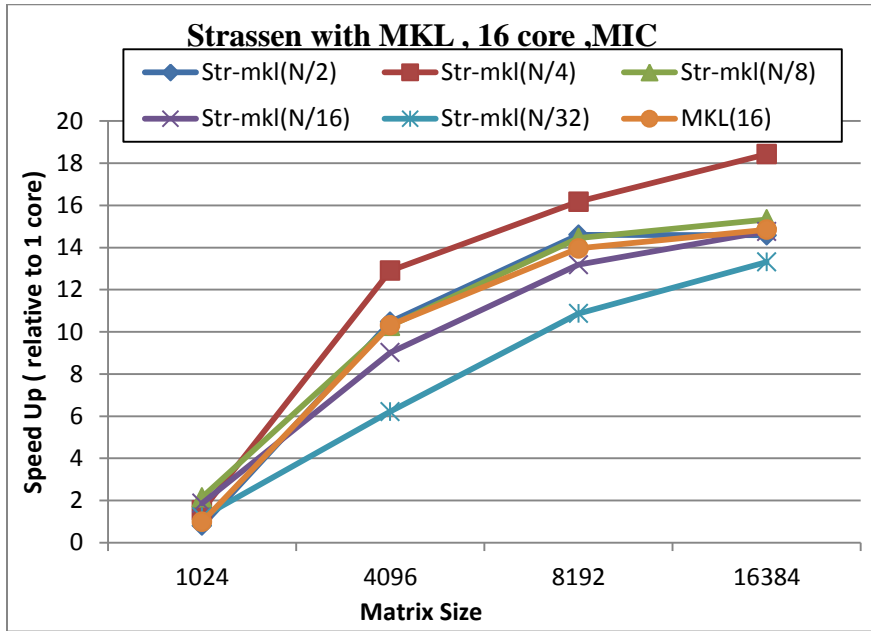


Figure 4-10 Speed Up of Strassen relative to 1 core, 16 core

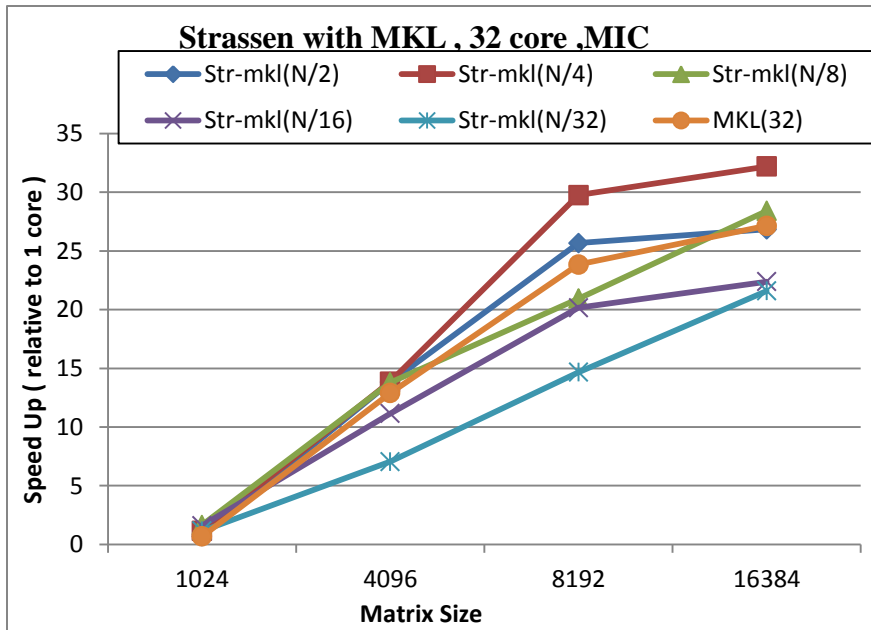


Figure 4-11 Speed Up of Strassen relative to 1 core, 32 core

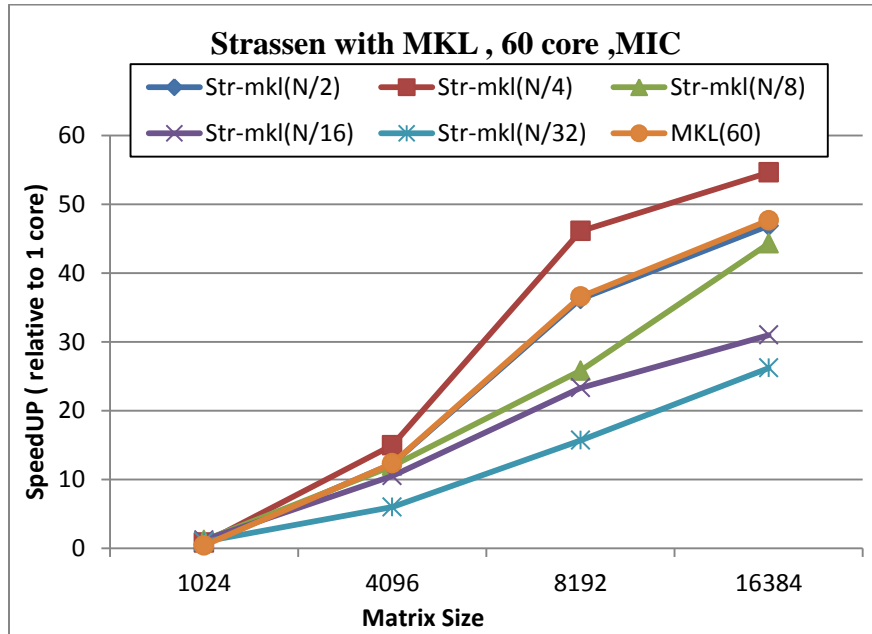


Figure 4-12 Speed Up of Strassen relative to 1 core, 60 core

It is clear to see that with small matrix sizes (1024 , 4096), the speed up does not scale well when the number of core increased. But, for larger matrix size the speedup increased linearly when the number of cores was increased. I got a speed up equals to the number of cores for matrix size 8192 and 16384.

In summary, experiments show that increasing the level of recursion cause increased of execution time. The cause of this poor performance of the machine at the level of recursion refers to the following points:

- When I used maximum number of threads, this increases the time of scheduling and managing threads in Linux. But this had low percentage of impact on the performance.

- STRASSEN algorithm uses divide and conquer algorithm and the size of the sub matrices decreases when I increase the level of recursion. So utilization of caches in the cores decreasing due to that the smaller number of matrices when goes into deeper recursion level. This depends on the size of the matrix and number of cores used. These two factors can be used as a collaboration factors for optimization.
- CBLAS\_DGEMM library time increases when the size of the matrix smaller than 2048 and increasing the number of core. Figure 4-13 shows the CBLAS\_DGEMM function from MKL performs with smaller size of matrix and larger number of Cores

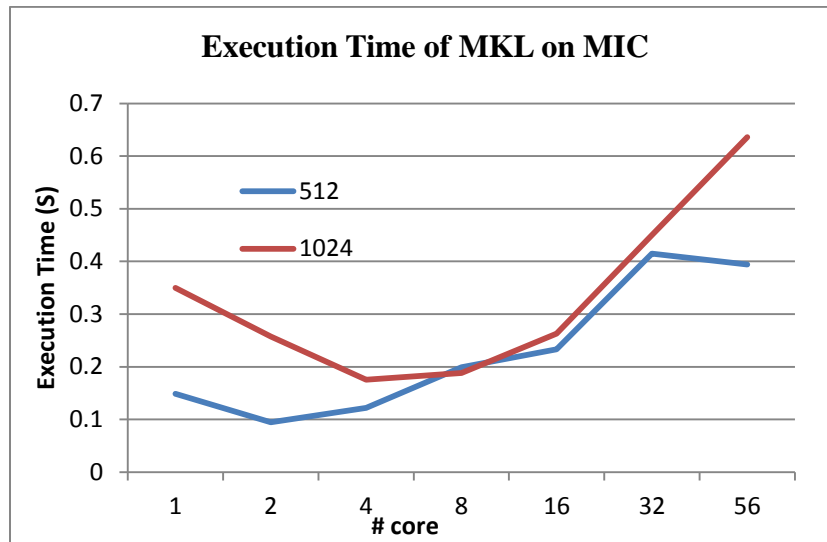


Figure 4-13 Execution time of MKL on smaller matrix size and different number Cores

### 4.2.3 Conclusion

I have implemented the basic S-MM algorithm having time complexity of  $O(N^{2.807})$  instead of  $O(N^3)$  of standard matrix multiplication algorithm following the same execution steps as proposed by Strassen. In addition, I have also implemented a reordered approach for Strassen to reduce memory allocations.

But increasing the recursion level in my implementations will increase the execution time due to the overhead of intermediate additions operations. I have also shown that this increase in execution time with the level of recursion will be reduced with the large space size. So, my implementations will be more profitable as the size of the matrices is increased. On MIC, the results show that the use of up to 2 recursion levels for S-MM with MKL as the basic MM library outperforms MKL alone by 8 to 24%. This shows the profitability of S-MM procedure with a few recursion levels to tune performance of optimized MM libraries.

### 4.3 Jacobi Solving Linear Equations

JACOBI is an iterative method used to solve a Linear System Equation  $AX=B$  with number of equations equal  $N$ . It start with an initial solution  $X^0$  and computes the  $X^{k+1}$  for  $k$  times of iteration. Any iteration  $k$  needs all the values of  $X$  from iteration  $k-1$  except the values of  $x_i$ . Also it needs the value of  $B$  and  $A$  which is constant. The equation of  $x_i$  can be written as the following:

$$a_{i1}x_1 + \dots + a_{ii}x_i + \dots + a_{iN}x_n = b_i \quad \text{for } i=1,2,3,\dots,N$$
$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1, j \neq i}^N a_{ij}x_j^{(k)} \right)$$



The process is initialized with a solution equal  $X^0$  and iterate until the value of  $X^{k+1}$  is converged or until specific value of  $K$  depends on the accuracy and ability of the matrices to converge. Figure 4-14 below shows sequential code of Jacobi iterative method.

```

for(k = 0; k < MAX_ITER; k++)
{
    for(i=0; i<N; i++){
        sum = 0.0;
        sum=sum-A[i*N+i] * X_seq[i]; // S1
        for(j=0; j<N; j++){
            sum += A[i*N+j] * X_seq[j]; // S2
        }
        new_x[i] = (B[i] - sum)/A[i*N+i]; //S3
    }
    for(i=0; i < N; i++)
    X_seq[i] = new_x[i]; // S4
}

```

**Figure 4-14 JACOBI sequential code implementation**

I conclude from the sequential code that there is a dependency distant 0 between (S2, S3) and (S3, S4). Also, there is a forward loop carried dependency (F-LCD) with dependency distance =1 between S2, S4.

For Jacobi I have matrix  $A$  with size  $N \times N$  elements, matrix  $B$  with six  $N \times 1$  and Matrix  $X$  with size  $N \times 1$  elements. Figure 4-15 below shows these matrices. Matrix  $A, B$  is constant in the algorithm and stored in the memory in row major but matrix  $X$  is changed during the iteration  $K$  as described above. To compute  $x_i$  for  $k+1$  each element in the  $a_{ij}$  row is multiplied by each element in  $x_j$  column except the elements  $i=j$ , after that subtract it from  $b_i$  and divide it by  $a_{ii}$ . If  $x_{ij}$  not exist in the memory it generates read miss every time of computation of  $x_{ij}$ . The most expensive part is the matrix vector multiplication, the complexity is of  $O(N^2)$ .

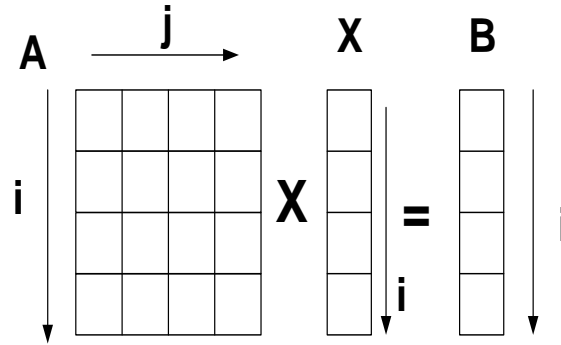


Figure 4-15 Jacobi Data Layout Representation

### 4.3.1 Jacobi Execution Model

To efficiently parallelize the Jacobi algorithm, the devised schemes should achieve data locality, minimize the number of synchronization, and maximize the core computations adjacent thread. By assuming, for simplicity, that the number threads divides exactly the dimension  $N$  of the  $N \times N$  matrix  $A$  and the vectors  $X$  and  $B$ . From dependency analysis the outer loop index  $k$  could not be carried out due to the F-LCD. So, the parallelization will occurred for the two loops inside the iteration loop.

Simplest parallelizing of Jacobi is done by inserting the directive on the outer most loop of the sequential implementation. Figure 4-16, explains the direct execution model of parallelizing of JACOBI.

```

for(k = 0; k < MAX_ITER; k++)
{
#pragma omp parallel for private(i,j,sum)
  for(i=0; i<N; i++){
    sum = 0.0;
    sum=sum-A[i*N+i] * X[i]; // S1
    for(j=0; j<N; j++){
      sum += A[i*N+j] * X[j]; // S2
    }
    new_x[i] = (B[i] - sum)/A[i*N+i]; //S3
  }
#pragma omp parallel for private(i)
  for(i=0; i < N; i++)
    X[i] = new_x[i]; // S4
}

```

**Figure 4-16 Direct Jacobi parallelization Code**

The most computation will be in the first loop (L1), where the dimension of matrix  $N$  is divided by the number of threads for the matrices  $A$ ,  $B$ ,  $X$ . So, each of thread will be responsible to compute sub solution for the matrix  $X$  depending on the thread number. The spreading of the work is done implicitly by the *FOR* constructs. In addition, to that no thread will go into the next iteration until all of the threads finish their work. Which is controlled by the implicit barrier inserted at the end of the construct. After all the threads have finished their work and have stored the results into *new\_x* the value was copied into the shared variable  $X$  and a new iteration started. The outermost loop keep executing until finishing the number of iteration.

#### **4.3.1.1 Synchronous Jacobi (SJ)**

The simple optimization code clarified above has a drawbacks on parallel programming style. However, entering and exiting from the parallel for constructs inside the iteration

space because an overhead combined with two for constructs. Hence, the code contains two implicit barriers that synchronize the work of the threads. To solve the previous problem a work sharing construct is used. Figure 4-17, illustrates an implementation that handles many issues in the first code. I called this implementation Synchronous Jacobi (SJ)

```

#pragma omp parallel shared (A,B,X,N,Xp,T,kk,temp) private(k,i,ii,j,sum)
{
    int tid=omp_get_thread_num();
    ii=tid*kk;
    for(k=0;k<MAX_ITER;k++)
    {
        for(i=0; i<kk ; i++){
            sum=0.0;
            sum=sum- (A[(i+ii)*N+i+ii]*X[i+ii]);
            for(j=0; j<N; j++){
                sum+=A[(i+ii)*N+j] * X[j];
            }
            Xp[i+ii]=(B[i+ii]]-sum)/A[(i+ii)*N+i+ii];
        }
        #pragma omp single
        {
            temp=X;
            X=Xp;
            Xp=temp;
        }
    }
}

```

**Figure 4-17 Synchronous Jacobi implementation, using work sharing constructs and single construct to optimize overhead**

Using the work sharing construct gives the programmer the facility to control the flow of the program and the work for each thread. I first omit the two parallel for construct and insert the iteration space into the parallel construct to reduce the overhead for entering and exiting from a parallel region. In addition, to that the number of the thread is retrieved depending on the thread ID the indices of the matrices  $A$ ,  $B$ ,  $X$ , is determined. After that each thread is responsible to compute a sub solution from matrix  $X$  in an iteration  $K$ . To remove the second for construct and copying the data into the shared variable, another variable is used. Hence, each thread in the iteration will read from a

variable  $X$  and store the result into the variable  $X_p$  in each iteration. To let the threads read from the last results from the next iteration  $K+1$  I swap the pointers using the single construct. It is used to make only the master thread swap the pointers and synchronize all the thread to this point. So, no thread will go into next iteration until the master thread finish swapping. In this case I only have one implicit barrier at the end of the single construct. Also using pointer swapping increases the possibility to find the sub solution  $X$  inside the cache when goes into the next iteration.

#### **4.3.1.2 Relaxed Jacobi (RJ)**

To reduce the overhead of synchronization and increase L2 cache reuse a Relaxed Jacobi (RJ) with blocking is implemented. I will assume there are  $N$  threads from  $(0, N-1)$ . Each thread will compute partial solution of  $X$  denote as  $X_{thi}$ . Also each thread will need partial matrix of  $A$  and all vector of  $B$ . Where  $A$  and  $B$  are constants during the iterative process. Only vector  $X$  is changing during computation process. Relax the synchronization causing an overlap between the iteration. By analyzing the dependency between the iteration only overlaps between iteration  $K$  and  $K+1$  can be done. This means that at any given time the thread can compute  $X(K)$  and partial solution from  $X(K+1)$ . This depends on the proceeding of the other threads computation. To Relax Synchronization I apply blocking technique using number of threads assigned to the processor. However, the number of partial solution will be number of threads ( $T$ ). Also the number of blocks for matrix  $A$  and  $B$  will equal number of threads. I simplify the algorithm in the following flow chart Figure 4-18.

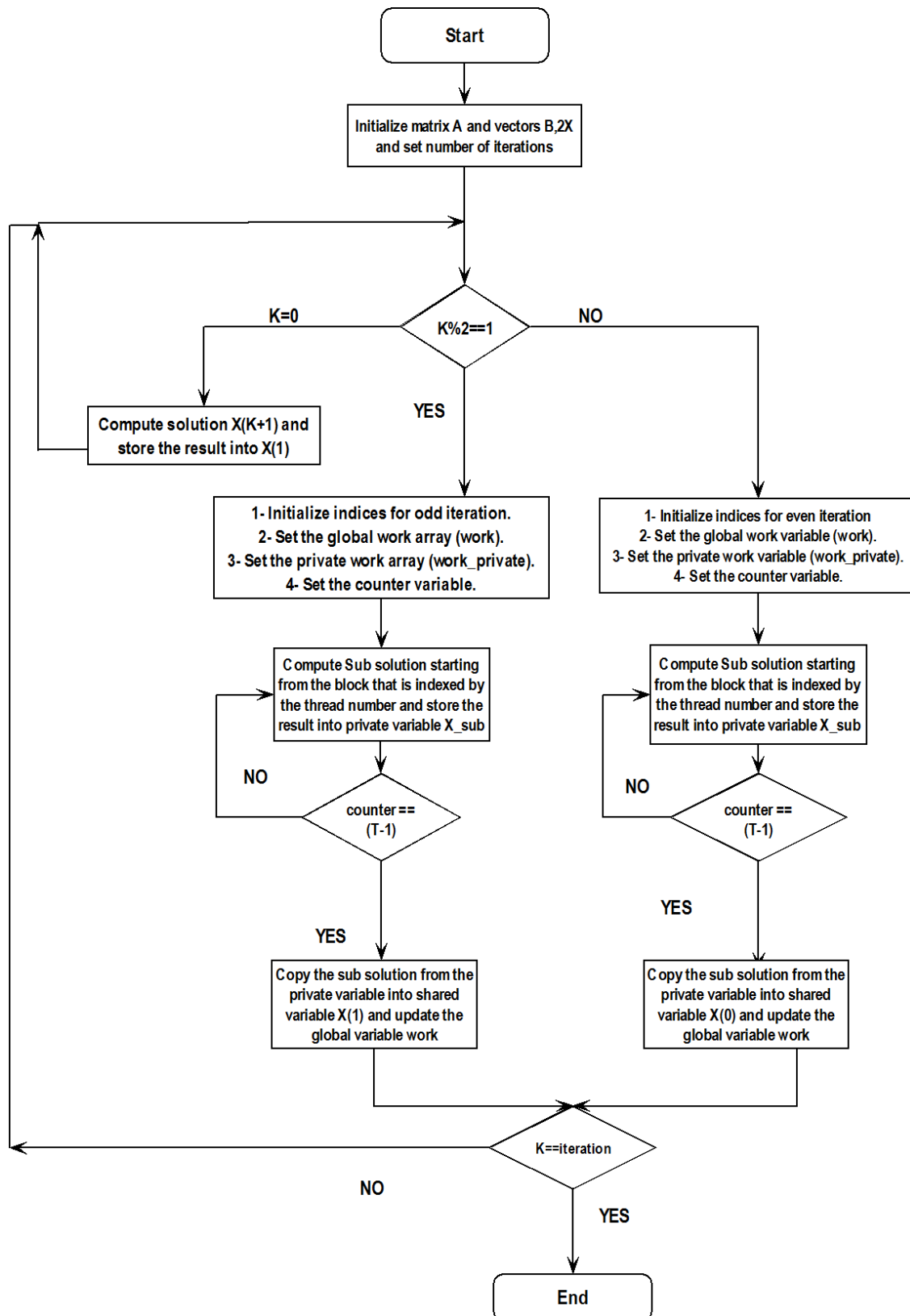


Figure 4-18 Relaxed Synchronization (RS) execution flow chart

The execution starts by initializing the matrix  $A$  and the vectors  $B$ ,  $2X$ . Where  $2X$  refers to the vector  $X$  which increased into double size to include the results from two iterations at the same time which are  $X(0)$  and  $X(1)$ . After that I check if  $K=0$ , which means that the computation of the blocks will proceed normally, because initially the solution  $X$  is exist at  $X(0)$  and no need for checking if the sub solution exist. When the execution start after  $K=0$  I need to differ from odd iteration and even iteration to determine the location and indices of the values that are needed in the computation process.

In both cases the process of execution starts by setting the shared variable work and the private variable *WORK\_PRIVATE*. In addition, to that the indices is initialized to point to the correct indices for matrix  $A$ ,  $B$ ,  $X$  and counter variable is settled.

Each thread starts computing sub solution by its ID. Because that block of solution will be ready as the thread will not proceed into next iteration until finishes its sub solution. Each thread computes the sub solution and increases the counter to insure that all the blocks have been proceeded. In case there is no blocks ready the thread spinning at this point and waiting for a work to be ready.

After the thread finishing its work at iteration  $K$  it copies the values from private variable *WORK\_PRIVATE* into shared variable  $X(K)$ . This allows the result to be shared between the threads. The threads continue computing until finishing all the iteration space.

### 4.3.2 Experiment Results

An experiment is done to evaluate the over head of the synchronization on JACOBI SOLVER. I compute the time that is spent in synchronization for one barrier over 100 iterations. Figure 4-19 shows time spent in synchronization one barrier over 100 iterations. To get accurate results the experiment is done 10 thousand times and the average time is taken.

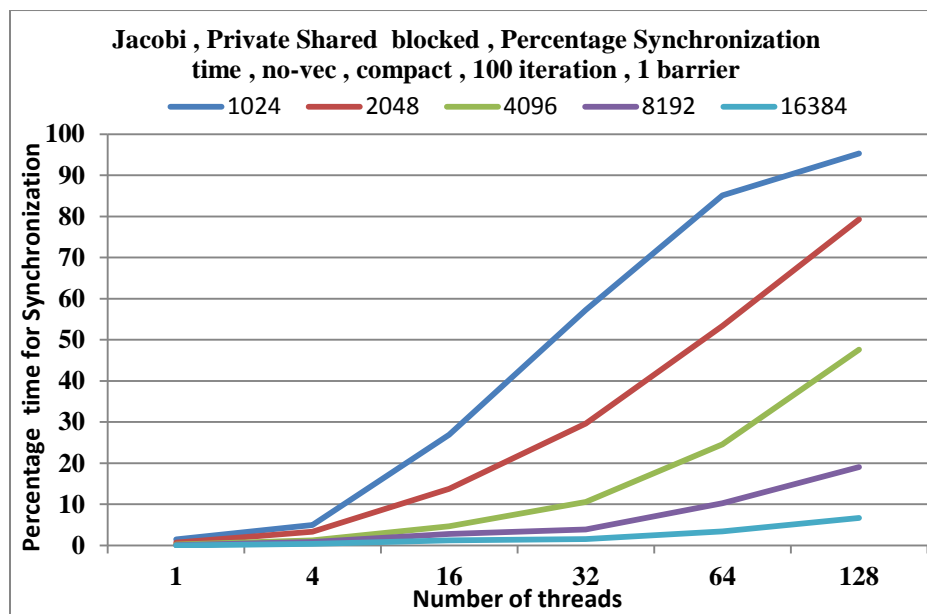


Figure 4-19 Percentage time spend in synchronization for 100 iteration

From the figure above I can conclude that the synchronization time take more than 95% of the execution time when matrix size is 1024. But its percentage decreases by 50% when the matrix size 4096. Also I can notice that when I have a large matrix size (16384) the percentage time is less than 9% of the total execution time. I can conclude that the synchronization has an overhead added to the execution time of JACOBI SOLVER which can affect the overall performance computation of the machine.



To evaluate my work in reducing the synchronization overhead and increase cache reuse. A set of experiments have been done on different implementations of JACOBI. I implement Synchronous Jacobi (SJ) which includes one synchronization barrier. Also I implement Asynchronous Jacobi (AJ) where the synchronization is removed. In addition, a Relaxed Synchronization (RS) is implemented. All the experiments run over 100 iterations for different matrix size and different number of threads. Figures below show the results

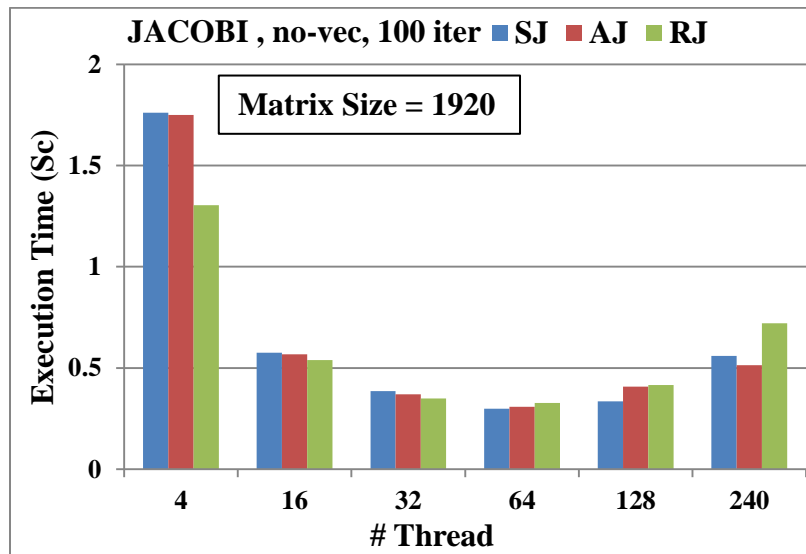


Figure 4-20 Jacobi experiment result for SJ, AJ, RJ for matrix size 1920

Figure 4-20 shows the execution time for running SJ, AJ and RJ for matrix size 1920 with different number of threads. It appears that the RJ is better than the other two implementations until number of threads equal to 32 threads. After that the execution time of the RJ becomes greater than the other. This happened due to that the overhead for blocking and relax synchronization for more than 32 threads for smaller matrix size will be greater than the execution time of the SJ, AJ. Therefore, the benefits of this method can be seen with larger problem size.

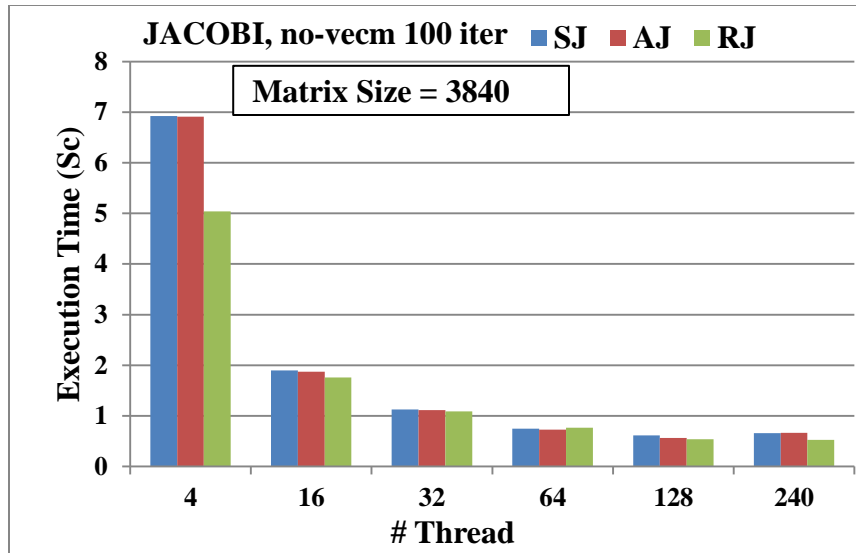


Figure 4-21 Jacobi experiment result for SJ,AJ,RJ, matrix size 3840

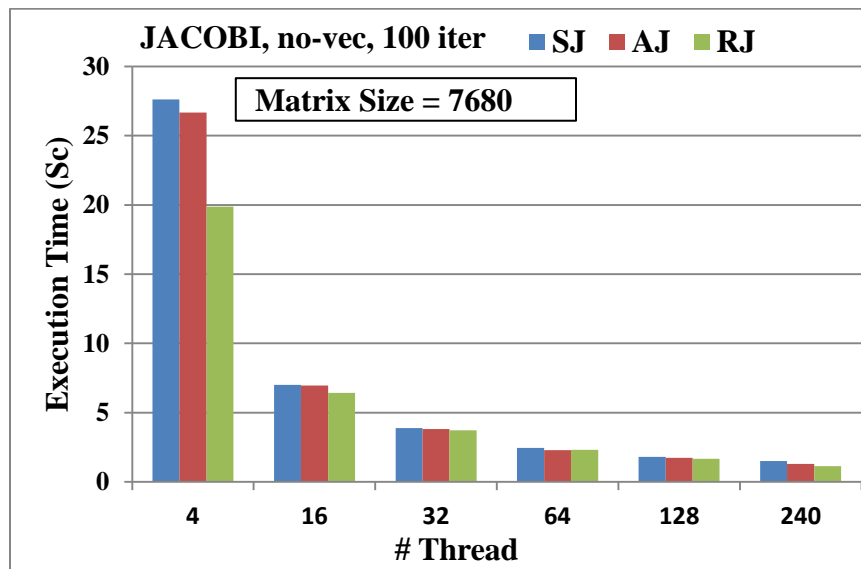


Figure 4-22 Jacobi experiment result for SJ,AJ,RJ, matrix size 7680

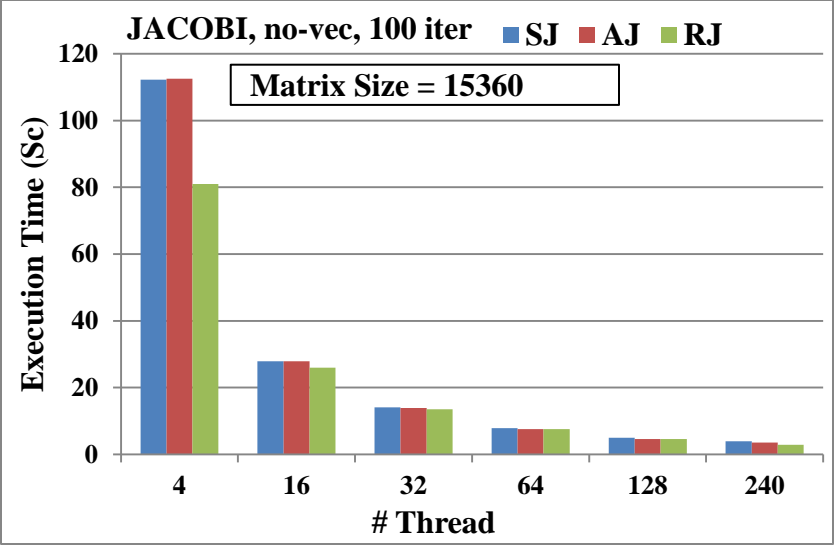


Figure 4-23 Jacobi experiment result for SJ,AJ,RJ, matrix size 15360

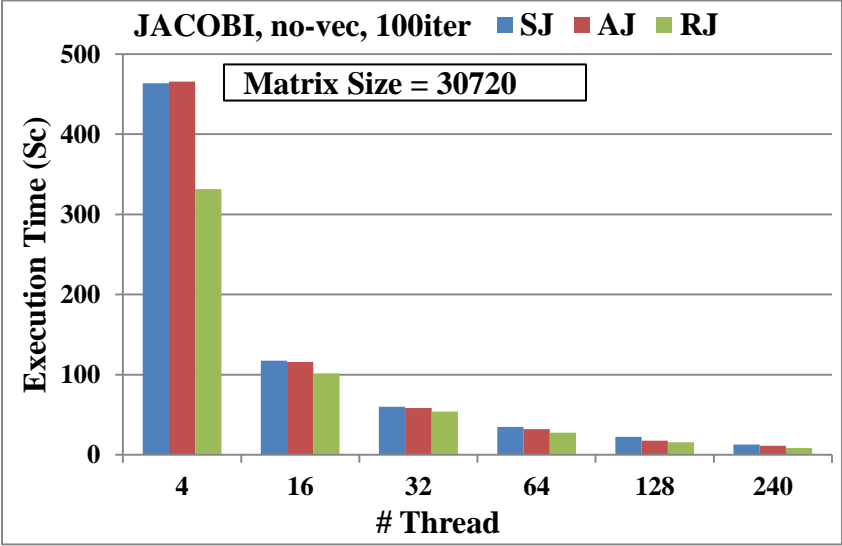


Figure 4-24 Jacobi experiment result for SJ,AJ,RJ, matrix size 30720

Figure 4-21 until Figure 4-24 show the execution time of SJ, AJ and RJ for matrix size 3840, 7680, 15360 and 30340 respectively with different number of threads. I gain a percentage of improvement in execution time from RJ 24.4%, 32.6%, 38.9%, 57.16% respectively over the SJ. The decreasing in execution time of RJ caused by the blocking matrixes A, B and D which increase the reuse of L2 cache and decreasing the cache

misses. However, every time the thread starts computing its new sub solution it will start from the previous block that is computed by it. So, in this case it is obviously will find it in the cache before it is evicted. Also, the other threads in this schema can find the block in other cores all around the MIC. Therefore, the time needed to get the data from cache of other core will be less than the time needed to read the data from the global memory.

### **4.3.3 Conclusion**

In summary, I have presented a synchronization optimization technique for JACOBI SOLVER. My technique includes relaxed synchronization across the iteration space. To achieve that I apply blocking for the matrixes  $A$ ,  $B$ ,  $X$  along with the number of threads. To evaluate my work three implementations of JACOBI SOLVER have been implemented SJ, AJ, and RJ. SJ contains one synchronization barrier, AJ the synchronization barrier removed and the RJ which contains a Relaxed Synchronization with blocking. Results show that my technique outperforms the SJ with a percentage of improvement up to 57% on large matrix size.

## CHAPTER 5

### SEMI-STATIC PROBLEM (N-BODY SIMULATION)

#### 5.1 Introduction

The nature of real world large and complex problems makes it inefficient to implement in single normal processing units. The computation of these problems requires more processing resources and larger storage and memory elements [36]. The N-body Simulation one of such classical problems that was concord in predicting the individual motions and forces a group of objects interacting with each other gravitationally. It is a semi static problem where the load balancing affects computation performance. Hierarchical methods such as the Barnes-Hut (BH) and Fast Multipole method (FMM) are recently being used to solve the N-body problem since these methods can be run faster by utilizing parallelism and applications that use them are likely to be among the domain of HPC. The challenges of such methods are the problem of partitioning and scheduling for effectively utilizing the parallelism. In addition, the distribution of the workload among the processing elements complicate more the computation since the structure is changing as the computation proceeds. As a result, the issues of load balancing and data locality were of the main concern.

The simplest approach to tackle N-Body problem is to iterate over a sequence of small time steps. Within each time step, the acceleration on a body is computed by summing the contribution from each of the other  $N-1$  bodies which is known as brute force

algorithm. While this method is conceptually simple, easy to parallelize on HPC, and a choice for many applications, its  $O(N^2)$  time complexity make it impractical algorithm for large-scale simulations involving millions of bodies.

To reduce the brute force algorithm time complexity, many algorithms have been proposed to get approximated solution for the problem within a reasonable time complexity and acceptable error bounds. These algorithms include Appel [37] and Barnes-Hut [38]. It was claimed that Appel's algorithm run in  $O(N)$  and Barnes-Hut (BH) run in  $O(N \log N)$  for uniformly distributed bodies around the space. Greengard and Rokhlin [39] developed the Fast Multipole Method (FMM) which runs in  $O(N)$  time complexity and can be adjusted to give any fixed precision accuracy.

For the semi static problem, the N-body simulation is considered as a model of semi static computations. A brute force approach for computing the gravitational forces for N bodies is on the  $O(N^2)$ . The Barnes Hut (BH) approximation enables treating a group of bodies as one if these are far enough from a given body. This drops the computational complexity to  $O(N \log N)$  when using BH. BH uses an oct-tree, in which each node stores the aggregate mass of all of its children nodes (sub-tree) at their center of mass. Another problem is that the thread load moderately changes from one iteration to another due to body motion in space. Therefore, a static problem partitioning strategy (S-BH) for BH is likely to suffer from accumulated load unbalance. It well known that dynamic load balancing (DLB) improves BH scalability. However, DLB is complex because of the need to measure the Dynamic Load (DL) and adopt an adequate data structure to minimize runtime overheads. In the beginning of iteration k, the body slowly motion

enables estimating the DL for  $K+1$  as being the aggregate load measured by all the threads in iteration  $k$ . Thus DLB is implemented by evenly partitioning the DL over the threads so that to preserve the data locality to the best possible. I implemented DLB-BH using an efficient data structure to ease load redistribution together with oct-tree implementation.

## 5.2 BARNES-Hut (BH) Algorithm

BH algorithm is based on dividing the body space that contributes on a given body into near and far bodies[38]. For near bodies, the brute force algorithm can be used to compute force applied on that body from other bodies while far bodies can be accumulated into a cluster of bodies with a mass that equal to the total mass of the bodies in that cluster and the position of the accumulated cluster is the center of mass of all bodies in that cluster.

BH suggested the use of tree data structure to achieve this clustering while working within a reasonable time complexity. Figure 5-1 illustrates an adaptive BH quad tree here each leaf contains only one particle. Tree data structures exploit the idea that an internal node in the tree will contains the center of mass and total mass of all of its descendants. In this case, computing the force applied on a far body from a given sub-tree will require accessing to the parent of the sub-tree and use its center of mass and total mass without the need to go farther in the sub-tree. This will decrease the time required for computing force on a given body noticeably. Sequential BH algorithm is sketched in the Table 5-1 which can be applied and implemented for both 2-D and 3-D space. This algorithm is repeated iteratively as many as required number of iterations.

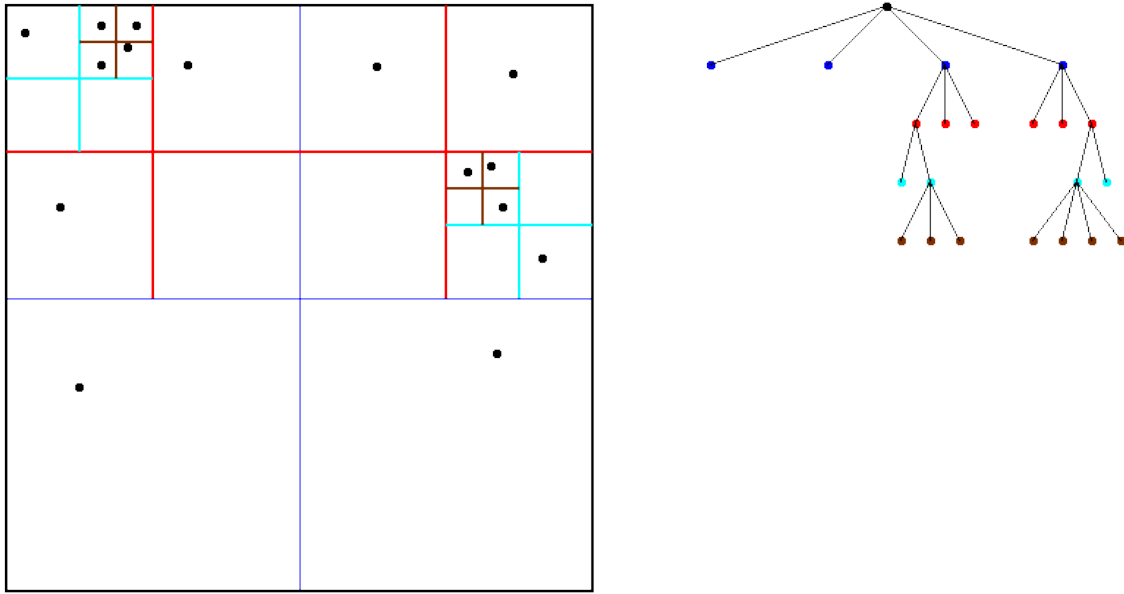


Figure 5-1 Adaptive Quad Tree of BH for 2D Simulation

The nodes of the Quad tree in 2D or Octree in 3D are traversed starting from the root to calculate the net force on a particular body that illustrates the BH approximation for force computation. If the center of mass of an internal node is sufficiently far from the body (p), bodies contained in this sub tree approximated as a single node. Otherwise the process continues for the other children.

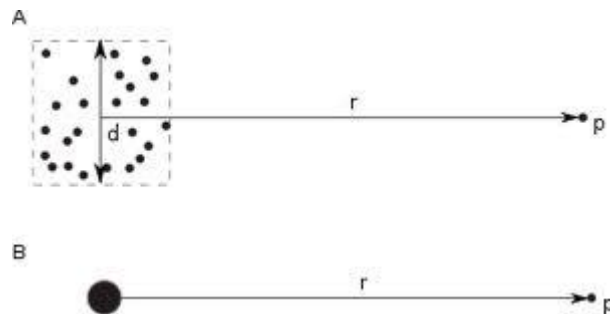


Figure 5-2 Barnes-Hut approximation in computing force for far bodies



Similarly the operation is done on 3D space. The difference is an Oct Tree is used.

Figure 5-3 demonstrate that each node will have 8 children.

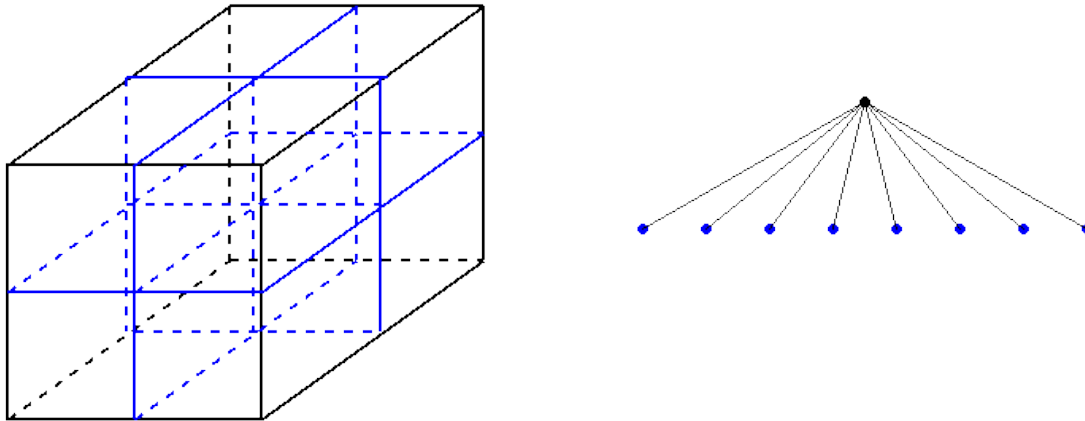


Figure 5-3 Oct-Tree for 3D Barnes-Hut Simulation

Table 5-1 Sequential Barnes-Hut Algorithm

For each time step:

- Construct the BH tree (quad-tree for 2-D and Oct-tree for 3-D)
- Compute center of mass and total mass bottom-up for each of the internal nodes.
- For each body:
  - Start depth-first traversal for the tree, if center of mass in a given internal node is far from the body of interest then compute force from that node and ignore the rest of the sub-tree
  - Finished traversing the tree then update the position of the body and its velocity.
- delete the tree

### 5.3 Related Work

The Barnes-Hut algorithm recently has been successfully parallelized using several techniques on both heterogeneous and shared memory HPC [40-48]. The key challenges

in parallelizing the Barnes-Hut algorithm include break down the domain of the BH tree across the allocated memory resources and load balancing the workloads across the threads. Load Balancing and Data Locality in hierarchical N-body algorithms, including the Barnes-Hut algorithm, was studied. The papers concluded that straightforward separation techniques which an automatic scheduler might implement do not scale well, because they are unable to simultaneously provide load balancing and data locality.

A source dividing strategy for MPI systems is implemented in [40]. A contra intuitive method is proposed in which the source points are divided among the processors. Each processor forms its own tree out of the source points assigned to it, and computes contributions from these source points on all the target points. Once this evaluation is complete, the processor communicates the results to the head processor. The head processor adds up the contributions on each target from all sources, and broadcasts the results to other processors. Also, a dynamic load balancing scheme for time dependent applications on heterogeneous systems composed of multiple CPUs and GPUs across multiple time steps[41] have been used. The load balancing strategy performs fine grain local modifications to the adaptive decomposition tree to minimize runtime informed by a time costing model. In addition, incremental global modifications track the evolving distribution of bodies. The load balancing machinery operates in one of three states: search, incremental, and observation. During the entire course of the simulation the load balancer is always in one of these states. Each lasts over multiple time steps. The current state of the load balancer defines how load balancing functionality is carried out and/or which actions shall be taken if undesirable run times are seen.

Optimizations for parallel BH algorithm based on NoC platform from both aspects of software and hardware is done in [43]. In terms of software, consider distribution tree data across physically distributed cache. Their platform is a shared L2 cache system the shared L2 cache is divided and distributed into different nodes in terms of cache slices. It is therefore reasonable to distribute the tree data, including body, cell and leaf information, to the local caches of cores.

A Partitioning global address Space (PGAS) using Unified parallel C (UPC) is implemented with cost zone [44]. UPC BH inherited from the shared memory SPLASH-2 BH code the cost-zone load-balancing algorithm. However, this algorithm is computation-centric. On distributed memory the need to access remote cells can disturb the balance. Because of SFC ordering, boundary processes on a node usually require more remote cells than do interior processes. Considering computation/ communication overlapping, the effect is hard to estimate upfront and thus is better attacked by dynamic scheduling enabled by multithreading. On the other hand, Orthogonal Recursive Bisection (ORB) is applied in [46]. The domain decomposition is used to divide the space into as many non-overlapping subspaces as processors, each of which contains an approximately equal number of bodies, and assign each subspace to a processor.

An introduction to the geometric characterization of a class of communication graphs that can be used to support hierarchical N-body methods, [23]. The issues that are related to the practical aspects and implementation of hierarchical N-body methods such as the depth of the hierarchical structure were also discussed. These confirm the need for another representation in practice rather than relying on the Oct-Tree. Data structure for the Barnes-Hut was also implemented by Dekat.al. The Oct-Tree was represented by two

arrays; one is the input array that represents nodes in the Oct-Tree and the second is the nodes organization for the next iteration. This approach was introduced to overcome limitation in the scalability for parallel implementations of Barnes-Hut and to effectively utilize and manage the space and memory resources. The scalability of the BH was deeply analyzed by Speck, [49] using UPC language. They suggested that using shared memory in which shared variables can be cached locally without changing the reference used to access them leads to achieving good performance while global references are being used.

The fundamental complexities to improving performance and scalability of parallel N-body simulations using the BH algorithm are as follows:

- Dynamic load balancing: A new unique tree is produced in each iteration during the simulation. Moreover, static load balancing technique is ineffective. It has a poor load balancing performance.
- Variable workload: In addition to the variation of the work load. These systems have a variable workload per particles. Therefore predicting work load per particle is difficult.
- Data-driven computation: BH is a data driven algorithm. Thus it has an irregular communication data access pattern. This makes conventional parallel optimization inefficient.
- Data locality: The irregular and unstructured computations in dynamic graphs can result in poor data locality resulting in degraded performance on conventional systems which rely on exploitation of data locality for their performance.

## 5.4 Effective parallelization using Cost Zone

In this section I define the scope of my objective to effectively parallelize force computation of BH, and the goals of my parallel implementations. As I described later there are two main issues that affect scalability of BH force computation:

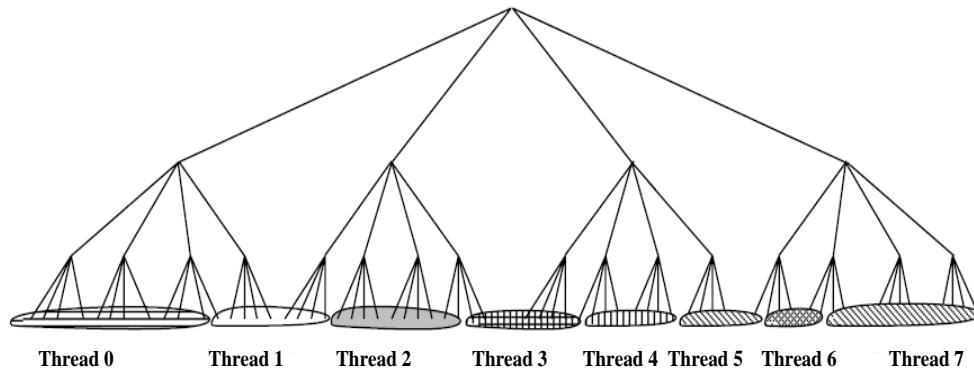
**Load Balancing:** The goal in load balancing is intuitive: workload should be assigned to threads evenly. Therefore, the maximum difference between the execution time of the threads are the minimum.

**Data Locality:** many cores are built with hierarchical memory systems, in which threads in each core have faster access to data the same core cache or in other cores. To improve performance of the applications, I need to increase data locality. Thus, I increase sharing of data between threads in the cores to decrease cache misses. For this reason, I focus my discussion of locality primarily on reducing cache misses by ordering the structure of the array in such way that accesses the same data on the same core. However, I do make all reasonable efforts to exploit locality within a core effectively.

## 5.5 Distribute Work Using Cost Zone

Cost zones partitioning technique takes advantage of another key insight into the BH hierarchical methods for conventional N-body problems, which is that they already have a representation of the spatial distribution implicitly found in the tree data structure they use. I can therefore partition the tree rather than partition space directly. In the cost zones approach, the tree is conceptually laid out in a two-dimensional plane, with a node's

children laid out from left to right in increasing order of child number. Figure 5-4 demonstrates an example using a quad tree for simplicity.



**Figure 5-4 Cost Zone Demonstration of work distribution for 8 threads**

The cost of every particle, as counted in the previous step, is stored with the particle. Every internal node contains the cumulative count of the costs of all particles that are contained within it, these node costs have been computed during computation of center of mass and total mass.

The total cost in the domain is divided among threads so that every thread has a evenly amount of work. For example, a total cost of 128 would be dividing across 32 threads so that the zone containing costs 1-4 is assigned to the first thread, zone 5-8 to the second, and so on. Which cost zone a particle belongs to is determined by the total cost up to that particle in an in order traversal of the tree. In the cost zones algorithm, threads descend the tree in parallel, selecting the particles that belong in their cost zone. A thread therefore performs only a partial traversal of the tree. To preserve locality of access to internal cells of the tree in later phases, internal cells are assigned to the thread that own

most of their children. In my work I concentrate on the effect of applying cost zone in the force computation, parallelization of tree traversal can be studied later in future work.

## 5.6 Reserve Locality Using Morton Order

Preserve locality when applying cost zone should produce partitions that is closer in the space as in the plane. How well this closeness in the tree corresponds to consecutively in physical space depends on how the locations of cells in the tree map to their locations in space. This depends on the order in which the children of cells are numbered. The simplest ordering scheme to use is the Morton Order (Z-order). Figure 5-5 demonstrates how it is applied for both 2D and 3D. It is a function that maps multidimensional data to one dimension while ensure locality of the data points [50].

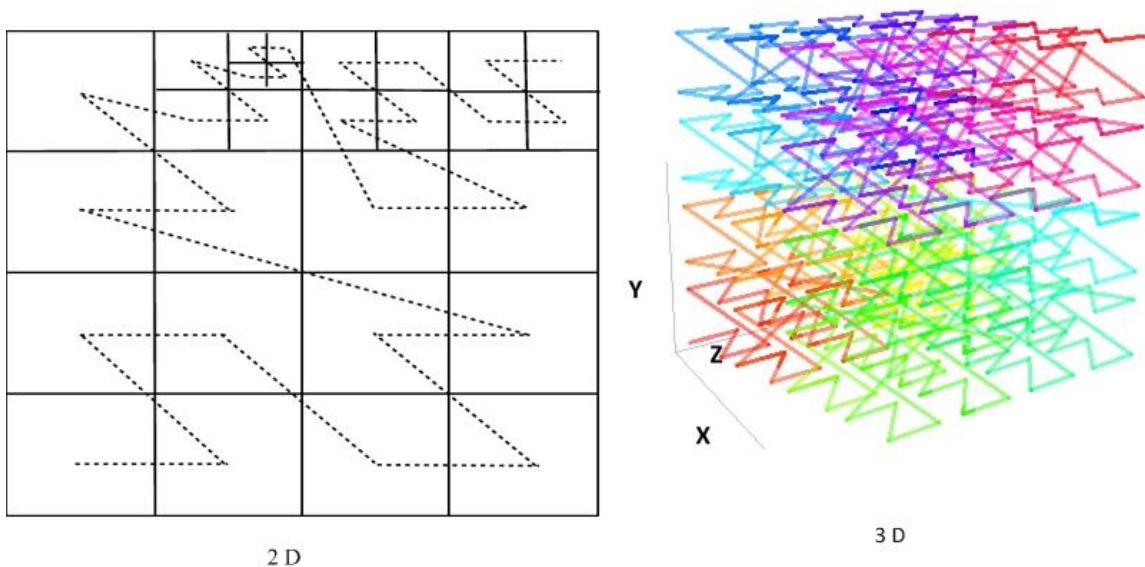


Figure 5-5 Morton Order representation, left for 2D, Right for 3D [50]

## 5.7 Iterative Cost Zone Load Balancing (ICZB) Implementation

In this section I will describe main steps of my implementation of the Dynamic Load Balancing for Barns Hut (DLB-BH). I implemented a dynamic load balancing using cost zone combined with reserved data locality. I will call it Iterative Cost Zone load Balancing (ICZB). I construct the tree as a plane data structure (array). Thus, the overhead of recursive call for traversing the tree is omitted. Moreover, arrays are more cache friendly with many core machines.

### 5.7.1 N-body Implementation Steps

The implementation of the BH N-body algorithm follows a certain Steps. Figure 5-6 below illustrates the main algorithm steps. I will describe them later in the next sections.

---

➤ Load bodies

For  $i=0$  to  $N$  where  $N$  number of iterations

- Oct-tree creation
  - Depth-First Tree Traversal
  - Sort the nodes array according to the traversal order
  - Sort the bodies array according to the traversal order
  - For each Body
    - Compute its force by traversing the arrays
    - Update the velocity and the position of each body
  - Delete Nodes and free the memory
  - Delete Bodies and free the memory
-



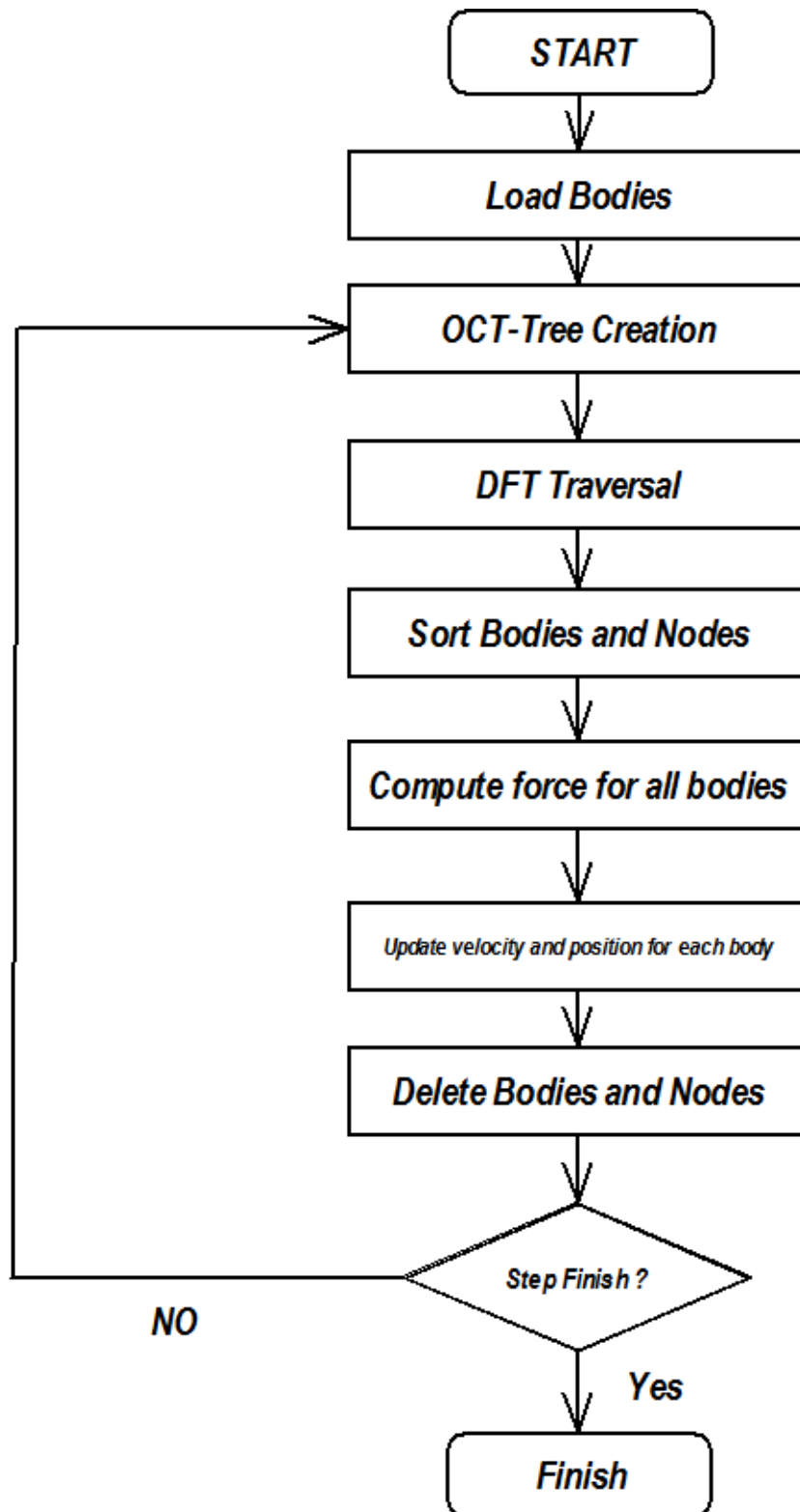


Figure 5-6 N-Body with Barnes Hut Execution Steps

## 5.7.2 Load Bodies

The algorithm starts by loading the bodies' data from a text file. The file consists of 10 columns which represents the mass, position, and velocity (i.e. mass, x, y, z, vx, vy, and vz respectively). Each row in the file represents a body. The data in each row are space-separated.

## 5.7.3 Oct-Tree Iterative Creation Algorithm

The Oct-tree creation is the first step in the iteration. The iterative algorithm as implemented illustrated in Figure 5-7. It takes all bodies in turn and inserts them in the Oct-Tree starting from the root. For each body it calculates the appropriate cube among the possible eight cubes. If it meets a free node, it puts the body there and loops for the next body. Otherwise, if it is not a free node, then it goes deeper following the appropriate path until it reaches the leaf. At this point, it may find an empty cube so it puts the body. If it meets a body that belongs to the same cube, then both bodies need to go deeper until getting a separate cube for each.

In Oct-Tree algorithm, the last step could loop forever. Consider an example where there are two points that are very close together or even exactly in the same location. These bodies could not be separated easily into two different cubes. It is important to check whether the cube dimension approached zero. For this case, I added an additional condition to test the dimension of the current node (e.g.  $d > 1.0E-6$ ).

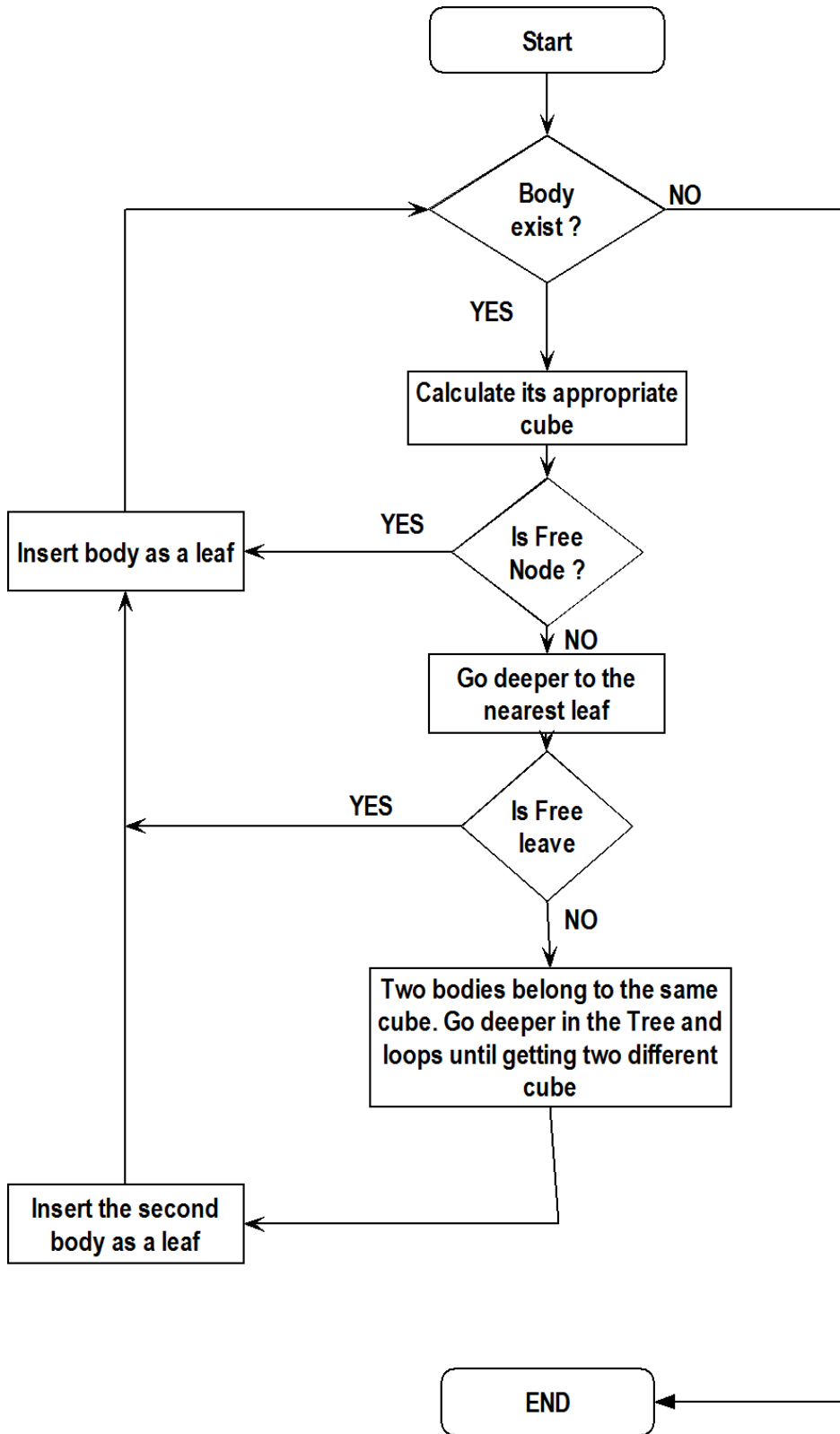


Figure 5-7 Oct-tree iterative creation algorithm

To reserve the locality for my implementation, I implement the Morton order for ordering the cubes of the Oct-tree as shown in Figure 5-8. Since each node in the tree has eight children a given body and a node, the following code line says exactly the cube order within the children of the node.

---

**$i = (\text{Body.z} > \text{Node.z}) * 1 + (\text{Body.y} > \text{Node.y}) * 2 + (\text{Body.x} > \text{Node.x}) * 4;$**

---

For example, if Body.z is greater than Node.z and Body.y is greater than Node.y while Body.x is less than Node.x then  $i = 3$ .

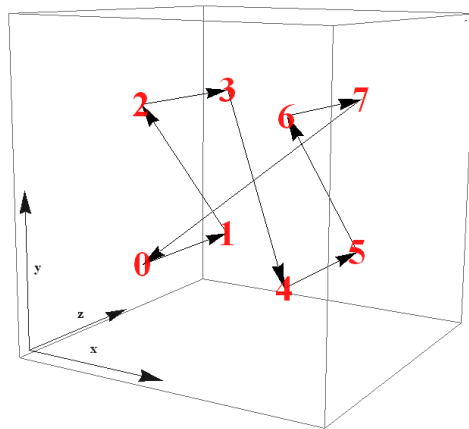


Figure 5-8 illustration of the chosen Morton order in my implementation. The numbers represents the order of selecting cubes.

### 5.7.4 Iterative Depth First Tree Traversal

After building the tree, I need to compute the center of mass in each node. This should be done from bottom to up (i.e. from leaves to the root). Since I have a tree, I should start from the root and traverse the tree in the depth-first order. While traversing, the algorithm Figure 5-9 computes the center of mass on each node. It also assigns a serial number for each node that reflects the traversing order. In addition to that, the algorithm assigns a pointer on each node to the next sub-tree. This index is used when applying BH in the force computation algorithm.

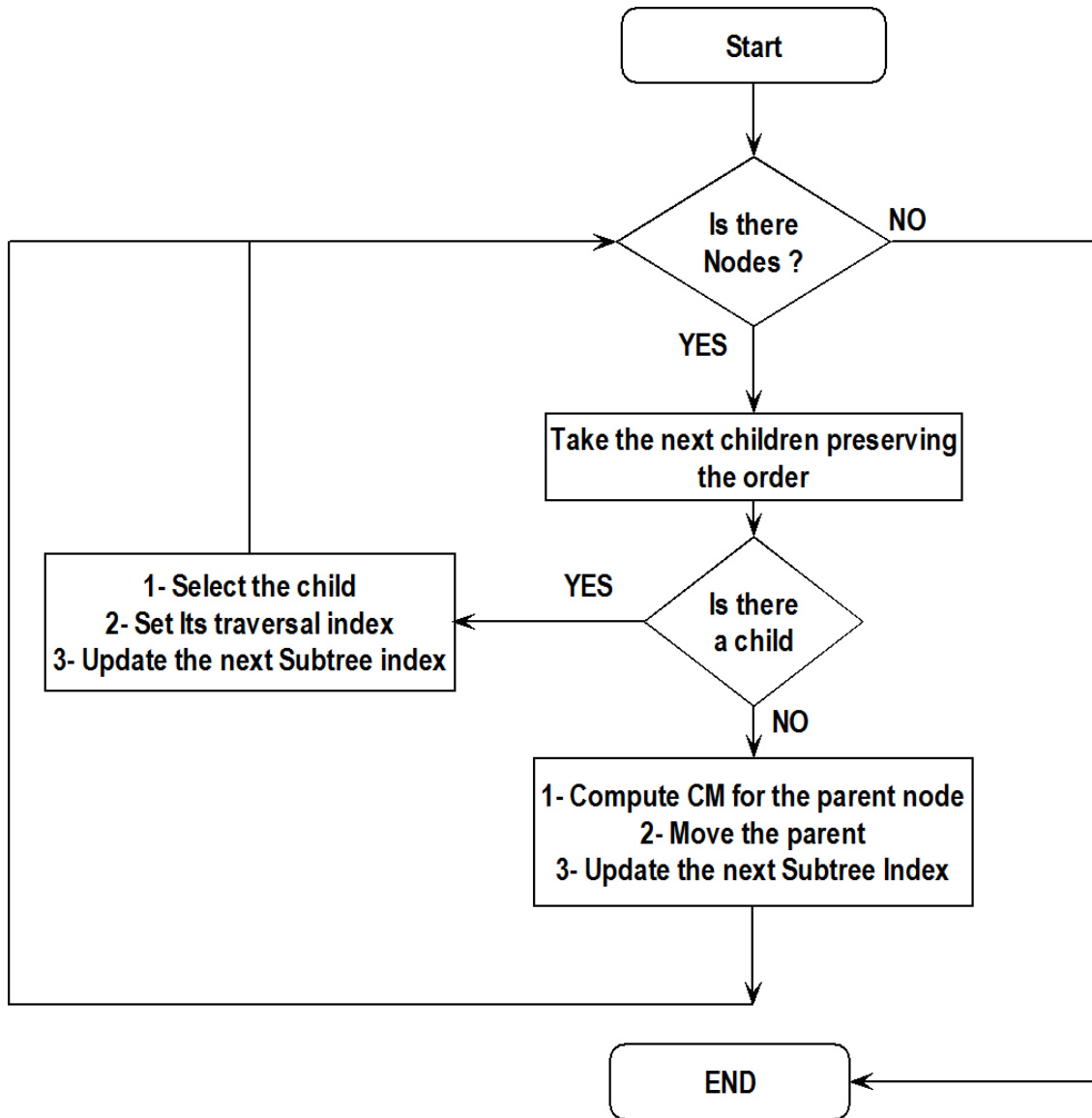


Figure 5-9 Tree traversal algorithm.

### 5.7.5 Sorting Node and Body Arrays

After the tree traversal step, I sort the node array according to the traversing order obtained from this step. I also sort the body array according to this traversal. This step is very essential in my implementation, although it adds some overhead to the overall algorithm. It decrement some overhead of the force computation step. There is no need to

traverse the tree for computing forces. Rather, the force computation visits arrays in a very smooth manner.

### 5.7.6 Converting Oct-Tree Into Data Structure

Considering the tree in Figure 5-10 which contains 19 nodes, the leaves represent at the edges of the tree. While the remaining are internal nodes (i.e. 0, 2, 7, 9, 13). The resulted nodes array after sorting has the following structure:

[(0, 19), (1, 2), (2, 6), (3, 4), (4, 5), (5, 6), (6, 7), (7, 17), (8, 9), (9, 10), (10, 11), (11, 12), (12, 13), (13, 14), (14, 15), (15, 16), (16, 17), (17, 18), (18, 19)]

But the bodies array has the following leaves

[1, 3, 4, 5, 6, 8, 10, 11, 12, 14, 15, 16, 17, 18]

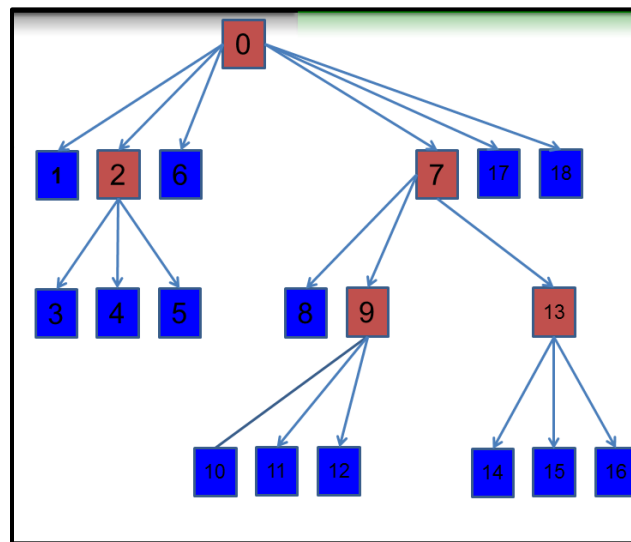


Figure 5-10 An example shows the depth-first traversal order. The nodes are sorted in the array according to this traversal. The leaves which represent the bodies also sorted in the array according to this order. Each node also store an index of the next node in the tree

Each element in the Nodes array has the index of the next node within the array which is always greater than the current index. This allows the force computation function to prone any sub-tree by moving to this index.

### 5.7.7 Iterative Force Computation

The force computation function receives two arrays, one for bodies and the other for the nodes. The bodies are sorted according to their locations as leaves of the tree. The node array is sorted in such a way to reflect the depth-first traversal. When the force computation function applies BH to a node, the node gives the index of the next element in the array and prone its children from the computation.

To fasten traversing the tree, arrays are used to allow smooth linear movement with no branching. The only branch is taken when BH applies which prone a sub-tree. The force computation iterative algorithm is explained in the following steps:

- 
- For each body in the body array.
    - For each node in the node array
      - If (the distance between body and current node is  $\geq d * 2$  apply BH Set the next index to the next sub-tree to prone the current sub-tree.) or (the current node is a leaf node ).
      - Compute the force of interacting with the current node and increment w ( counter for the work ).
    - Increment the index and loop to the next node.
  - Loop to the next body.
-

### 5.7.8 Iterative Cost Zone Load Balancing (ICZB)

Parallelization of BH presents challenging load balancing problem that must be addressed dynamically as the system evolves to distribute the work among the threads. Construction of the tree iteratively makes applying Cost Zone efficiently. However, traversing the tree and cumulative the work over the nodes done without extra overhead that affect the overall performance. To compute the work I put a counter inside the inner loop of the force computation algorithm. This counter reflects the number of elements visited for each body and the force applied from it. I used this number to reflect the Work of that body.

After building the tree in each iteration, I traverse the tree from bottom to top and accumulate the work of each sub-tree. Since the bodies are sorted before entering the force computation, the work per thread could be computed easily by traversing these bodies in order. I can explain my method in the following steps:

---

Let  $W$  = Overall Work, and  $T$  = number of threads

$Work\_per\_thread = W/T$

Thread [t]. start = body [i]

while sum <  $Work\_per\_thread$

{

    sum = sum + body [i].Work

    increment i

}

Thread [t]. end = body [i]

Let sum = 0,  $T = T - 1$ ,  $W = W - sum$



```
If T > 1 loop to 2
Thread [t]. End = body [last]
```

---

The algorithm simply divides the overall work over the number of threads and assigns the first thread the first M bodies whose works sums to thread quota or less. Then it reduces that sum from the overall work and divides the rest work among the rest of the threads.

### **5.7.9 Practical Challenges**

In addition to the challenges that are faced in applying BH also I face practical problem in implementing BH. I address several problems that could limit the efficiency of my implementation. It includes the limitation of stack space and the limitation of oct-tree depth.

#### **5.7.9.1 Stack Overflow**

To handle very deep recursive calls while building the Oct-Tree. An iterative version of the Oct-Tree has been tested. I use it to observe how much the increase in number of bodies the algorithm can handle. Experiments show that deep recursive calls add extra overhead in the execution time.

#### **5.7.9.2 Limitation Space dimension**

Maximum depth of the Oct-Tree is another problem. When the algorithm starts, I put the dimension for the root node equal to space dimension. Then when I go deeper in the tree I decrement  $d$  by a factor of two and put  $d = d/2$  for each level. Hence, for a given level I have  $d = d/2^{\text{Level}}$ . It is obvious that  $d$  value drops very fast. Since the maximum number of bits of the machine for any data type is 64 bits, it is not possible to have a tree that have a

depth of more than 64 levels. If  $d$  approaches zero, no need to divide the space any more to have new eight children. I just distribute the bodies among these children without concerning about the Morton order at this point.

## **5.8 Implementation Correctness Checking**

I run several experiments to check the implementation correctness for both sequential and parallel implementations. To make it possible to trace the execution results for sequential, I create different number of bodies that is spread across a sphere inside the space. To insure that the force on each body will be the same, I test the result from 4 particles and increase it to 16 particles. After that I check manually the computation of the force and the motion of the particles during different iteration. After that I simulate thousands of bodies and plot their motion using MATLAB. All the operations (OCT-Tree creation, compute enter of Mass an total mass, compute forces, update velocity and positions, delete OCT tree and far Tree)I s tested during the hand check process for the sequential.

Parallel implementation also has been checked. However, both the results from sequential and parallel implementation are reported into different text files. After that a script has been written to compare the results from both to insure that the parallel implantation working right. An individual test was also carried out to check ICZB implementation by reporting the total work for different problem size and was compared to actual result that must be obtained. In addition to that the number of times of BH applied and don't applied is used.

## 5.9 Body Generation and Dataset

The main galaxy is generated using the McLuster tool[51]. It is a tool that used to generate different types of galaxies for astronomy simulation. The bodies data is produced using the king model option. The generated galaxy is shown in Figure 5-11. The tool generates 100 thousands of bodies. Then I used the code shown in Figure 5-12 to generate my dataset by duplicating them many times. The distance between each galaxy is 60 point in a space of size  $400 \times 400$ . Hence, I saved the resulted bodies into a file. Figure 5-13 represents the distribution of the galaxies and particles for 1M size problem.

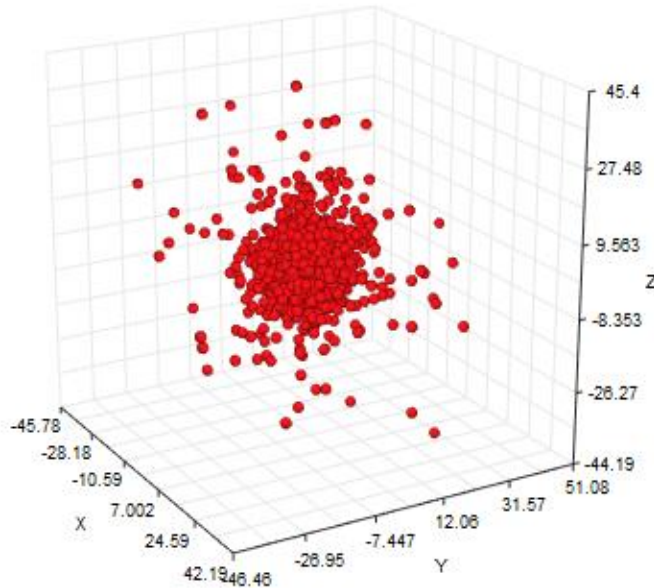


Figure 5-11 King Model galaxy, which contains  $10^6$  bodies. Plotted using TeraPlot Visualizer

```

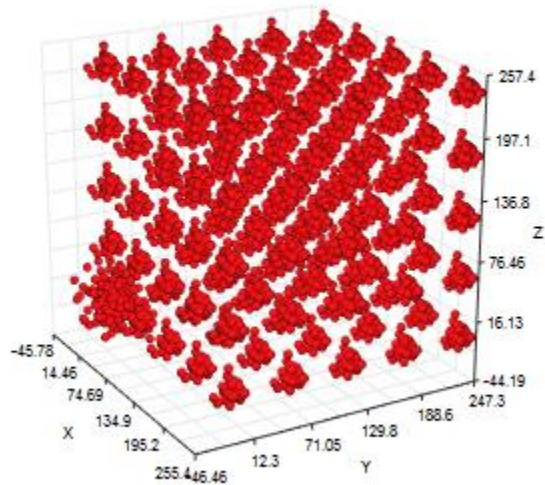
void Generate_GalaxyKing()
{
    int ii = 0;
    MaxNBodies = 100000;
    loadBodies(filename_GalaxyKingModel_100);
    for (int b = 0; b < 100000; b++)
        for(int x = 0; x < 5; x++)
            for(int y = 0; y < 5; y++)
                for(int z = 0; z < 5; z++)
                {
                    ii = 100000 + z + y*5 + x*5*5 + b*5*5*5;
                    Bodies[ii] = new Body();
                    Bodies[ii]->posx = Bodies[b]->posx + x * 60;
                    Bodies[ii]->posy = Bodies[b]->posy + y * 60;
                    Bodies[ii]->posz = Bodies[b]->posz + z * 60;

                    Bodies[ii]->mass = Bodies[b]->mass;
                    Bodies[ii]->vx = Bodies[b]->vx;
                    Bodies[ii]->vy = Bodies[b]->vy;
                    Bodies[ii]->vz = Bodies[b]->vz;
                }
    MaxNBodies = ii + 1;
    saveBodies(filename_GalaxyKingModel_1M_nonsorted);

    std::sort(Bodies, &Bodies[MaxNBodies], Body_comparer_function);
    saveBodies(filename_GalaxyKingModel_1M);
}

```

**Figure 5-12 Galaxies generation procedure**



**Figure 5-13 Bodies distribution for a data set of 1M**

## 5.10 ICZB Evaluation

A specific metrics is used to evaluate my implementation with comparison to the static approach (divide the bodies evenly across the threads). Effectiveness comparison is carried out depending on the Speedup that obtained over the same sequential program. I present speedup obtained on MIC using different number of threads and particles. In addition to speedups, I also present results that separately compare the load balancing, locality and overhead.

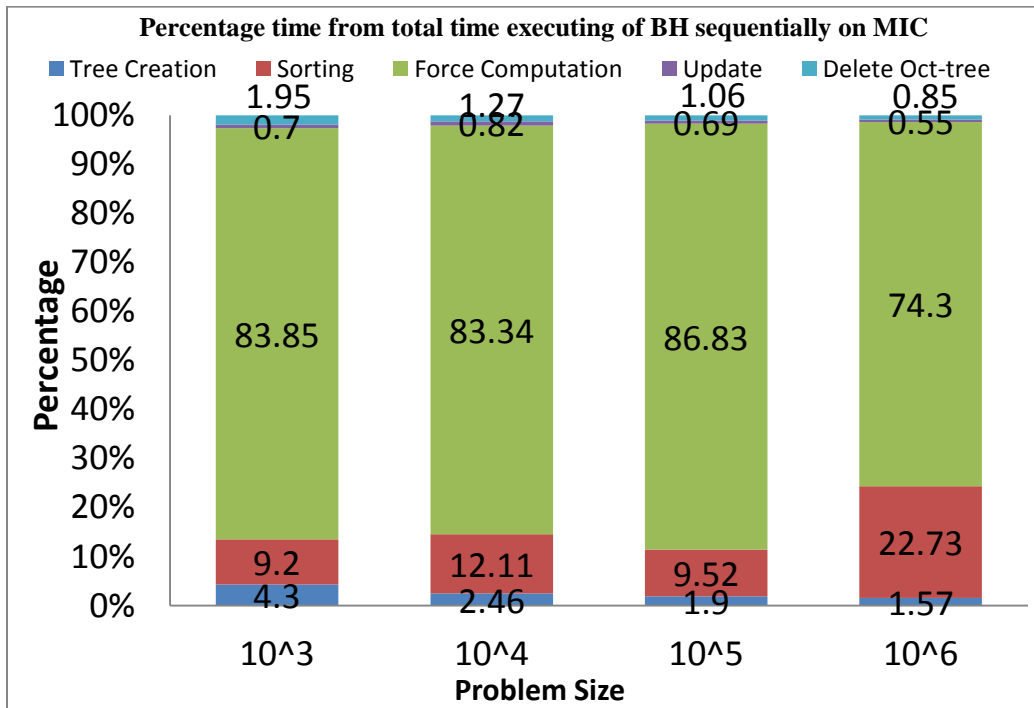
To estimate the overhead of my implementation I compare the number of data read and write that is generated in both experiments. For the locality checking, I used the VTune to check the number of L2 cache misses. On the other hand, for the load balancing I plot the Average Thread time (ATt), Thread Time minimum (Ttmin) and Thread Time max (Ttmax). This can show us how the time can change and linearity of the load balancing.

### 5.10.1 Algorithm Time Distribution

Experiments show that Force Computation is the hotspot step in N-body problem on MIC. Figure 5-14 shows that force computation of N-body simulation takes about 80% in the average of the total execution time. Which indicate that reducing the execution time of it; will affect the overall performance of the implementation. Table 5-2 illustrates percentage time of each step from total time of N-Body BH which is sequentially run using 1 thread.

**Table 5-2 Hotspot analysis of the Algorithm Steps**

Size of Problem	Percentage time from total time of Executing BH sequentially on MIC				
	Oct-tree Creation	Sorting	Compute Force	Update	Delete Oct-Tree
10 <sup>3</sup>	4.3	9.2	83.85	0.7	1.95
10 <sup>4</sup>	2.46	12.11	83.34	0.82	1.27
10 <sup>5</sup>	1.9	9.52	86.83	0.69	1.06
10 <sup>6</sup>	1.57	22.73	74.3	0.55	0.85



**Figure 5-14 Percentage Execution Time For N-body for Each Step**

To assess the efficiency ICZB on many core machines. Several Experiments are carried out. I run Experiments for different thread number 32 threads up to 240 threads with different problem size.

### 5.10.2 Speedup of STATIC and ICZB

Figure 5-15 shows the speedup of ICZB and static approach vs. different problem size. The speedup is plotted using 240 threads (60core). In addition to that I plotted average speedup (age), minimum speedup (min) and maximum speed up (max) for both of the approaches during 20 iteration of simulation.

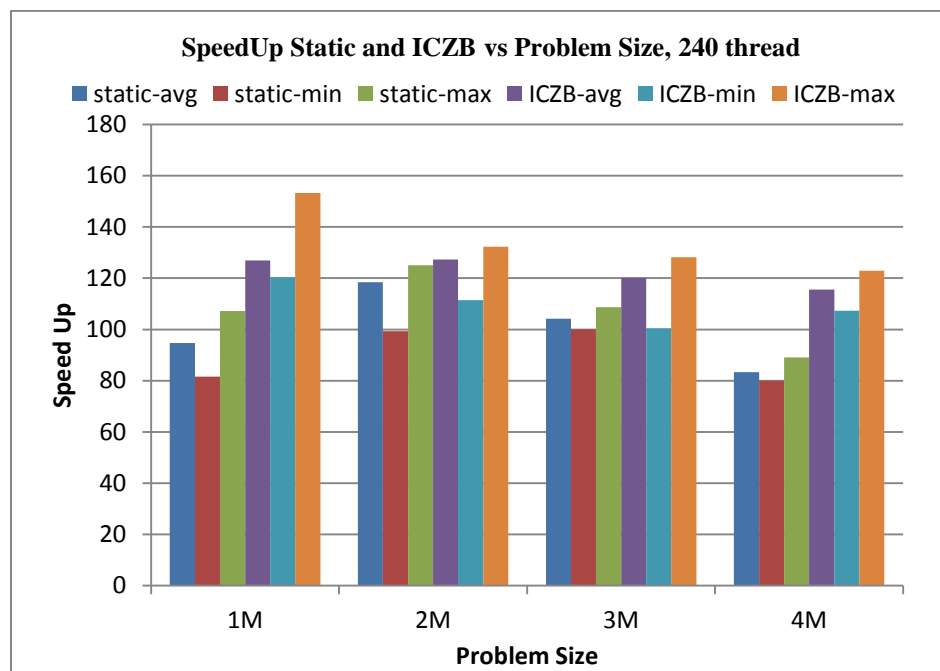


Figure 5-15 Speedup of STATIC and ICZB vs. Problem Size ( 1M,2M,3M,4M)

I found that speedup of ICZB and Static approaches decreases when the size of the problem increases, due to the data locality problem. However, the problem size increased and the data array increased. So, the percentage of fitting the sub-tree into the cache will

be decreased. In addition, to that the sharing of the data between the cores decreased. These two problems will increase the number of cache misses.

Also, I found that speedup of ICZB is better than STATIC with the change of the problem size. Particularly, Speed up of ICZB is always better if I compare the minimum, maximum and average speedup. I found that ICZB speed up is 42%, 36% better than STATIC respectively on problem size 1M, 4M.

### **5.10.3 Overhead of ICZB**

I show that my method has better speedup with respect to the static. But this is not enough to judge my work. I need to understand the overhead that my method adds to the static approach. For that, I use the VTune profiler to record the number of reads and writes that each approach did. Figure 5-16 presents the number of reads and writes with different number of threads and different problem size. In conclusion from the graphs; the overhead is small in both problem size and even when I increase the number of the problem size with factor of 10 it is still small. This is a promising result for my implementation than can shows how the effectiveness of my implementation.



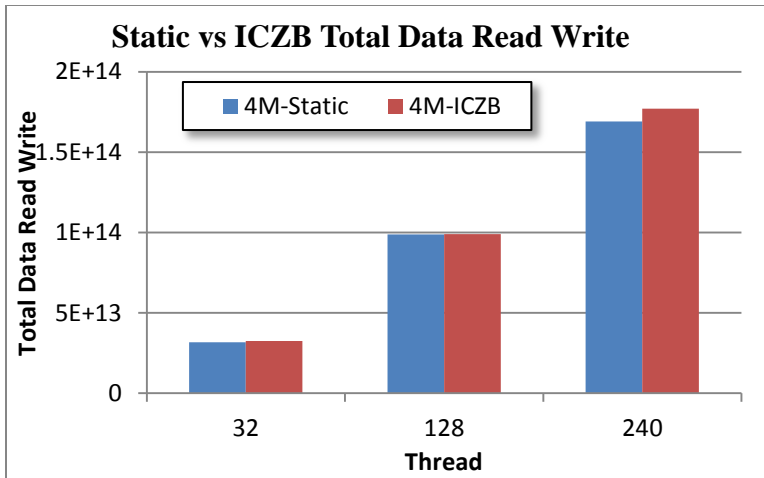


Figure 5-16 Data Read and Write for Static vs. ICZB for 4M bodies

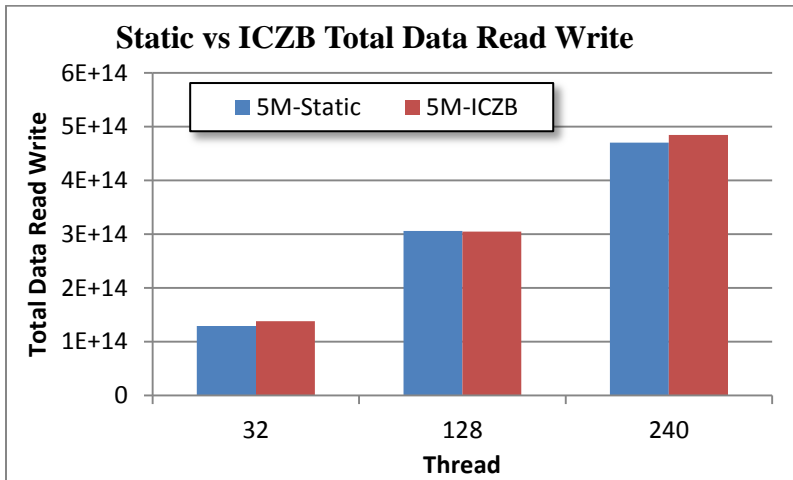


Figure 5-17 Data Read and Write for Static vs. ICZB for 5M bodies

### 5.10.4 Locality

Data locality is applied using Morton order. I use L2 cache misses that generated from Vtune in both approaches to test locality. Figure 5-18 illustrate the difference. ICZB has lower number of L2 cache misses.

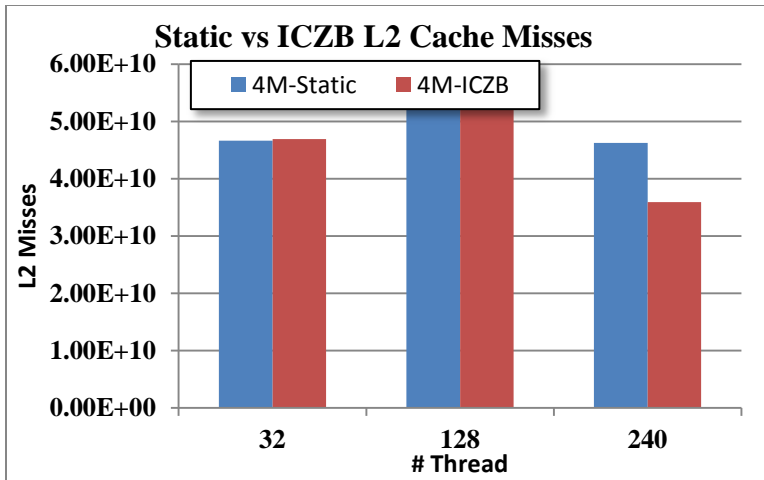


Figure 5-18 L2 cache misses for static and ICZB for 4M bodies

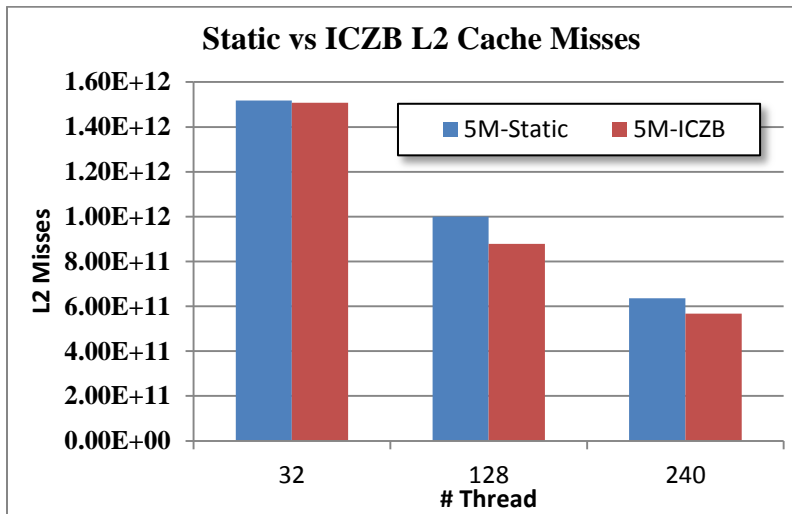


Figure 5-19 L2 cache misses for Static and ICZB for 5M bodies

### 5.10.5 Linearity and Effectiveness of Dynamic Load Balancing

Incorporating physical locality with dynamic load balancing does indeed lead to dramatically better performance. However, in static load balancing, since thread particles are physically clumped together, giving every thread an equal number of particles introduces structural load imbalances: A thread with particles in a dense region of the distribution has much more work to do than a thread with particles in a sparse region. So,

I can conclude that the bottleneck in the performance of static approach is from how to divide the work across the threads. In this section I will discuss the results of linearity on N-body simulation, max percentage of the average relative work deviation and max percentage of average time relative deviation for the ICZB approach on 5 million of bodies running using 240 threads.

Figure 5-20 includes linearity plot at the y-axis the average time of the threads and on the x-axis the average work of the threads for a given problem size.

I observe that the average time for threads increases with work for thread in two steps. The average time increases slowly until problem size 2M at the beginning. Then the average time increases rapidly. I conclude that memory resources of the machine exploited. In addition, the average work is linearly increased with respect to the average work.

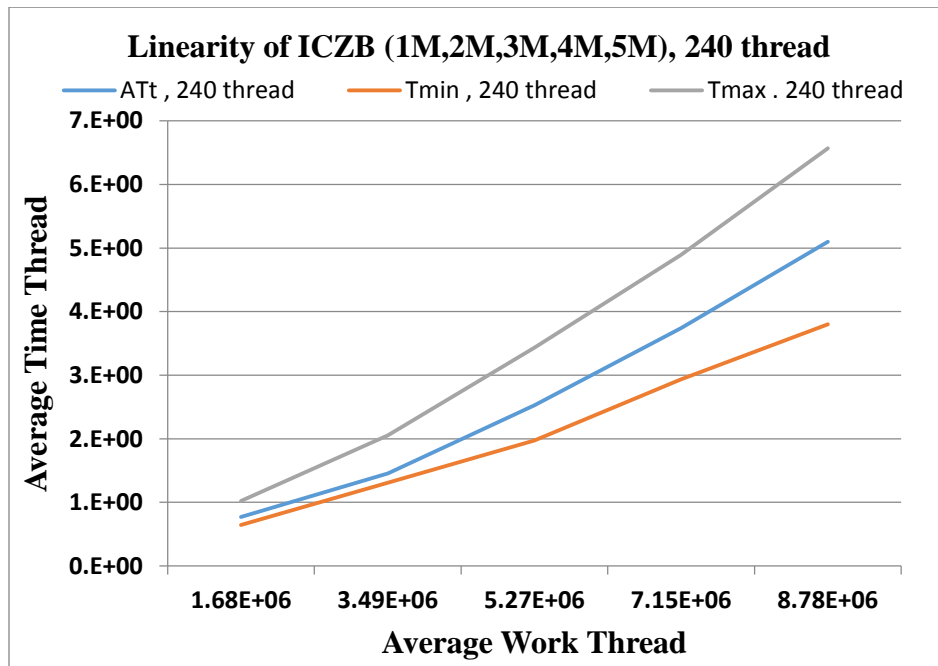


Figure 5-20 Linearity of ICZB (1M,2M,3M,4M,5M) 240 thread

To dig more for understanding the effectiveness of my method, a set of plots has been constructed for a given problem size. The percentage deviation of the thread time for the iteration is plotted to understand how much the time fluctuating from the average time. Also, the percentage deviation of the work for the iteration space is plotted to show how much the load balancing change from one iteration to another. In addition, the speedup for the same problem size is reported a cross the iteration space. This allows us to understand the effect of the deviation of the work and time on the speedup of my method.

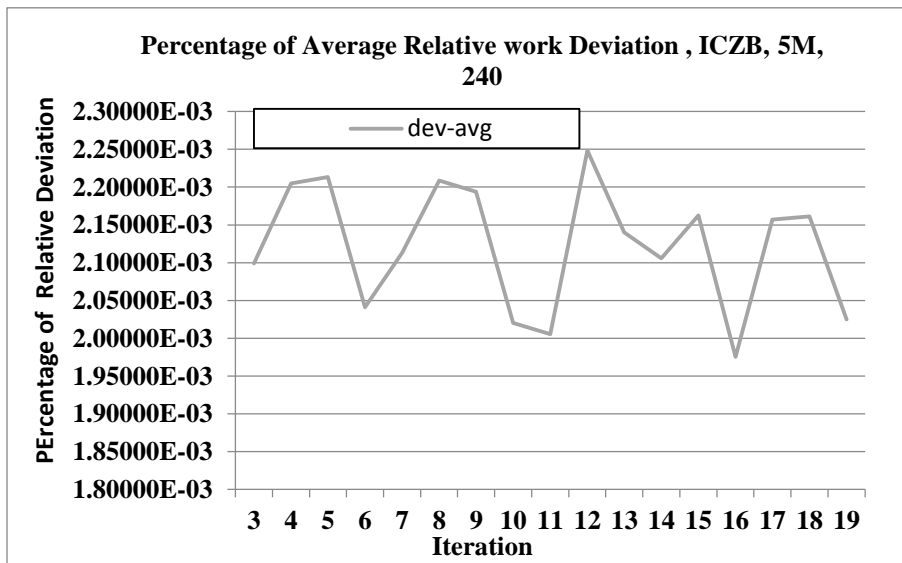


Figure 5-21 Percentage of average relative work deviation of ICZB, 5M, 240

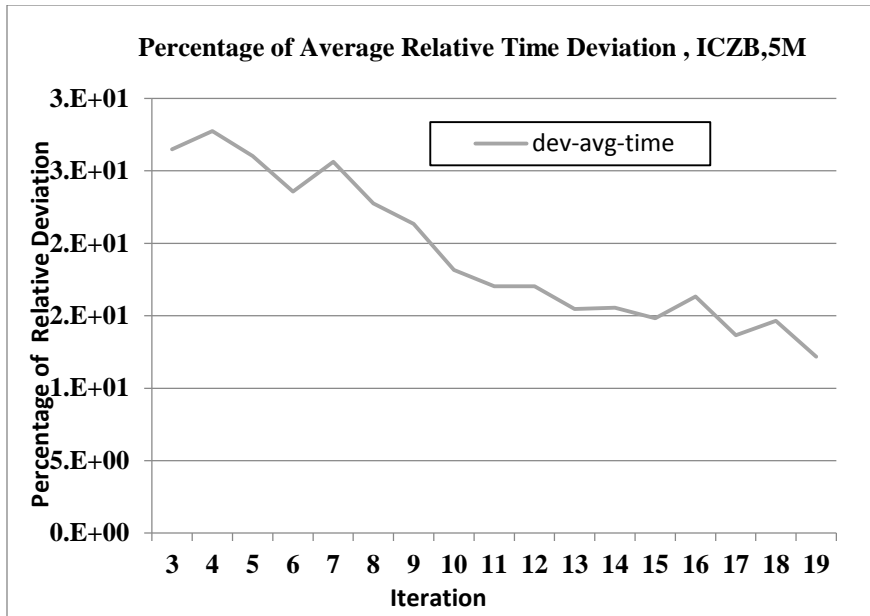


Figure 5-22 Percentage of Average Relative Time Deviation, ICZB, 5M

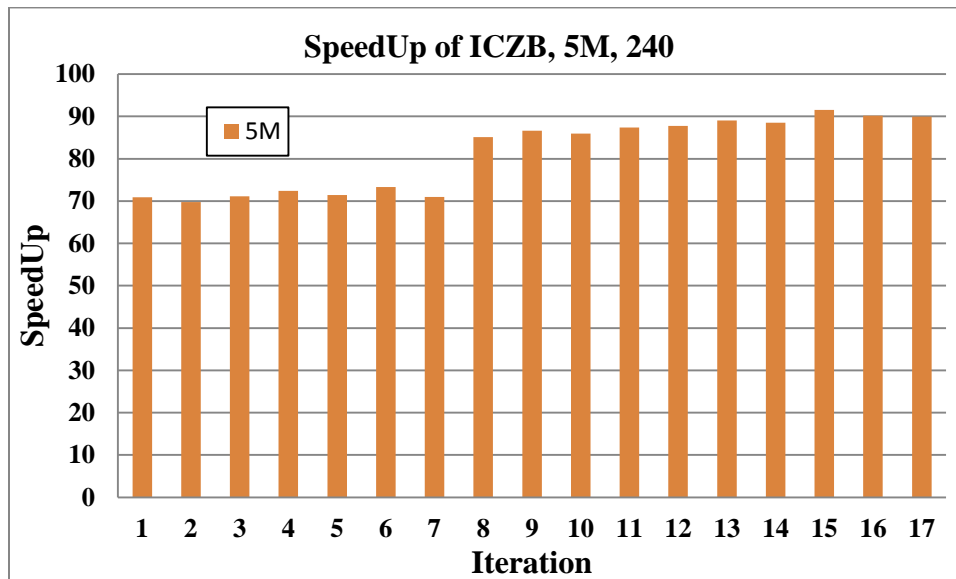


Figure 5-23 Speedup of ICZB, problem size 5M, 240

Figure 5-22 shows the percentage relative deviation of the work of the threads which is about 0.0002% from average work of the threads. This is a low value compared to the average work for the thread. Figure 5-22 shows the average relative time deviation which is decreased during the simulation process. I can conclude that the algorithm is adaptively

correct the amount of work distributed across the threads. However, Figure 5-23 explains this conclusion. From the figure, the speedup increased during the simulation, comparing it to the percentage relative deviation of the time it is decreased during the simulation, which means that my method is adaptively load balance the work in the simulation.

## **5.11 Conclusion**

The dynamic load balancing using ICZB was implemented combined with data locality for the N-body Simulation. The results show that the dynamic load balancing is an essential factor in increasing the force computation parallelism and therefore significantly reduces the computation time. I concentrate my work on two main challenges. The data locality and the dynamic load balancing. The implementation of the ICZB method along with the data structure suggested outperforms the static approach. I obtain a speedup always better than static approach vs the problem size. Also, the overhead of my method is low and cache misses decreased. In a conclusion, the ICZB is working better than static approach, but it needs to be improved more for the larger problem size. My implementation on MIC shows that the execution time and aggregate load scales linearly with the problem size when using 60 cores for problem sizes within the range of 1 million to 4 million. In addition, my DLB-BH provides an increased speedup of 42% and 36% on problem size 1 million and 4 million respectively, as compared to traditional Static Barnes Hut (S-BH). DLB is recommended as a compiler strategy as one optimization strategy for semi-static applications.

## **Appendix A: STRASSEN MATRIX-MATRIX**

### **MULTIPLICATION CODE**

- **Nature of the application**

The basic Strassen-MM (S-MM) algorithm time complexity is of  $O(N^{2.807})$  instead of  $O(N^3)$  of standard MM algorithm. It computes  $C=AxB$  where  $A$ ,  $B$  and  $C$  matrices of Size  $N \times N$ . It is a recursive algorithm where matrices are partitioned in each level. This matrices partition process can be done recursively until the sub matrices degenerate into numbers. In this code I implement the reorder approach of STRASSEN to reduce memory usage. I only used T1 and T2 as intermediate subMatrices ( See chapter 4 for more details).

- **Data Structure**

I used one dimensional array data structure in my implementation. I have matrix  $A$  and  $B$  as input and matrix  $C$  as output.

- **Procedures**

I have 3 main arithmetic operations procedures Addition, Subtraction and Multiplication in the implementation. Each Procedure takes 12 parameters described below as follows:

1. Matrix A,B as input and C as output.
2. Column index and Row index for Matrix A.
3. Column index and Row index for Matrix B.
4. Column index and Row index for Matrix C.
5. Size of Matrix A.
6. Size of Matrix B.
7. Size of Matrix C.

In addition to the a above explanation, each procedure has a comments inside the code to understand its structure and input and output of it.

- **Input and Output**

I have a procedure that initialize matrix A and B called “accuracyTestinit”. Therefore, I initialized the matrices randomly with double numbers.

- **Correctness**

To check correctness of my implementation. Each time I run the program the sequential MM is computed and compared to the results of the parallel implementation.



```

1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <time.h>
6  #include <string.h>
7  #include <sys/time.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10 #include "mkl.h"
11
12 /* This code implements the REORDER APPROACH OF STRASSEN MATRIX-MATRIX
13 MULTIPLICATION
14 USING OpenMP programming Model
15 It takes A and B as an input and produce C ----> C=A*B where A, B and C
16 matrices of Size NxN
17 * I implemented Strassen matrix matrix multiplication using reorder approach,
18 where we need only 2 intermediate matrices ( T1, T2).
19 This reduce the memory usage for Strassen.
20 * We have 3 main arithmetic operation Addition, Subtraction and
21 Multiplication
22 *Each Procedure takes 12 parameters described below
23 * Matrix A,B as input and C as output.
24 * Column index and Row index for Matrix A.
25 * Column index and Row index for Matrix B.
26 * Column index and Row index for Matrix C.
27 * Size of Matrix A.
28 * Size of Matrix B.
29 * Size of Matrix C.
30 *The matrix implemented as one Dimensional Array to increase performance.
31 * The commented parts in the code can be used to debug it.
32 */
33 int DIM_N =1024; // Matrix Size
34 int threads= 32; // Number of threads
35 int threshold= 1024; // Threshold value for Strassen Algorithm
36 //int mkl_threads= 16;
37
38 //other stuff
39 double sum, snorm;
40
41 //matrices
42 double *A, *B, *C,*CC;
43 //#####
44 #####
45 //##### Procedure Prototypes
46 #####
47 //#####
48 #####
49
50 //Prototypes for the 3 basic operations used in the implementation Addition,
51 subtraction and multiplication.
52 // Multiplication Procedure Prototypes
53 void
54 strassenMultMatrix(double*,double*,double*,int,int,int,int,int,int,int,int,int,
55 int);
56 void normalMultMatrix(double*, double*, double*,
57 int,int,int,int,int,int,int,int,int,int);
58 //Subtraction Procedure Prototypes
59 void subMatrices(double*, double*, double*, int,int,int,int,int,int,int,int);
60 void subMatrices1(double*, double*, double*, int,int,int,int,int,int,int,int);
61 void subMatricesc(double*, double*, double*,
62 int,int,int,int,int,int,int,int,int,int);
63 //Addition Procedures Prototypes

```

```

64 void addMatrices(double*, double*, double*, int,int,int,int,int,int,int,int);
65 void addMatrices1(double*, double*, double*, int,int,int,int,int,int,int,int);
66 void addMatricesc(double*, double*, double*,
67 int,int,int,int,int,int,int,int,int);
68 //This Procedure used for Debugging Purpose
69 void myprint(double *,char *,int,int ,int,int);
70
71 //#####
72 #####
73 //Error calculation
74 //#####
75 #####
76 // Procedure For Error Calculation
77
78 //#####
79 #####
80
81 void checkPracticalErrors (double *c, double *seq, int n)
82 {
83 int ii;
84 int n2 = n*n;
85 double sum =0;
86 double low = c[0] - seq[0];
87 double up = low;
88 for (ii=0; ii<n2; ii++)
89 {
90 double temp = c[ii] - seq[ii];
91 sum += (temp<0 ? -temp: temp);
92 if (temp > up)
93 up = temp;
94 else if (temp< low)
95 low = temp;
96 }
97 printf ("average error: %.20f\n", sum/n2);
98 printf ("lower-bound: %.20f\n", low);
99 printf ("upper-bound: %.20f\n", up);
100 printf ("\n");
101 }
102
103 void accuracyTestInit (double* a, double *b, int n)
104 {
105 int i,j;
106 double *uvT = malloc ( n*n*sizeof(double*) );
107 //initiate a and b
108 for (i =0 ; i< n; i++)
109 {
110 for (j =0; j< n; j++)
111 {
112 //int index = i*n+j;
113 a[i*n+j] = b[i*n+j] = (i==j?1.0f:0.0f);
114 }
115 }
116 double *u = malloc ( n*sizeof(double*) );
117 double *v = malloc ( n*sizeof(double*) );
118 //initiate u and v
119 for (i= 1; i< n+1; i++)
120 {
121 u[i-1] = 1.0f/(n+1.0f-i);
122 v[i-1] = sqrt(i);
123 }
124 //vTu
125 double vTu = 0.0f;
126 for (i= 0; i< n; i++)

```

```

127     {
128     vTu += u[i]*v[i];
129     }
130     double scalar = 1.0f/(1.0f+vTu);
131     //uvT
132     for (i= 0; i< n; i++)
133     {
134     for (j= 0; j< n; j++)
135     {
136     uvT[i*n+j] = u[i]*v[j];
137     }
138     }
139     //construct a and b
140     for (i=0; i< n; i++)
141     {
142     for (j= 0; j< n; j++)
143     {
144     int index = i*n+j;
145     a[i*n+j] += uvT[index];
146     b[i*n+j] -= scalar*uvT[index];
147     }
148     }
149     free (uvT);
150     free (u);
151     free (v);
152     }
153     //#####
154
155     //MAIN
156     int main (int argc, char *argv[]){
157         if(argc > 1)
158             DIM_N = atoi(argv[1]); // here to enter the size of the matrix
159         if(argc > 2)
160             threads = atoi(argv[2]); // here to enter the number of threads
161         if(argc > 3)
162             threshold = atoi(argv[3]); // here to enter the level of recursion
163         //if(argc > 4 )
164             //mkl_threads= atoi(argv[4]);
165
166
167         double etime=0.0,stime=0.0; // for
168         double dtime=0.0;
169         int i,j,k;
170         // double *A=malloc(N*N*sizeof(double));
171
172         A = malloc(sizeof(double)*DIM_N*DIM_N);
173
174         B = malloc(sizeof(double)*DIM_N*DIM_N);
175
176         C = malloc(sizeof(double)*DIM_N*DIM_N);
177
178         // CC = malloc(sizeof(double)*DIM_N*DIM_N);
179
180         accuracyTestInit(A,B, DIM_N); // To intiliza the matrices
181         //#####
182         //### Print the A , B matrices #####
183         //#####
184         //print out the result
185         //myprint(A,"A Matrix",DIM_N,0,0,DIM_N);
186         //myprint(B,"B Matrix",DIM_N,0,0,DIM_N);
187         // This is the sequantail Computation of Matrix multiplication we used it to
188         chech the correctness of Strassen implementation
189         /*printf("computing sequential\n");

```

```

190     stime=omp_get_wtime();
191         for(i=0; i<DIM_N; i++)
192             for(j=0; j<DIM_N; j++){
193                 CC[i*DIM_N+j] = 0;
194                 for(k=0; k < DIM_N; k++)
195                     CC[i*DIM_N+j] += A[i*DIM_N+k] * B[j*DIM_N+k];
196             }
197
198     etime=omp_get_wtime();
199     printf("computed sequential\n");
200     dtime=etime-stime;
201     printf("Sequential Time taken = %0.5f \n", dtime);*/
202
203     //printf("Num Threads = %d\n",threads);
204     //start timer
205     stime=omp_get_wtime();
206
207     //Strassen Multiplication
208     omp_set_num_threads(threads);
209     strassenMultMatrix(A,B,C,DIM_N,0,0,0,0,0,0,DIM_N,DIM_N,DIM_N); // Calling of
210     Strassen MM
211
212     //stop timer
213     etime=omp_get_wtime();
214
215     //calculate time taken
216     dtime=etime-stime;
217     printf("Strassen Time taken=          %0.5f \n",dtime);
218
219     //stime=omp_get_wtime();
220     //mkl_set_num_threads(threads);
221     //cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, DIM_N, DIM_N, DIM_N,
222     1, A, DIM_N, B, DIM_N, 0, CC, DIM_N);
223     //etime=omp_get_wtime();
224     //dtime=etime-stime;
225     // printf("MKL Time taken = %0.5f \n",dtime);
226
227
228     /******Triple Loop Multiplication, with OpenMP, for Comparison*****/
229     //start timer
230     /*stime=omp_get_wtime();
231
232     #pragma omp parallel shared(A,B,CC,chunk) private(i,j,k) num_threads(threads)
233     {
234         //multiplication process
235         #pragma omp for schedule(dynamic) nowait
236         for (j = 0; j < DIM_N; j++){
237             for (i = 0; i < DIM_N; i++){
238                 CC[i][j] = 0.0;
239                 for (k = 0; k < DIM_N; k++)
240                     CC[i][j] += A[i][k] * B[k][j];
241             }
242         }
243     }
244     //normalMultMatrix(A,B,C,DIM_N);*/
245
246     //stop timer
247     //etime=omp_get_wtime();
248
249
250     // dtime=etime-stime;
251     //printf("Non-Strassen Time taken = %0.3f \n", dtime);
252     /#####

```

```

253 //A
254 //#####
255 //stime=omp_get_wtime();
256
257 /*int result = 0;
258 int xx=0;
259     for(i=0; i < DIM_N; i++){
260         for(j=0; j < DIM_N; j++){
261             if(fabs(C[i*DIM_N+j]-CC[i*DIM_N+j])>0.0001){
262                 //printf("(%d, %d) : (%.20f, %.20f)\n", i, j, C[i][j], CC[i][j]);
263                 result = 1;
264                 xx++;
265                 //break;
266             }
267         }
268         //printf("\n");
269         //if(result == 1) break;
270     }
271
272     printf("\n\nPercentage Error =%.3f\n Error
273 cell=%d\n", (double)xx/(DIM_N*DIM_N),xx);
274     printf("Test %s\n", (result == 0) ? "Passed" : "Failed");
275     checkPracticalErrors(C, CC, DIM_N);*/
276 //myprint(A,"A Matrix",DIM_N,0,0);
277 //myprint(B,"B Matrix",DIM_N,0,0);
278 //myprint(C,"Strassen Algorithem",DIM_N,0,0,DIM_N);
279 //myprint(CC,"Sequantial Algorithem",DIM_N,0,0,DIM_N);
280 free(A);
281 free(B);
282 free(C);
283 }
284
285 void addMatrices(double *x, double *y, double *z, int size,int srow1 , int
286 scoll1,int srow2,int scol2 , int DIM0,int DIM1,int DIM2){
287 //performs a matrix addition operation, z=x+y
288     int i,j;
289     int index1,index2,index3;
290     #pragma omp parallel shared(x,y,z,srow1,sroll1,srow2,scol2,size)
291     private(i,j) num_threads(threads)
292     {
293         #pragma omp for schedule(static) nowait
294         for (i = 0; i < size; i++)
295         {
296             index1=i*DIM2;
297             index2=((i+srow1)*DIM0)+sroll1;
298             index3=((i+srow2)*DIM1)+scol2;
299             for (j = 0; j < size; j++)
300                 z[index1+j] = x[index2+j] + y[index3+j];
301         }
302     }
303
304 void addMatricesc(double *x, double *y, double *z, int size,int srow1 , int
305 scoll1,int srow2,int scol2,int srow3,int scol3,int DIM0,int DIM1,int DIM2){
306 //performs a matrix addition operation, z=x+y
307     int i,j;
308     int index1,index2,index3;
309     #pragma omp parallel
310     shared(x,y,z,srow1,sroll1,srow2,scol2,srow3,scol3,size) private(i,j)
311     num_threads(threads)
312     {
313         #pragma omp for schedule(static) nowait
314         for (i = srow3; i < size+srow3; i++)
315         {
316             index1=i*DIM2;

```

```

316         index2=((i-srow3+srow1)*DIM0)-scol3+scol1;
317         index3=((i-srow3+srow2)*DIM1)-scol3+scol2;
318         for (j = scol3; j < size+scol3; j++){
319             //printf("\n%.0f %.0f %.0f ",z[i][j],x[i-
320 srow3+srow1][j-scol3+scol1],y[i-srow3+srow2][j-scol3+scol2]);
321             z[index1+j] = x[index2+j] + y[index3+j];
322         }
323
324             //printf("\n printing from inside the funtion %.0f
325 %.0f %.0f ",z[i][j],x[i-srow3+srow1][j-scol3+scol1],y[i-srow3+srow2][j-
326 scol3+scol2]);
327         }
328     }
329 }
330 void addMatrices1(double *x, double *y, double *z, int size,int srow1 , int
331 scol1,int srow2,int scol2,int DIM0,int DIM1,int DIM2){
332 //performs a matrix addition operation, z=x+y
333     int i,j;
334     int index1,index2,index3;
335     #pragma omp parallel shared(x,y,z,srow1,scol1,srow2,scol2,size)
336 private(i,j) num_threads(threads)
337     {
338         #pragma omp for schedule(static) nowait
339         for (i = srow2; i < size+srow2; i++)
340         {
341             index1=i*DIM2;
342             index2=((i-srow2+srow1)*DIM0)-scol2+scol1;
343             index3=((i-srow2)*DIM1)-scol2;
344             for (j = scol2; j < size+scol2; j++)
345                 z[index1+j] = x[index2+j] + y[index3+j];
346         }
347     }
348 }
349
350 void subMatrices(double *x, double *y, double *z, int size , int srow1 , int
351 scol1,int srow2,int scol2,int DIM0,int DIM1,int DIM2){
352 //performs a matrix subtraction operation, z=x-y
353     int i,j;
354     int index1,index2,index3;
355     #pragma omp parallel shared(x,y,z,srow1,scol1,srow2,scol2,size)
356 private(i,j) num_threads(threads)
357     {
358         #pragma omp for schedule(static) nowait
359         for (i = 0; i < size; i++)
360         {
361             index1=i*DIM2;
362             index2=(i+srow1)*DIM0+scol1;
363             index3=((i+srow2)*DIM1)+scol2;
364             for (j = 0; j < size; j++)
365                 z[index1+j] = x[index2+j] - y[index3+j];
366         }
367     }
368 }
369 void subMatricesc(double *x, double *y, double *z, int size , int srow1 , int
370 scol1,int srow2,int scol2,int srow3,int scol3,int DIM0,int DIM1,int DIM2){
371 //performs a matrix subtraction operation, z=x-y
372     int i,j;
373     int index1,index2,index3;
374     #pragma omp parallel
375 shared(x,y,z,srow1,scol1,srow2,scol2,srow3,scol3,size) private(i,j)
376 num_threads(threads)
377     {
378         #pragma omp for schedule(static) nowait

```

```

379         for (i = srow3; i < size+srow3; i++)
380         {
381             index1=i*DIM2;
382             index2=(i-srow3+srow1)*DIM0-scol3+scol1;
383             index3=(i-srow3+srow2)*DIM1-scol3+scol2;
384             for (j = scol3; j < size+scol3; j++)
385                 z[index1+j] = x[index2+j] - y[index3+j];
386         }
387     }
388 }
389 void subMatrices1(double *x, double *y, double *z, int size , int srow1 , int
390 scol1,int srow2,int scol2,int DIM0,int DIM1,int DIM2){
391 //performs a matrix subtraction operation, z=x-y
392     int i,j;
393     int index1,index2,index3;
394     #pragma omp parallel shared(x,y,z,srow1,scol1,srow2,scol2,size)
395     private(i,j) num_threads(threads)
396     {
397         #pragma omp for schedule(static) nowait
398         for (i = srow2; i < size+srow2; i++)
399         {
400             index1=i*DIM2;
401             index2=(i-srow2+srow1)*DIM0-scol2+scol1;
402             index3=(i-srow2)*DIM1-scol2;
403             for (j = scol2; j < size+scol2; j++)
404                 z[index1+j] = x[index2+j] - y[index3+j];
405         }
406     }
407 }
408
409
410
411 void normalMultMatrix(double *x, double *y, double *z, int size,int srow1 , int
412 scol1,int srow2,int scol2,int srow3,int scol3,int DIM0,int DIM1,int DIM2)
413 {
414 //multiplies two matrices: z=x*y
415     //int i,j,k;
416
417     //#pragma omp parallel
418     shared(x,y,z,size,srow1,scol1,srow2,scol2,srow3,scol3,DIM0,DIM1,DIM2)
419     private(i,j,k) num_threads(threads)
420     //{
421         //multiplication process
422         //#pragma omp for schedule(static)
423         //for (i = srow3; i < size+srow3; i++){
424             //for (j = scol3; j < size+scol3; j++){
425                 //z[i*DIM2+j] = 0.0;
426                 //for (k = 0; k < size; k++)
427                     //{
428                         //z[i*DIM2+j] += x[(i-
429 srow3+srow1)*DIM0+(k+scol1)] * y[(k+srow2)*DIM1+(j-scol3+scol2)];
430 //cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, p, alpha, A, p,
431 B, n, beta, C, n);
432
433     mkl_set_num_threads(threads);
434     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, size,size ,size,1,
435 &x[srow1*DIM0+scol1], DIM0, &y[srow2*DIM1+scol2], DIM1, 0,
436 &z[srow3*DIM2+scol3], DIM2);
437 }
438
439 void myprint(double *xx,char *name,int size,int x,int y,int DIM)
440 {
441     int i,j;

```

```

442 printf("\nStart of Print %s\n",name);
443 for(i=0;i<size;i++)
444 {
445     for(j=0;j<size;j++)
446     {
447         printf("%.5f\t",xx[(i+x)*DIM+(j+y)]);
448     }
449     printf("\n");
450 }
451 printf("\nEnd of Print\n");
452 }
453
454
455 void strassenMultMatrix(double *a,double *b,double *c,int size,int srow1, int
456 scoll, int srow2 , int scol2 , int srow3 ,int scol3,int DIM0,int DIM1,int
457 DIM2){
458 //Performs a Strassen matrix multiply operation
459
460     double *t1, *t2;
461
462     int newsize = size/2;
463     int i;
464     //printf("\nindecies=%d\t\t %d %d %d %d %d
465 %d\n", size,srow1,scoll,srow2,scol2,srow3,scol3);
466
467     if (size >= threshold) {
468
469         t1 = malloc(sizeof(double*)*newsize*newsize);
470         t2 = malloc(sizeof(double*)*newsize*newsize);
471
472         //addMatrices(a11,a22,t1,newsize);
473         //addMatrices(b11,b22,t2,newsize);
474         // strassenMultMatrix(t1,t2,c21,newsize);
475         addMatrices(a,a,t1,newsize,srow1,scoll,newsize+srow1,newsize+scoll,DIM0,DIM0,ne
476 wsize);
477         //myprint(a,"print a11",newsize,srow1,scoll,size);
478         //myprint(a,"print a22",newsize,newsize+srow1,newsize+scoll,size);
479         //myprint(t1,"addition result of a11,a22",newsize,0,0,newsize);
480         addMatrices(b,b,t2,newsize,srow2,scol2,newsize+srow2,newsize+scol2,DIM1,DIM1,ne
481 wsize);
482         //myprint(t2,"addition result of b11,b22",newsize,0,0,newsize);
483         strassenMultMatrix(t1,t2,c,newsize,0,0,0,0,newsize+srow3,scol3,newsize,newsize,
484 DIM2);
485         //myprint(c,"Result Matrix of t1*t2 calculate
486 M1",newsize,newsize+srow3,scol3,DIM2);
487         //myprint(c,"All C matrix",DIM2,0,0,DIM2);
488
489
490         // subMatrices(a21,a11,t1,newsize);
491         subMatrices(a,a,t1,newsize,newsize+srow1,scoll,srow1,scoll,DIM0,DIM0,newsize);
492         //myprint(t1,"subtraction of a21,a11 ",newsize,0,0,newsize);
493
494         //addMatrices(b11,b12,t2,newsize);
495         addMatrices(b,b,t2,newsize,srow2,scol2,srow2,newsize+scol2,DIM1,DIM1,newsize);
496         //myprint(t2,"addition of b11,b12",newsize,0,0,newsize);
497
498         //strassenMultMatrix(t1,t2,c22,newsize);
499         strassenMultMatrix(t1,t2,c,newsize,0,0,0,0,newsize+srow3,newsize+scol3,ne
500 wsize,newsize,DIM2); //Calculate M6
501         //myprint(c,"Calculate M6 t1*t2",newsize,newsize+srow3,newsize+scol3,DIM2);
502         //myprint(c,"All C matrix",DIM2,0,0,DIM2);
503

```



```

504 subMatrices(a,a,t1,newsize,srow1,newsize+srow1,newsize+srow1,newsize+scoll,DIM0
505 ,DIM0,newsize);
506 //myprint(t1,"subtration a12,a22",newsize,0,0,newsize);
507 //addMatrices(b21,b22,t2,newsize);
508 addMatrices(b,b,t2,newsize,newsize+srow2,scol2,newsize+srow2,newsize+scol2,DIM1
509 ,DIM1,newsize);
510 //myprint(t2,"addition b21,b22",newsize,0,0,newsize);
511 //strassenMultMatrix(t1,t2,c11,newsize);
512 strassenMultMatrix(t1,t2,c,newsize,0,0,0,0,srow3,scol3,newsize,newsize,DIM2); //
513 calculate M7
514 //myprint(c,"calculate M7 t1*t2",newsize,srow3,scol3,DIM2);
515 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
516 //addMatrices(c11,c21,c11,newsize);
517 addMatricesc(c,c,c,newsize,srow3,scol3,newsize+srow3,scol3,srow3,scol3,DIM2,DIM
518 2,DIM2);
519 //myprint(c,"Problem Submatrix c11",newsize,srow3,scol3,DIM2);
520 //myprint(c,"Problem Submatrix c21",newsize,newsize+srow3,scol3,DIM2);
521 //myprint(c,"Addition the problem Start here C's sub matrix c11 ,
522 c21",newsize,srow3,scol3,DIM2);
523
524
525 //addMatrices(c21,c22,c22,newsize);
526 addMatricesc(c,c,c,newsize,newsize+srow3,scol3,newsize+srow3,newsize+scol3,news
527 ize+srow3,newsize+scol3,DIM2,DIM2,DIM2);
528 //myprint(c,"Addition C's sub matrix of
529 c21,c22",newsize,newsize+srow3,newsize+scol3,DIM2);
530 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
531
532 #####
533 #####
534 //addMatrices(a21,a22,t1,newsize);
535 addMatrices(a,a,t1,newsize,newsize+srow1,scol1,newsize+srow1,newsize+scoll,DIM0
536 ,DIM0,newsize);
537 //myprint(t1,"Additon a21 , a22",newsize,0,0,newsize);
538 //strassenMultMatrix(t1,b11,c21,newsize);
539 strassenMultMatrix(t1,b,c,newsize,0,0,srow2,scol2,newsize+srow3,scol3,newsize,D
540 IM1,DIM2); // Compute M2
541 //myprint(c,"Calculate M2 t1*b11",newsize,newsize+srow3,scol3,DIM2);
542 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
543
544
545 //subMatrices(b12,b22,t2,newsize);
546 subMatrices(b,b,t2,newsize,srow2,newsize+scol2,newsize+srow2,newsize+scol2,DIM1
547 ,DIM1,newsize);
548 //myprint(t2,"Subtration b12,b22",newsize,0,0,newsize);
549 //strassenMultMatrix(a11,t2,c12,newsize)
550 strassenMultMatrix(a,t2,c,newsize,srow1,scol1,0,0,srow3,newsize+scol3,DIM0,news
551 ize,DIM2); //Compute M3
552 //myprint(c,"Calculate M3 a11*t2",newsize,srow3,newsize+scol3,DIM2);
553 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
554 //subMatrices(c22,c21,c22,newsize);
555 subMatricesc(c,c,c,newsize,newsize+srow3,newsize+scol3,newsize+srow3,scol3,news
556 ize+srow3,newsize+scol3,DIM2,DIM2,DIM2);
557 //myprint(c,"Subtraction C of
558 c22*c21",newsize,newsize+srow3,newsize+scol3,DIM2);
559 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
560 // addMatrices(c22,c12,c22,newsize);
561 addMatricesc(c,c,c,newsize,newsize+srow3,newsize+scol3,srow3,newsize+scol3,news
562 ize+srow3,newsize+scol3,DIM2,DIM2,DIM2);
563 //myprint(c,"Addition C of c22*c12",newsize,newsize+srow3,newsize+scol3,DIM2);
564 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
565
566 #####

```

```

567     //subMatrices(b21,b11,t2,newsize);
568 subMatrices(b,b,t2,newsize,newsize+srow2,scol2,srow2,scol2,DIM1,DIM1,newsize);
569 //myprint(t2,"Subtraction t2 of b21 ,b11",newsize,0,0,newsize);
570     //strassenMultMatrix(a22,t2,t1,newsize);
571 strassenMultMatrix(a,t2,t1,newsize,newsize+srow1,newsize+scol1,0,0,0,0,DIM0,new
572 size,newsize);//compute M4
573 //myprint(t1,"Calculate M4 a22*t2",newsize,0,0,newsize);
574
575     //addMatrices(c11,t1,c11,newsize);
576 addMatrices1(c,t1,c,newsize,srow3,scol3,srow3,scol3,DIM2,newsize,DIM2);
577 //myprint(c,"Addition 1 C of c11,t1 ",newsize,srow3,scol3,DIM2);
578 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
579     // addMatrices(c21,t1,c21,newsize);
580 addMatrices1(c,t1,c,newsize,newsize+srow3,scol3,newsize+srow3,scol3,DIM2,newsize,
581 e,DIM2);
582 //myprint(c,"Addition C of c21, t1",newsize,newsize+srow3,scol3,DIM2);
583 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
584
585     //addMatrices(a11,a12,t1,newsize);
586 addMatrices(a,a,t1,newsize,srow1,scol1,srow1,newsize+scol1,DIM0,DIM0,newsize);
587 //myprint(t1,"Additon t1 , a11,a12",newsize,0,0,newsize);
588     //strassenMultMatrix(t1,b22,t2,newsize);
589 strassenMultMatrix(t1,b,t2,newsize,0,0,newsize+srow2,newsize+scol2,0,0,newsize,
590 DIM1,newsize);
591 //myprint(t2,"Strassen Matrix Multiplication t1*b11",newsize,0,0,newsize);
592
593     //subMatrices(c11,t2,c11,newsize);
594 subMatrices1(c,t2,c,newsize,srow3,scol3,srow3,scol3,DIM2,newsize,DIM2);
595 //myprint(c,"Subtraction 1 C c11-t2",newsize,srow3,scol3,size);
596     //addMatrices(c12,t2,c12,newsize);
597 //myprint(c,"All C matrix",8,0,0,8);
598 addMatrices1(c,t2,c,newsize,srow3,newsize+scol3,srow3,newsize+scol3,DIM2,newsize,
599 e,DIM2);
600 //myprint(c,"Addition C c12,t2",newsize,srow3,newsize+scol3,DIM2);
601 //myprint(c,"All C matrix",DIM2,0,0,DIM2);
602
603
604     free(t1);free(t2);
605 }
606 else {
607
608 normalMultMatrix(a,b,c,size,srow1,scol1,srow2,scol2,srow3,scol3,DIM0,DIM1,DIM2)
609 ;
610 }

```

## Appendix B: JACOBI SOLVER

- **Nature of the application**

JACOBI is an iterative method used to solve a Linear System Equation  $AX=B$  with number of equations equal  $N$ . It start with an initial solution  $X^0$  and computes the  $X^{k+1}$  for  $k$  times of iteration. Any iteration  $k$  needs all the values of  $X$  from iteration  $k-1$  except the values of  $x_i$ . I implemented 3 versions of JACOBI. Synchronous Jacobi, Asynchronous Jacobi and Relaxed Jacobi ( refer to chapter 4, section 4.3 for more details).

- **Data Structure**

I used one dimensional array data structure in my implementation. Where  $A$  is a matrix of size  $N \times N$  and  $X$  and  $B$  a vector of size  $N$ .

- **Procedures**

I have 3 files separated from each other. Each code is commented to simplify explaining.

- **Input and Output**

I have a procedure that initialize matrix A, B and X called “`randomInit`”. It is used to guarantee that the solution X will converge.

- **Correctness**

To check correctness of my implementation. Each time I run the program the sequential JACOBI SOLVER is computed and compared to the results of the parallel implementation.

```

1 // SYNCHRONOUSE JACOBI SOLVER
2 #include <stdio.h>
3 #include <math.h>
4 #include <unistd.h>
5 #include <time.h>
6 #include <sys/time.h>
7 #include <math.h>
8 #include <stdlib.h>
9 #include <omp.h>
10
11
12 #define MAX_ITER 100
13 #define ERR_THRESHOLD 0.00001
14
15
16
17 int N = 8;
18 int T = 2;
19 int rows_size=2;
20 int cols_size=2;
21 //#####
22 #####
23 //##### Intialization Funtions and checking of the
24 errors#####
25 //#####
26 #####
27 // Procedure used to intialixe the matrices A,B and X
28 void randomInit(double *A,double *X,double *B,int wA)
29 {
30     int i,j;
31     for(i = 0; i < wA; i++)
32     {
33         for(j = 0; j < wA; j++)
34         {
35             if (i==j)
36             {
37                 A[i*wA+j] = wA;
38             }
39             else
40             {
41                 A[i*wA+j] = -1 ;
42             }
43         }
44
45         X[i] = 0;
46         B[i] = 1;
47     }
48 }
49
50
51 void print1d(double *a,int N)
52 {
53     printf("\nStart printing\n");
54     int i,j;
55     for(i=0;i<N;i++)
56     {
57         printf("%.5f\n",a[i]);
58     }
59     printf("\nEnd printing\n");
60 }
61 void print2d(double *a,int N)
62 {
63     int i,j;

```

```

64 printf("\nStart printing\n");
65     for(i=0;i<N;i++)
66     {
67         for(j=0;j<N;j++)
68         {
69             printf("%.5f\t",a[i*N+j]);
70         }
71     printf("\n");
72     }
73 printf("\nEnd printing\n");
74 }
75
76
77
78 int main(int argc, char *argv[]){
79
80 int iter=1;
81     if(argc > 1)
82
83         N = atoi(argv[1]);
84
85     if(argc > 2)
86
87         T = atoi(argv[2]);
88     if(argc > 3)
89         rows_size=atoi(argv[3]);
90     if(argc > 4)
91         cols_size=atoi(argv[4]);
92
93
94
95
96     double sum;
97
98     int i,il, j, k , ii , jj;
99     int kk=N/T; // to compute the size of the rows for each thread.
100    double *A=malloc(sizeof(double)*N*N);
101    double *B=(double*)malloc(N*sizeof(double));
102    double *X=(double*)malloc(3*N*sizeof(double)); // this is to store X at
103    the intialization , at K , K+1
104    //double *XX=(double*)malloc(N*sizeof(double)); // this is to store
105    result X that produced from K iteration which will be referenced by the mod2=0
106    //double *XXX=(double*)malloc(N*sizeof(double)); // this is to store
107    result X that produced from K+1 iteration which will be referenced by mod2==1
108    //double *Xp; // to store the value of X for each thread
109    double *Xnew_sub; // to store the new value that is computed by each
110    thread
111    double *new_x=(double*)malloc(N*sizeof(double));
112    double *X_seq=(double*)malloc(N*sizeof(double));
113
114
115    //accuracyTestInit2D(A,N);
116    //accuracyTestInit1D(B,N);
117    //accuracyTestInit1D(X,N);
118    //accuracyTestInit1D(X_seq,N);
119    /*for(i=0;i<N;i++)
120    {
121        for(j=0;j<N;j++)
122        {
123            A[i*N+j]=sqrt(i+j)*0.2546;
124        }
125    }
126    for(i=0;i<N;i++)

```

```

127 {
128 B[i]=sqrt(i*j);
129 }*/
130 randomInit(A,B,X,N);
131 for(i=0;i<N;i++)
132 {
133 X_seq[i]=X[i];
134 }
135
136
137
138
139 double dtime=0.0;
140
141 double etime=0.0, stime=0.0;
142
143
144
145 // here to compute the Sequential version
146 //printf("\nstart computing Sequential jacobi....\n");
147
148 stime = omp_get_wtime();
149 for(k = 0; k < MAX_ITER; k++){
150
151     for(i=0; i<N; i++){
152         sum = 0.0;
153         sum=sum-(A[i*N+i] * X_seq[i]);
154
155         for(j=0; j<N; j++){
156
157             sum =sum + (A[i*N+j] * X_seq[j]);
158         }
159
160         new_x[i] = (B[i] - sum)/A[i*N+i];
161     }
162
163     for(i=0; i < N; i++)
164         X_seq[i] = new_x[i];
165 }
166 etime = omp_get_wtime();
167
168 dtime = etime - stime;
169
170 printf("\ncomputing sequential for 1 dimention=      %.5f\t", dtime);
171
172
173
174
175 // Optimized Parallel version of jacobi using blocking
176 // *****
177 // *****
178 omp_set_num_threads(T);
179 dtime=0.0;
180 int kkk;
181
182 for(kkk=0;kkk<iter;kkk++)
183 {
184 //accuracyTestInit1D(X,N);
185 stime = omp_get_wtime();
186
187 #pragma omp parallel shared (A,B,X,N,T,kk,rows_size,cols_size)
188 private(k,i,ii,j,sum,Xnew_sub)
189 {

```

```

190         //Xp=(double*)malloc(N*sizeof(double));
191         Xnew_sub=(double*)malloc((kk)*sizeof(double)*2);//we multiply it by 2 to
192 store for K and for K+1
193         int tid=omp_get_thread_num();
194         ii=tid*kk;// index of the rows that related to the the thread
195         int i_n=kk/rows_size;//number of rows chunk
196         int j_n=N/cols_size;//number of column chunk
197         int iii,jjj;
198
199     for(k=0;k<MAX_ITER;k++)
200     {
201
202         for(i=0;i<kk;i++)
203             Xnew_sub[i]=0;
204         // #pragma omp barrier // here is the most appropriate palce to put
205 the pragma to insure the consistency of the result
206         /*for(i=0;i<N;i++)
207         {
208             Xp[i]=X[i];
209         } */
210
211     for(iii=0;iii<i_n;iii++)
212     {
213         for(jjj=0;jjj<j_n;jjj++)
214         {
215             for(i=0; i<rows_size; i++){
216                 sum=0.0;
217                 //sum = sum-(A[(i+ii+iii)*N+i+ii+iii]*Xp[i+ii+iii]);
218
219                 for(j=0; j<cols_size; j++){
220                     sum+= A[(i+ii+(iii*rows_size))*N+j+(jjj*cols_size)] *
221 X[j+(jjj*cols_size)];
222                 }
223
224                 //Xnew_sub[i+iii]+= (B[i+ii+iii] -
225 sum)/A[(i+ii+iii)*N+i+ii+iii];
226                 Xnew_sub[i+(iii*rows_size)]+=sum;
227             }
228             //end of jjj
229             for(i=0;i<rows_size;i++)
230             {
231                 Xnew_sub[i+(iii*rows_size)]=Xnew_sub[i+(iii*rows_size)]-
232 (A[(i+ii+(iii*rows_size))*N+i+ii+(iii*rows_size)]*X[i+ii+(iii*rows_size)]);
233 Xnew_sub[i+(iii*rows_size)]=(B[i+ii+(iii*rows_size)]-
234 Xnew_sub[i+(iii*rows_size)])/A[(i+ii+(iii*rows_size))*N+i+ii+(iii*rows_size)];
235             }
236
237         }
238     }
239     #pragma omp barrier //here is incorret add of the barrier , the barrier
240 must be added before reading the data to update Xp
241     for(i=0;i<kk;i++){
242         X[i+ii]=Xnew_sub[i];
243     }
244 }
245 //end of the iteration MAX_ITER
246
247 free(Xnew_sub);
248 }
249 //parallel pragma
250
251 etime=omp_get_wtime();
252 dtime+=(etime-stime);
253 }

```



```

253 printf("parallel for 1D optimized blocked jacobi= \t%5f", (dtime/iter));
254
255 //printf("X_seq,N");
256 //printf("X,N");
257
258 // Here we use test the correctness of our implementation by comparing it to
259 the result of sequential code
260 int result = 0;
261     for(i=0; i < N; i++){
262         if(fabs(X[i]-X_seq[i]) > ERR_THRESHOLD)
263             {
264                 printf("(%d) : (%.5f,%.5f)\n", i, X[i], X_seq[i]);
265                 result = 1;
266             }
267         //if(result == 1) break;
268     }
269
270     printf("\tTest %s", (result == 0) ? "Passed\n" : "Failed\n");
271
272
273 free(A);
274 free(B);
275 free(X);
276 free(X_seq);
277 free(new_x);
278
279     return 0;
280
281 }
1

```

```

1 //ASYNCHRONOUSE JACOBI SOLVER
2 #include <stdio.h>
3 #include <math.h>
4 #include <unistd.h>
5 #include <time.h>
6 #include <sys/time.h>
7 #include <math.h>
8 #include <stdlib.h>
9 #include <omp.h>
10
11 #define MAX_ITER 100
12 #define ERR_THRESHOLD 0.00001
13
14 int N = 8;
15 int T = 2;
16 int rows_size=2;
17 int cols_size=2;
18 //#####
19 #####
20 //##### Intialization Funtions and checking of the
21 errors#####
22 //#####
23 #####
24
25 void checkPracticalErrors (double *c, double *seq, int n)
26 {
27 int ii;
28 int n2 = n*n;
29 double sum =0;
30 double low = c[0] - seq[0];
31 double up = low;
32 for (ii=0; ii<n2; ii++)
33 {
34 double temp = c[ii] - seq[ii];
35 sum += (temp<0 ? -temp: temp);
36 if (temp > up)
37 up = temp;
38 else if (temp< low)
39 low = temp;
40 }
41 printf ("average error: %.20f\n", sum/n2);
42 printf ("lower-bound: %.20f\n", low);
43 printf ("upper-bound: %.20f\n", up);
44 printf ("\n");
45 }
46
47 void accuracyTestInit2D(double* a, int n)
48 {
49 int i,j;
50 double *uvT = (double *) malloc ( n*n*sizeof(double) );
51 //initiate a and b
52 for (i =0 ; i< n; i++)
53 {
54 for (j =0; j< n; j++)
55 {
56 //int index = i*n+j;
57 a[i*n+j] = (i==j?1.0f:0.0f);
58 }
59 }
60 double *u = (double *) malloc ( n*sizeof(double) );
61 double *v = (double *) malloc ( n*sizeof(double) );
62 //initiate u and v
63 for (i= 1; i< n+1; i++)

```

```

64         {
65             u[i-1] = 1.0f/(n+1.0f-i);
66             v[i-1] = sqrt(i);
67         }
68 //vTu
69 double vTu = 0.0f;
70     for (i= 0; i< n; i++)
71     {
72         vTu += u[i]*v[i];
73     }
74     double scalar = 1.0f/(1.0f+vTu);
75 //uvT
76     for (i= 0; i< n; i++)
77     {
78         for (j= 0; j< n; j++)
79         {
80             uvT[i*n+j] = u[i]*v[j];
81         }
82     }
83 //construct a and b
84     for (i=0; i< n; i++)
85     {
86         for (j= 0; j< n; j++)
87         {
88             int index = i*n+j;
89             a[index] += uvT[index];
90         }
91     }
92 free (uvT);
93 free (u);
94 free (v);
95 }
96
97 void accuracyTestInit1D(double* a, int n)
98 {
99     int i,j;
100 double *uvT = (double *) malloc ( n*n*sizeof(double) );
101 //initiate a and b
102     for (i =0 ; i< n; i++)
103     {
104         for (j =0; j< n; j++)
105         {
106             //int index = i*n+j;
107             a[i] = (i==j?1.0f:0.0f);
108         }
109     }
110 double *u = (double *) malloc ( n*sizeof(double) );
111 double *v = (double *) malloc ( n*sizeof(double) );
112 //initiate u and v
113     for (i= 1; i< n+1; i++)
114     {
115         u[i-1] = 1.0f/(n+1.0f-i);
116         v[i-1] = sqrt(i);
117     }
118 //vTu
119 double vTu = 0.0f;
120     for (i= 0; i< n; i++)
121     {
122         vTu += u[i]*v[i];
123     }
124     double scalar = 1.0f/(1.0f+vTu);
125 //uvT
126     for (i= 0; i< n; i++)

```

```

127         {
128             for (j= 0; j< n; j++)
129             {
130                 uvT[i] = u[i]*v[j];
131             }
132         }
133 //construct a and b
134         for (i=0; i< n; i++)
135         {
136             for (j= 0; j< n; j++)
137             {
138                 int index = i*n+j;
139                 a[i] += uvT[index];
140             }
141         }
142 free (uvT);
143 free (u);
144 free (v);
145 }
146 void randomInit(double *A,double *X,double *B,int wA)
147 {
148     int i,j;
149     for(i = 0; i < wA; i++)
150     {
151         for(j = 0; j < wA; j++)
152         {
153             if (i==j)
154             {
155                 A[i*wA+j] = wA;
156             }
157             else
158             {
159                 A[i*wA+j] = -1 ;
160             }
161         }
162
163         X[i] = 0;
164         B[i] = 1;
165     }
166 }
167
168 void print1d(double *a,int N)
169 {
170     printf("\nStart printing\n");
171     int i,j;
172     for(i=0;i<N;i++)
173     {
174         printf("%.5f\n",a[i]);
175     }
176     printf("\nEnd printing\n");
177 }
178 void print2d(double *a,int N)
179 {
180     int i,j;
181     printf("\nStart printing\n");
182     for(i=0;i<N;i++)
183     {
184         for(j=0;j<N;j++)
185         {
186             printf("%.5f\t",a[i*N+j]);
187         }
188     }
189     printf("\n");

```

```

190     }
191     printf("\nEnd printing\n");
192 }
193
194
195
196 int main(int argc, char *argv[]){
197
198     int iter=1;
199     if(argc > 1)
200         N = atoi(argv[1]);
201
202     if(argc > 2)
203         T = atoi(argv[2]);
204     if(argc > 3)
205         rows_size=atoi(argv[3]);
206     if(argc > 4)
207         cols_size=atoi(argv[4]);
208
209
210
211
212
213
214     double sum;
215
216     int i,il, j, k , ii , jj;
217     int kk=N/T; // to compute the size of the rows for each thread.
218     double *A=malloc(sizeof(double)*N*N);
219     double *B=(double*)malloc(N*sizeof(double));
220     double *X=(double*)malloc(3*N*sizeof(double)); // this is to store X at
221 the intialization , at K , K+1
222     //double *XX=(double*)malloc(N*sizeof(double)); // this is to store
223 result X that produced from K iteration which will be referenced by the mod2=0
224     //double *XXX=(double*)malloc(N*sizeof(double)); // this is to store
225 result X that produced from K+1 iteration which will be referenced by mod2==1
226     //double *Xp; // to store the value of X for each thread
227     double *Xnew_sub; // to store the new value that is computed by each
228 thread
229     double *new_x=(double*)malloc(N*sizeof(double));
230     double *X_seq=(double*)malloc(N*sizeof(double));
231
232
233 //accuracyTestInit2D(A,N);
234 //accuracyTestInit1D(B,N);
235 //accuracyTestInit1D(X,N);
236 //accuracyTestInit1D(X_seq,N);
237 /*for(i=0;i<N;i++)
238 {
239     for(j=0;j<N;j++)
240     {
241         A[i*N+j]=sqrt(i+j)*0.2546;
242     }
243 }
244 for(i=0;i<N;i++)
245 {
246     B[i]=sqrt(i*j);
247 }*/
248 randomInit(A,B,X,N);
249 for(i=0;i<N;i++)
250 {
251     X_seq[i]=X[i];
252 }

```

```

253
254
255
256
257     double dtime=0.0;
258
259     double etime=0.0, stime=0.0;
260
261
262
263 // here to compute the Sequential version
264 //printf("\nstart computing Sequential jacobi....\n");
265
266 stime = omp_get_wtime();
267 for(k = 0; k < MAX_ITER; k++){
268
269     for(i=0; i<N; i++){
270         sum = 0.0;
271         sum=sum-(A[i*N+i] * X_seq[i]);
272
273         for(j=0; j<N; j++){
274
275             sum =sum + (A[i*N+j] * X_seq[j]);
276         }
277
278         new_x[i] = (B[i] - sum)/A[i*N+i];
279     }
280
281     for(i=0; i < N; i++)
282         X_seq[i] = new_x[i];
283 }
284 etime = omp_get_wtime();
285
286     dtime = etime - stime;
287
288 printf("\ncomputing sequential for 1 dimention=      %.5f\t", dtime);
289
290
291
292
293 // Optimized Parallel version of jacobi using blocking
294 // *****
295 // *****
296 omp_set_num_threads(T);
297 dtime=0.0;
298 int kkk;
299
300 for(kkk=0; kkk<iter; kkk++)
301 {
302 //accuracyTestInit1D(X,N);
303 stime = omp_get_wtime();
304
305 #pragma omp parallel shared (A,B,X,N,T, kk, rows_size, cols_size)
306 private (k, i, ii, j, sum, Xnew_sub)
307 {
308     //Xp=(double*)malloc(N*sizeof(double));
309     Xnew_sub=(double*)malloc((kk)*sizeof(double)*2); //we multiply it by 2 to
310 store for K and for K+1
311     int tid=omp_get_thread_num();
312     ii=tid*kk; // index of the rows that related to the the thread
313     int i_n=kk/rows_size; //number of rows chunk
314     int j_n=N/cols_size; //number of column chunk
315     int iii, jjj;

```

```

316
317 for(k=0;k<MAX_ITER;k++)
318 {
319
320     for(i=0;i<kk;i++)
321         Xnew_sub[i]=0;
322     // #pragma omp barrier // here is the most appropriate place to put
323 the pragma to insure the consistency of the result
324     /*for(i=0;i<N;i++)
325     {
326         Xp[i]=X[i];
327     } */
328
329 for(iii=0;iii<i_n;iii++)
330 {     for(jjj=0;jjj<j_n;jjj++)
331     {
332         for(i=0; i<rows_size; i++){
333             sum=0.0;
334             //sum = sum-(A[(i+ii+iii)*N+i+ii+iii]*Xp[i+ii+iii]);
335
336             for(j=0; j<cols_size; j++){
337                 sum+= A[(i+ii+(iii*rows_size))*N+j+(jjj*cols_size)] *
338 X[j+(jjj*cols_size)];
339             }
340
341             //Xnew_sub[i+iii]= (B[i+ii+iii] -
342 sum)/A[(i+ii+iii)*N+i+ii+iii];
343             Xnew_sub[i+(iii*rows_size)]+=sum;
344         }
345     } //end of jjj
346     for(i=0;i<rows_size;i++)
347     {
348 Xnew_sub[i+(iii*rows_size)]=Xnew_sub[i+(iii*rows_size)]-
349 (A[(i+ii+(iii*rows_size))*N+i+ii+(iii*rows_size)]*X[i+ii+(iii*rows_size)]);
350 Xnew_sub[i+(iii*rows_size)]=(B[i+ii+(iii*rows_size)]-
351 Xnew_sub[i+(iii*rows_size)])/A[(i+ii+(iii*rows_size))*N+i+ii+(iii*rows_size)];
352
353     }
354
355 } //end of iii
356     // #pragma omp barrier
357     for(i=0;i<kk;i++){
358         X[i+ii]=Xnew_sub[i];
359     }
360
361 } //end of the iteration MAX_ITER
362
363
364 free(Xnew_sub);
365 } //parallel pragma
366
367 etime=omp_get_wtime();
368 dtime+=(etime-stime);
369 }
370 printf("parallel for 1D optimized blocked jacobi= \t%5f", (dtime/iter));
371
372 //printf("X_seq,N);
373 //printf("X,N);
374
375
376 // to test that the code working well
377 int result = 0;
378     for(i=0; i < N; i++){

```

```
379         if(fabs(X[i]-X_seq[i]) > ERR_THRESHOLD)
380     {
381         // printf("(%d) : (%.5f,%.5f)\n", i, X[i], X_seq[i]);
382         result = 1;
383     }
384 //if(result == 1) break;
385 }
386
387
388     printf("\tTest %s", (result == 0) ? "Passed\n" : "Failed\n");
389
390 free(A);
391 free(B);
392 free(X);
393 free(X_seq);
394 free(new_x);
395
396     return 0;
397 }
398
```



```

1 //RELAXED JACOBI SOLVER
2 #include <stdio.h>
3 #include <math.h>
4 #include <unistd.h>
5 #include <time.h>
6 #include <sys/time.h>
7 #include <math.h>
8 #include <stdlib.h>
9 #include <omp.h>
10 #define MAX_ITER 100
11 #define ERR_THRESHOLD 0.00001
12 #define RUN_CRITICAL
13 #define RUN_ATOMIC
14 #define MAX 4
15
16 int N = 8;
17 int T = 2;
18 int tr=0;
19
20 int rows_size=2;
21 int cols_size=2;
22 //This is the prototype for the queue function that will be used in the
23 implementation
24
25 //#####
26 #####
27 //##### Intialization Funtions that used in JACOBI TO INSURE
28 CORRECTNESS OF OUR WORK###
29 //#####
30 #####
31
32
33 void randomInit(double *A,double *X,double *B,int wA)
34 {
35     int i,j;
36     for(i = 0; i < wA; i++)
37     {
38         for(j = 0; j < wA; j++)
39         {
40             if (i==j)
41             {
42                 A[i*wA+j] = wA;
43             }
44             else
45             {
46                 A[i*wA+j] = -1 ;
47             }
48         }
49
50         X[i] = 0;
51         B[i] = 1;
52     }
53 }
54
55
56 void print1d(double *a,int N)
57 {
58     printf("\nStart printing\n");
59     int i,j;
60     for(i=0;i<N;i++)
61     {
62         printf("%.5f\n",a[i]);
63     }

```

```

64 printf("\nEnd printing\n");
65 }
66 void print2d(double *a,int N)
67 {
68     int i,j;
69     printf("\nStart printing\n");
70     for(i=0;i<N;i++)
71     {
72         for(j=0;j<N;j++)
73         {
74             printf("%.5f\t",a[i*N+j]);
75         }
76         printf("\n");
77     }
78     printf("\nEnd printing\n");
79 }
80
81
82
83 int main(int argc, char *argv[])
84 {
85
86     int iter=1;
87     if(argc > 1)
88         N = atoi(argv[1]);
89     if(argc > 2)
90         T = atoi(argv[2]);
91     if(argc > 3)
92         rows_size=atoi(argv[3]);
93     if(argc > 4)
94         cols_size=atoi(argv[4]);
95     if(argc > 5)
96         tr=atoi(argv[5]);
97
98
99     if ( argc < 1 )
100 printf("\n You muste Enter the following parameter : Mtrix size , number of
101 threads , rows size , column size. For Relaxed jacobi number of rows and blocks
102 will be the number of threads for simplicity we will extend them later to
103 improve the performance\n");
104
105
106     double sum;
107
108     int i,i1,j,k,ii,jj;
109     int kk=N/T; // to compute the size of the rows for each thread.
110     double *A=malloc(sizeof(double)*N*N);
111     double *B=(double*)malloc(N*sizeof(double));
112     double *X=(double*)malloc(2*N*sizeof(double)); // this is to store X at
113 the intialization ,at K and also K+1
114 //double *XX=(double*)malloc(N*sizeof(double)); // this is to store
115 result X that produced from K iteration which will be referenced by the mod2=0
116 //double *XXX=(double*)malloc(N*sizeof(double)); // this is to store
117 result X that produced from K+1 iteration which will be referenced by mod2==1
118 //double *Xp; // to store the value of X for each thread
119 double *Xnew_sub; // to store the new value that is computed by each
120 thread
121 double *new_x=(double*)malloc(N*sizeof(double));
122 double *X_seq=(double*)malloc(N*sizeof(double));
123 int work[T]; // this is a shared array that contains 0 , 1 for the
124 blocks that is processed
125
126

```

```

127
128 //we an assumption that the number of blocks will be the number of
129 threads in column dimention to insure simplicity for the programming
130
131 //accuracyTestInit2D(A,N);
132 //accuracyTestInit1D(B,N);
133 //accuracyTestInit1D(X,N);
134 //accuracyTestInit1D(X_seq,N);
135 /*for(i=0;i<N;i++)
136 {
137     for(j=0;j<N;j++)
138     {
139         A[i*N+j]=sqrt(i+j)*0.2546;
140     }
141 }
142 for(i=0;i<N;i++)
143 {
144     B[i]=sqrt(i*j);
145 }*/
146 randomInit(A,B,X,N);
147 for(i=0;i<N;i++)
148 {
149     X_seq[i]=X[i];
150 }
151     double dtime=0.0;
152     double etime=0.0, stime=0.0;
153
154 // here to compute the Sequential version
155 //printf("\nstart computing Sequential jacobi....\n");
156 /*printf("\nStart Computing Sequential\n");
157 stime = omp_get_wtime();
158 for(k = 0; k < MAX_ITER; k++){
159
160     for(i=0; i<N; i++){
161         sum = 0.0;
162         sum=sum-(A[i*N+i] * X_seq[i]);
163
164         for(j=0; j<N; j++){
165
166             sum =sum + (A[i*N+j] * X_seq[j]);
167         }
168
169         new_x[i] = (B[i] - sum)/A[i*N+i];
170
171     }
172
173     for(i=0; i < N; i++)
174         X_seq[i] = new_x[i];
175 }
176 etime = omp_get_wtime();
177
178     dtime = etime - stime;*/
179
180 //printf("\ncomputed sequential for 1 dimention=%.5f\n", dtime);
181
182
183
184 // Optimized Parallel version of jacobi using RELAXED SYNCHRONIZATION
185 // *****
186 // *****
187 omp_set_num_threads(T);
188 dtime=0.0;
189 int kkk;

```

```

190
191 for(kkk=0;kkk<iter;kkk++)
192 {
193 //accuracyTestInit1D(X,N);
194 stime = omp_get_wtime();
195
196 #pragma omp parallel shared (A,B,X,N,T,kk,rows_size,cols_size,work)
197 private(k,i,ii,j,sum,Xnew_sub)
198 {
199     //Xp=(double*)malloc(N*sizeof(double));
200     Xnew_sub=(double*)malloc((kk)*sizeof(double));//we multiply it by 2 to
201 store for K and for K+1
202     int p_work[T];// this is a private variable that all the time its value
203 copied from the shared variable "work"
204     int tid=omp_get_thread_num();
205     ii=tid*kk;// index of te rows that related to the the thread
206     int i_n=kk/rows_size;//number of rows chunk
207     int j_n=N/cols_size;//number of column chunk
208     int w[T];// the blocks that is processed for each iteration by the thread
209     int iii,jjj;
210     int index_r,index_w;
211     int temp;
212     int counter;
213     int spin=1;
214     iii=0;
215     int loop=1;
216     int av=0;
217
218
219
220
221
222 for(i=0;i<kk;i++)
223 Xnew_sub[i]=0;
224
225 for(k=0;k<MAX_ITER;k++)
226 {
227 // we must reset the shared work matrix by the thread0
228 if(tid==0)
229 {
230     //printf("\nThe work setted by thread 0\n");
231     for(i=0;i<T;i++)
232     {
233         work[i]==0;
234     }
235 }
236
237 //if(omp_get_thread_num()==tr)
238 //printf("\nIteration#=%d",k);
239 // To decrease overhead of index computation. We must reorganize the location
240 of index computation.
241 if(k%2==0)
242 {
243     index_r=0;
244     index_w=1;
245 }
246 else
247 {
248     index_r=1;
249     index_w=0;
250 }
251
252 for(i=0;i<T;i++)

```

```

253 {
254     w[i]=0;
255 }
256
257 counter=0;
258
259
260 if(k==0)
261 {
262     iii=0;
263     jjj=0;
264 }
265 else
266 {
267     //iii=tid;//becuase the first block they will compute the block that computed
268     by it
269     jjj=tid;//becuase the first block they will compute the block that computed by
270     it
271     w[tid]=1;
272 }
273
274 //for(iii=0;iii<i_n;iii++) // here to count the rows
275 //{
276     //for(jjj=0;jjj<j_n;jjj++) // here to count the blocks
277     //{
278     while( counter < (T-1)) // this is means that we process all the blocks for a
279     specific iteration
280     {
281     spin=1; // we must reset it
282     loop=1; // reset of the loop
283     //if(omp_get_thread_num()==tr)
284     //printf("\nCounter=%d\n",counter);
285     for(i=0; i<rows_size; i++){
286     sum=0.0;
287     //sum = sum-(A[(i+iii+iii)*N+i+ii+iii]*Xp[i+ii+iii]);
288     for(j=0; j<cols_size; j++){
289     sum+= A[(i+ii+(iii*rows_size))*N+j+(jjj*cols_size)] *
290     X[j+(jjj*cols_size)+index_r*N];
291     }
292     //Xnew_sub[i+iii]+= (B[i+ii+iii] -
293     sum)/A[(i+ii+iii)*N+i+ii+iii];
294     Xnew_sub[i+(iii*rows_size)]+=sum;
295     }
296     if(k!=0) // it is not the first iteration
297     {
298     while(spin==1 )
299     {
300     while(loop==1)
301     {
302     #pragma omp flush(work)
303     for(i=0;i<T;i++)
304     {
305     p_work[i]=work[i];
306     if(p_work[i]==1 && w[i] == 0)
307     {
308     loop=0; // to stop looping
309     jjj=i;
310     w[i]=1;
311     spin=0;
312     break;
313     }
314     }
315     //printf("\nSpinning Searching for a block\n");

```

```

316                                     } // this is to keep spinning untill a work a
317 available
318                                     //printf("\nSpinning Searching for a work\n");
319                                     } // this is to spin searching for a work
320                                     }
321                                     else
322                                     {
323                                     //iii++;
324                                     jjj++;
325                                     }
326 counter++;
327 }
328                                     //} //end of jjj
329                                     for(i=0;i<rows_size;i++)
330                                     {
331 Xnew_sub[i+(iii*rows_size)]=Xnew_sub[i+(iii*rows_size)]-
332 (A[(i+ii+(iii*rows_size))*N+i+ii+(iii*rows_size)]*X[i+ii+(iii*rows_size)+index_
333 r*N]);
334 Xnew_sub[i+(iii*rows_size)]=(B[i+ii+(iii*rows_size)]-
335 Xnew_sub[i+(iii*rows_size)])/A[(i+ii+(iii*rows_size))*N+i+ii+(iii*rows_size)];
336                                     }
337                                     }
338 //} //end of iii
339 //here is incorret add of the barrier , the barrier must be added before
340 reading the data to update Xp
341     for(i=0;i<kk;i++){
342         X[i+ii+(N*index_w)]=Xnew_sub[i]; // here to define where to store
343 the data for iter k , or k+1
344     }
345     work[tid]=1; // to indicate that a block is a available
346     /*if(tid==tr)
347     {
348     for(i=0;i<T;i++)
349     {
350     printf("\t%d",work[i]);
351     }
352     }*/
353
354
355 } //end of the iteration MAX_ITER
356
357 free(Xnew_sub);
358 } //parallel pragma
359
360 etime=omp_get_wtime();
361 dtime+=(etime-stime);
362 }
363 printf("\nparallel for 1D optimized Relaxed Jacobi= \t%5f\n", (dtime/iter));
364
365 //print1d(X_seq,N);
366 //print1d(X,N);
367
368
369 // to test that the code working well
370 /*int result = 0;
371 for(i=0; i < N; i++){
372     if(fabs(X[i]-X_seq[i]) > ERR_THRESHOLD)
373     {
374         // printf("(%d) : (%.5f,%.5f)\n", i, X[i], X_seq[i]);
375         result = 1;
376     }
377 }
378 //if(result == 1) break;

```

```
379     }
380
381     printf("\tTest %s", (result == 0) ? "Passed\n" : "Failed\n");*/
382
383
384     free(A);
385     free(B);
386     free(X);
387     free(X_seq);
388     free(new_x);
389
390     return 0;
391 }
392
```

## Appendix C: BARNES-HUT N-BODY SIMULATION

- **Nature of the application**

The N-body simulation is considered as a model of semi static computations. A brute force approach for computing the gravitational forces for N bodies is on the  $O(N^2)$ . The Barnes Hut (BH) approximation enables treating a group of bodies as one if these are far enough from a given body. This drops the computational complexity to  $O(N \log N)$  when using BH. BH uses an oct-tree, in which each node stores the aggregate mass of all of its children nodes (sub-tree) at their center of mass. Another problem is that the thread load moderately changes from one iteration to another due to body motion in space. Therefore, a static problem partitioning strategy (S-BH) for BH is likely to suffer from accumulated load unbalance. It well known that dynamic load balancing (DLB) improves BH scalability. However, DLB is complex because of the need to measure the Dynamic Load (DL) and adopt an adequate data structure to minimize runtime overheads. In the beginning of iteration k, the body slowly motion enables estimating the DL for K+1 as being the aggregate load measured by all the treads in iteration k. Thus DLB is implemented by evenly partitioning the DL over the threads so that to preserve the data locality to the best possible. I implemented DLB-BH using an efficient data structure to ease load redistribution together with oct-tree implementation.

- **Data Structure**

I have two main arrays in the implementation. The bodies array which contains the velocity, postions and the mass. Also, it contains an index for the DFT which is used to



order the array (refere to chapter 5 for more details). In addition, the nodes array is implemented which contains center of mass, pointer to the next subtree , index of DFT and variable to differe between node and body object.

- **Procedures**

There are many procedures in the code. To simplify understanding each procedure is commented to expalain its purpose and how it is work.

- **Input and Output**

Inside the code there is a procedure called "Generate\_GalaxyKing" . It takes a galaxy of Size  $10^6$  and generates a 125 galaxies. They are spreaded in the space of size (400x400). (refere to chapter 5, section 5.9 for more detailes).

```

1 //BARNES-HUT N-BODY SIMULATION
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <algorithm>
6 #include <math.h>
7 #include <time.h>
8 #include <omp.h>
9 // number of steps for the program
10
11 unsigned long report1[10][10];
12 double allam_report[5*4800][10]; // this is added to report the
13 data that is produced from the program
14 #define steps 20
15 unsigned long normal=0,seq=0,zone=0,dynamic=0;
16 unsigned long threads= 1;
17 #define MAX_THREADS 240
18 unsigned long ThreadStart[MAX_THREADS];
19 unsigned long ThreadEnd[MAX_THREADS];
20 unsigned long long ThreadCDF[MAX_THREADS];
21 double ThreadStartTime[MAX_THREADS];
22 double ThreadEndTime[MAX_THREADS];
23 double ThreadAvgTime[MAX_THREADS];
24 unsigned long generate=0;
25 #define PI (atan((float)1)*4)
26 #define maxB 13000000
27 #define maxNodes (maxB * 10)
28 unsigned long MaxNBodies = 0;
29 #define minimum_space 0.01
30 struct Body {
31     unsigned long id;
32     float posX, posY, posz;
33     float vx, vy, vz;
34     float fx, fy, fz;
35     void* my_node;
36     float mass;
37     unsigned long long zoneCost;
38     void operator= (const Body &);
39     inline bool operator< (const Body &);
40 };
41
42 struct node {
43     struct Body *B;
44     float cx, cy, cz; //Center of the square
45     float d; //Half side of the square
46     unsigned long is_internal_node; //1 = internal node,
47 0 = leaf node
48     struct node* child[8]; //Pointers to the node children
49     //struct node* next;
50     struct node* parent;
51     unsigned long BH;
52     unsigned long index_in_brothers;

```

```

53     unsigned long index_of_next_neighbour;
54     unsigned long BreadthFirstIndex;
55 };
56
57 static Body * Bodies[maxB] = {NULL}; // here it is an array of
58 bodies
59 static node * AllNodes2[maxNodes] = {NULL}; // here it is an
60 array of nodes
61
62 // softening parameter
63 #define EPS 30000
64 #define EPS2 (EPS * EPS)
65
66 int id_b=0;
67
68 //time for each step
69 const float dt = (float) 0.01;
70
71 // gravational constant
72 const float G = (float)6.67* (float)pow(10.0,-11.0);
73
74 const char * filename_simple = "data/Bodies001_simple.txt";
75 const char * filename_simple_sorted =
76 "data/Bodies001_simple_sorted.txt";
77
78 const char * filename_poisson = "data/Bodies001_poisson.txt";
79 const char * filename_poisson_sorted =
80 "data/Bodies001_poisson_sorted.txt";
81
82 const char * filename_poisson2 = "data/Bodies001_poisson2.txt";
83 const char * filename_poisson_sorted2 =
84 "data/Bodies001_poisson_sorted2.txt";
85 const char * filename_test ="data/Bodies001_poisson2.txt";
86 //"data/100t.txt";
87 const char * filename_GalaxyKingModel_100 =
88 "data/KingModel_0.1MBodies.txt";
89 const char * filename_GalaxyKingModel_1M=
90 "data/KingModel_1MBodies.txt";
91 const char * filename_GalaxyKingModel_1M_nonsorted=
92 "data/KingModel_1MBodies_nonsorted.txt";
93
94 char * filepath;// = "D:/NBdata/";
95
96 unsigned long No_of_Nodes = 0;
97
98 #define maxVelBound 10 //The maximum initial
99 velocity of a body
100 #define maxMassBound 9 //The maximum mass for a
101 given body
102 #define maxPosBound 400 //The maximum value for x
103 and y coordinates of a body

```

```

104 #define minPosBound 0 //The maximum value for x and y
105 coordinates of a body
106 #define spaceCenterX (maxPosBound / 2) //The position
107 of the x coordinate of Center of space
108 #define spaceCenterY (maxPosBound / 2) //The position
109 of the y coordinate of Center of space
110 #define spaceCenterZ (maxPosBound / 2) // <anas added for 3d :
111 The position of z coordinate of Center of space />
112 #define spaceCenterHfside (maxPosBound / 2) //Half side of Space
113
114 void Body::operator= (const Body &B2)
115 {
116     id = B2.id;
117     posx = B2.posx;
118     posy = B2.posy;
119     posz = B2.posz;
120     mass = B2.mass;
121     vx = B2.vx;
122     vy = B2.vy;
123     vz = B2.vz;
124 }
125
126 bool Body::operator< (const Body &B2) {
127     float my_dist = sqrt( (posx-spaceCenterX) * (posx-
128 spaceCenterX) +
129 (posy-spaceCenterY) * (posy-spaceCenterY) +
130 (posz-spaceCenterZ) * (posz-spaceCenterZ));
131     float his_dist = sqrt( (B2.posx-spaceCenterX) * (B2.posx-
132 spaceCenterX) +
133 (B2.posy-spaceCenterY) * (B2.posy-spaceCenterY) +
134 (B2.posz-spaceCenterZ) * (B2.posz-spaceCenterZ));
135     return my_dist < his_dist;
136 }
137 }
138
139 inline bool Body_comparer_function(const Body* B1, const Body*
140 B2) {
141     return sqrt( (B1->posx-spaceCenterX) * (B1->posx-
142 spaceCenterX) +
143 (B1->posy-spaceCenterY) * (B1->posy-spaceCenterY) +
144 (B1->posz-spaceCenterZ) * (B1->posz-spaceCenterZ) )
145 <
146 sqrt( (B2->posx-spaceCenterX) * (B2->posx-
147 spaceCenterX) +
148 (B2->posy-spaceCenterY) * (B2->posy-spaceCenterY) +
149 (B2->posz-spaceCenterZ) * (B2->posz-spaceCenterZ) );
150 }
151 }
152
153 struct node* newNode(float cx, float cy, float cz, float d, Body
154 *B1) { //
155     node* nd = new node();

```

```

156     //nd = new(struct node);
157     if (No_of_Nodes >= maxNodes)
158     {
159         printf("Error: increase the no of nodes\nPress any key
160 to exit");
161         getchar();
162         exit(-1);
163     }
164     if (AllNodes2[No_of_Nodes] != NULL)
165         delete(AllNodes2[No_of_Nodes]);
166     AllNodes2[No_of_Nodes++] = nd;
167     nd->B = B1;
168     if(B1 != NULL)
169         B1->my_node = nd;
170     nd->cx = cx;
171     nd->cy = cy;
172     nd->cz = cz;
173     nd->d = d;
174     nd ->is_internal_node = 0;
175     nd ->BH = 0;
176     //nd ->next = NULL;
177     nd ->parent= NULL;
178     nd ->index_in_brothers = 8; //8 means no index // a correct
179 index should be between 0 and 7
180     // nd ->H_index= -1;
181     nd->BreadthFirstIndex = No_of_Nodes - 1;
182     for(unsigned long i = 0; i < 8; i++)
183         nd->child[i] = NULL;
184     return(nd);
185 }
186
187 bool saveBodiesLocations(const char * filename1)
188 {
189     char filename [255] = "";
190     strcpy(filename,filepath);
191     strcat(filename,filename1);
192
193     char bodydata[60];
194     FILE *f;
195     if ((f=fopen(filename , "w")) == NULL)
196     {
197         printf("The file '%s' was not opened\n", filename);
198         return true;
199     }
200     else
201     {
202         for (unsigned long ii = 0; ii < MaxNBodies; ii++)
203         {
204             sprintf( bodydata,"%12f %12f %12f\n", Bodies[ii]-
205 >posx, Bodies[ii]->posy, Bodies[ii]->posz);
206             fputs (bodydata , f);
207         }

```

```

208     }
209     fclose(f);
210     return false;
211 }
212
213
214 inline char * Body_toString(Body * b, char * bodydata)
215 {
216     sprintf(bodydata,"%6.12f, %6.12f, %6.12f, %6.12f, %6.12f,
217 %6.12f, %6.12f, %6.12f, %6.12f, %6.12f\n\0",
218         b->mass,
219         b->posx ,b->posy, b->posz,
220         b->vx ,b->vy, b->vz,
221         b->fx ,b->fy, b->fz);
222
223     return bodydata;
224 }
225
226 bool saveBodies(const char * filename1)
227 {
228     char filename [255]= "";
229     strcpy(filename,filepath);
230     strcat(filename,filename1);
231
232     char bodydata[(10*21+5)];
233     FILE *f;
234     if ((f=fopen(filename , "w")) == NULL)
235     {
236         printf("The file '%s' was not opened\n", filename);
237         return true;
238     }
239     else
240     {
241         for (unsigned long ii = 0; ii < MaxNBodies; ii++)
242         {
243             Body_toString(Bodies[ii], bodydata);
244             fputs (bodydata , f);
245         }
246     }
247     fclose(f);
248     return false;
249 }
250
251
252 inline void parseString(Body *b, char * bodydata)
253 {
254     char s [30];
255     unsigned long jj=0, ii=0;
256
257     while( (bodydata[ii]!='\t') && (bodydata[ii]!='\n')&&
258 (bodydata[ii]!=' ')&& (bodydata[ii]!='\0')) s[jj++] =
259 bodydata[ii++];

```

```

260     s[jj]='\0';
261     b->mass = (float) atof(s);
262     ii++;
263     jj = 0;
264
265     while( (bodydata[ii]!='\t') && (bodydata[ii]!='\n')&&
266 (bodydata[ii]!=' ')&& (bodydata[ii]!='\0')) s[jj++] =
267 bodydata[ii++];
268     s[jj]='\0';
269     b->posx = (float) atof(s);
270     ii++;
271     jj = 0;
272
273     while( (bodydata[ii]!='\t') && (bodydata[ii]!='\n')&&
274 (bodydata[ii]!=' ')&& (bodydata[ii]!='\0')) s[jj++] =
275 bodydata[ii++];
276     s[jj]='\0';
277     b->posy = (float) atof(s);
278     ii++;
279     jj = 0;
280
281     while( (bodydata[ii]!='\t') && (bodydata[ii]!='\n')&&
282 (bodydata[ii]!=' ')&& (bodydata[ii]!='\0')) s[jj++] =
283 bodydata[ii++];
284     s[jj]='\0';
285     b->posz = (float) atof(s);
286     ii++;
287     jj = 0;
288
289     while( (bodydata[ii]!='\t') && (bodydata[ii]!='\n')&&
290 (bodydata[ii]!=' ')&& (bodydata[ii]!='\0')) s[jj++] =
291 bodydata[ii++];
292     s[jj]='\0';
293     b->vx = (float) atof(s);
294     ii++;
295     jj = 0;
296
297     while( (bodydata[ii]!='\t') && (bodydata[ii]!='\n')&&
298 (bodydata[ii]!=' ')&& (bodydata[ii]!='\0')) s[jj++] =
299 bodydata[ii++];
300     s[jj]='\0';
301     b->vy = (float) atof(s);
302     ii++;
303     jj = 0;
304
305     while( (bodydata[ii]!='\t') && (bodydata[ii]!='\n')&&
306 (bodydata[ii]!=' ')&& (bodydata[ii]!='\0')) s[jj++] =
307 bodydata[ii++];
308     s[jj]='\0';
309     b->vz = (float) atof(s);
310     ii++;
311     jj = 0;

```

```

312 }
313
314 void generate_positions_poisson2() {
315     srand((unsigned long) (time(NULL)*100));
316     unsigned long i;
317     float a,b,r0_y,r_y,r_x,u,r0_x,r_z,r0_z;
318     r0_y = maxPosBound / 15;
319     r0_x = maxPosBound / 15;
320     r0_z = maxPosBound / 15;
321     for (i = 0; i < MaxNBodies; i++) {
322         Bodies[i] = new Body;
323         Bodies[i]->id = i;
324         do {
325             u = (float) rand() / RAND_MAX;
326             r_y = -r0_y*log(1-u);
327             r_x = -r0_x*log(1-u);
328             r_z = -r0_z*log(1-u);
329             a = ((float) (rand()) / RAND_MAX) * 2 * PI;
330             b = ((float) (rand()) / RAND_MAX) * 2 * PI;
331             Bodies[i]->posx = (r_x*cos(a)*cos(b)) +
332 maxPosBound / 2;
333             Bodies[i]->posy = (r_y*cos(a)*sin(b)) +
334 maxPosBound / 2 ;
335             Bodies[i]->posz = (r_z*sin(a)) + maxPosBound / 2;
336         } while (Bodies[i]->posx > maxPosBound || Bodies[i]-
337 >posy > maxPosBound || Bodies[i]->posz > maxPosBound ||
338 Bodies[i]->posx < minPosBound || Bodies[i]->posy
339 < minPosBound || Bodies[i]->posz < minPosBound
340 );
341             Bodies[i]-> mass = 10;//(float) rand() / RAND_MAX *
342 maxMassBound + 1;
343             Bodies[i]->vx = ((float) rand() / RAND_MAX *
344 maxVelBound) - (float)maxVelBound/2;
345             Bodies[i]->vy = ((float) rand() / RAND_MAX *
346 maxVelBound) - (float)maxVelBound/2;
347             Bodies[i]->vz = ((float) rand() / RAND_MAX *
348 maxVelBound) - (float)maxVelBound/2;
349             Bodies[i]->fx = (float) rand() / RAND_MAX *
350 maxVelBound;
351             Bodies[i]->fy = (float) rand() / RAND_MAX *
352 maxVelBound;
353             Bodies[i]->fz = (float) rand() / RAND_MAX *
354 maxVelBound;
355             Bodies[i]->fx = Bodies[i]->fy = Bodies[i]->fz = 0;
356         }
357     }
358
359 bool loadBodies(const char * filename1)
360 {
361     char filename [255] = "";
362     strcpy(filename,filepath);
363     strcat(filename,filename1);

```



```

364
365     char bodydata[(10*21+5)];
366     FILE *f;
367     if ((f=fopen(filename , "r")) == NULL)
368     {
369         printf("The file '%s' was not opened\n", filename);
370         return true;
371     }
372     else
373     {
374         for (unsigned long ii = 0; ii < MaxNBodies; ii++)
375         {
376             fgets (bodydata, (10*21+5), f);
377             Bodies[ii] = new Body();
378             //Bodies[ii]->zoneCost = 0;
379             parseString(Bodies[ii], bodydata);
380         }
381     }
382     fclose(f);
383     return false;
384 }
385
386
387 void CountTree(struct node* root, unsigned long* NoOfNodes,
388 unsigned long* NoOfBodies)
389 {
390     if (root==NULL)
391         return;
392
393     if (root->is_internal_node)
394         (*NoOfNodes)++;
395     else
396         (*NoOfBodies)++;
397
398     for(unsigned long i = 0; i < 8; i++)
399         if(root->child[i] != NULL)
400             CountTree(root->child[i], NoOfNodes, NoOfBodies);
401     else
402         return;
403 }
404 void CountTree2(unsigned long* NoOfNodes, unsigned long*
405 NoOfBodies)
406 {
407     for(unsigned long i = 0; i < No_of_Nodes; i++)
408         if (AllNodes2[i]->is_internal_node)
409             (*NoOfNodes)++;
410         else
411             (*NoOfBodies)++;
412 }
413
414 void PrintTree_DFS2()
415 {

```

```

416     printf("  i BF_i chld next zoneCost mass\n");
417     for (unsigned long ii=0; ii<No_of_Nodes; ii++)
418         printf("%3d %3d %3d %3d %3llu %f\n", ii,
419 AllNodes2[ii]->BreadthFirstIndex, AllNodes2[ii]-
420 >index_in_brothers, AllNodes2[ii]->index_of_next_neighbour,
421 AllNodes2[ii]->B->zoneCost, AllNodes2[ii]->B->mass);
422 }
423 // this function is added to compute the tree depth
424 /*void printoctTree(struct node * root)
425 {
426 if ( root==NULL);
427 printf("The Tree is empty");
428 else
429 {
430 while
431
432
433 }
434 }*/
435
436 bool compare_nodes(node* n1, node* n2)
437 {
438     return ( (n1->BreadthFirstIndex) < (n2-
439 >BreadthFirstIndex));
440 }
441
442 bool compare_bodies(Body* b1, Body* b2)
443 {
444     return ( (((node *) (b1->my_node))->BreadthFirstIndex) <
445 (((node *) (b2->my_node))->BreadthFirstIndex) );
446 }
447 void compute_Center_of_Mass(node * Node)
448 {
449     float CMX = 0.0, CMY = 0.0, CMZ = 0.0;
450     Node->B = new Body;
451     Node->B->my_node = Node;
452     Node->B->zoneCost = 0;
453     Node->B->mass = 0;
454     for(unsigned long i = 0; i < 8; i++)
455         if(Node->child[i] != NULL)
456             {
457                 Node->B->mass += Node->child[i]->B->mass;
458                 CMX += Node->child[i]->B->mass * Node->child[i]-
459 >B->posx;
460                 CMY += Node->child[i]->B->mass * Node->child[i]-
461 >B->posy;
462                 CMZ += Node->child[i]->B->mass * Node->child[i]-
463 >B->posz;
464                 Node->B->zoneCost += Node->child[i]->B->zoneCost;
465 // here we must check this operation
466             }
467 }

```

```

468         Node->B->posx = CMX / Node->B->mass;
469         Node->B->posy = CMY / Node->B->mass;
470         Node->B->posz = CMZ / Node->B->mass;
471     }
472
473     unsigned long get_next_brother_index(node * Node, unsigned long
474     start)
475     {
476         unsigned long i =start;
477         while ( (i < 8) && (Node->child[i] == NULL) ) i++;
478         return i;
479     }
480
481     void compute_indices_BreadthFirst(node * Node)
482     {
483         unsigned long k = 0, i, j = 0;
484         node* nd = Node;
485         nd->BreadthFirstIndex = k++;
486         while(nd != NULL)
487         {
488             i = get_next_brother_index(nd, j);
489             if (i > 7 )
490             {
491                 if ( (nd->is_internal_node == 1) &&
492 (get_next_brother_index(nd, j) > 7) )
493                     compute_Center_of_Mass(nd);
494                 j = nd->index_in_brothers + 1;
495                 nd = nd->parent;
496                 if(nd != NULL)
497                     nd->index_of_next_neighbour = k;
498             }
499             else
500             {
501                 j = 0;
502                 nd = nd->child[i];
503                 nd->BreadthFirstIndex = k++;
504                 nd->index_of_next_neighbour = k;
505             }
506         }
507         if(Node != NULL)
508             compute_Center_of_Mass(Node);
509     }
510
511     void delete_node_tree()
512     {
513         for(unsigned long i = 0; i < No_of_Nodes; i++)
514         {
515             if (AllNodes2[i] != NULL)
516             {
517                 if (AllNodes2[i]->is_internal_node == 1)
518                 {
519                     if (AllNodes2[i]->B != NULL)

```

```

520             {
521                 delete (AllNodes2[i]->B);
522                 AllNodes2[i]->B = NULL;
523             }
524         }
525         delete (AllNodes2[i]);
526         AllNodes2[i] = NULL;
527     }
528 }
529 No_of_Nodes = 0;
530 }
531 void delete_AllBodies()
532 {
533     for(unsigned long i = 0; i < MaxNBodies; i++)
534     {
535         if (Bodies[i] != NULL)
536         {
537             delete (Bodies[i]);
538             Bodies[i] = NULL;
539         }
540     }
541     MaxNBodies = 0;
542 }
543 void delete_far_bodies ()
544 {
545     for(unsigned long i = 0; i < MaxNBodies; i++)
546         if (Bodies[MaxNBodies-1] != NULL)
547         {
548             if ( (Bodies[i]->posx < minPosBound) ||
549 (Bodies[i]->posx > maxPosBound)
550                 || (Bodies[i]->posy < minPosBound) ||
551 (Bodies[i]->posy > maxPosBound)
552                 || (Bodies[i]->posz < minPosBound) ||
553 (Bodies[i]->posz > maxPosBound) )
554             {
555                 Bodies[i] = Bodies[MaxNBodies-1];
556                 delete (Bodies[MaxNBodies-1]);
557                 Bodies[MaxNBodies-1] = NULL;
558                 MaxNBodies--;
559                 printf("\nA far body have been delete\n");
560             }
561         }
562 }
563
564 void update_velocity_and_position(struct Body *B)
565 {
566     //update velocity
567     if ((B->mass) != 0)
568     {
569         B->vx += dt * (B->fx) / (B->mass);
570         B->vy += dt * (B->fy) / (B->mass);
571         B->vz += dt * (B->fz) / (B->mass);

```

```

572     }
573     //update position
574     B->posx += dt * B->vx;
575     B->posy += dt * B->vy;
576     B->posz += dt * B->vz;
577 }
578
579 float distance(node* nd, Body* bd)
580 {
581     return sqrt((nd->cx - bd->posx)*(nd->cx - bd->posx)
582               + (nd->cy - bd->posy)*(nd->cy - bd->posy)
583               + (nd->cz - bd->posz)*(nd->cz - bd->posz));
584 }
585
586 bool InItsCube(node* nd, Body* bd)
587 {
588     return ( ((nd->cx - nd->d) <= bd->posx) && (bd->posx <=
589 (nd->cx + nd->d))
590           && ((nd->cy - nd->d) <= bd->posy) && (bd->posy <= (nd-
591 >cy + nd->d))
592           && ((nd->cz - nd->d) <= bd->posz) && (bd->posz <= (nd-
593 >cz + nd->d)) );
594 }
595
596 void add_Leaf(struct node* Node, struct Body *B1, unsigned long
597 i)
598 {
599     float newd = Node->d / 2;
600     float new_cx = (B1->posx > Node->cx) ? newd : -newd;
601     float new_cy = (B1->posy > Node->cy) ? newd : -newd;
602     float new_cz = (B1->posz > Node->cz) ? newd : -newd;
603     Node->child[i] = newNode(Node->cx + new_cx, Node->cy +
604 new_cy, Node->cz + new_cz, newd, B1);
605     Node->child[i]->is_internal_node = 0;
606     Node->is_internal_node = 1;
607     Node->B = NULL;
608     Node->child[i]->parent = Node;
609     Node->child[i]->index_in_brothers = i;
610 }
611 void add_Internal_Node(struct node* Node, struct Body *B1,
612 unsigned long i)
613 {
614     float newd = Node->d / 2;
615     float new_cx = (B1->posx > Node->cx) ? newd : -newd;
616     float new_cy = (B1->posy > Node->cy) ? newd : -newd;
617     float new_cz = (B1->posz > Node->cz) ? newd : -newd;
618     Node->child[i] = newNode(Node->cx + new_cx, Node->cy +
619 new_cy, Node->cz + new_cz, newd, NULL);
620     Node->child[i]->is_internal_node = 1;
621     Node->is_internal_node = 1;
622     Node->child[i]->parent = Node;
623     Node->child[i]->index_in_brothers = i;

```

```

624 }
625
626 void insert_body_in_octtree(struct node* Node, struct Body *B1)
627 //
628 {
629     Body* B2 = NULL;
630     node* nd = Node;
631     long j = -1;
632     unsigned long i = (B1->posz > Node->cz) * 1 + (B1->posy >
633 Node->cy) * 2 + (B1->posx > Node->cx) * 4;
634     if (nd->child[i] == NULL)
635         add_Leaf(nd, B1, i);
636     else
637     {
638         while ( (nd->child[i] != NULL) )
639         {
640             if (nd->child[i]->is_internal_node == 0)
641                 break;
642             nd = nd->child[i];
643             i = (B1->posz > nd->cz) * 1 + (B1->posy > nd->cy)
644 * 2 + (B1->posx > nd->cx) * 4;
645         }
646         if (nd->child[i] == NULL)
647             add_Leaf(nd, B1, i);
648         else
649         {
650             nd = nd->child[i];
651             i = (B1->posz > nd->cz) * 1 + (B1->posy > nd->cy)
652 * 2 + (B1->posx > nd->cx) * 4;
653             B2 = nd->B;
654             j = (B2->posz > nd->cz) * 1 + (B2->posy > nd->cy)
655 * 2 + (B2->posx > nd->cx) * 4;
656             while ( (i == j) && (nd->d > minimum_space) )
657             {
658                 add_Internal_Node(nd, B1, i);
659                 nd = nd->child[i];
660                 i = (B1->posz > nd->cz) * 1 + (B1->posy >
661 nd->cy) * 2 + (B1->posx > nd->cx) * 4;
662                 j = (B2->posz > nd->cz) * 1 + (B2->posy >
663 nd->cy) * 2 + (B2->posx > nd->cx) * 4;
664             }
665             add_Leaf(nd, B1, i);
666             if (i != j)
667                 add_Leaf(nd, B2, j);
668         }
669     }
670 }
671
672 node* createOcttree(struct node * root)
673 {
674     if (MaxNBodies<1)
675         return NULL;

```

```

676     root = newNode(spaceCenterX, spaceCenterX, spaceCenterZ,
677 spaceCenterHfside, NULL);
678     root->is_internal_node = 1;
679     unsigned long k;
680     float d;
681     for (unsigned long i = 0; i < MaxNBodies; i++)
682         insert_body_in_octtree(AllNodes2[0], Bodies[i]);
683     return root;
684 }
685
686
687 //Compute force on a given body
688 unsigned long long compute_body_force(Body *B)
689 {
690
691     node* r;
692     int bh_applied=0;
693     int int_applied=0;
694     unsigned long ii = 0;
695     unsigned long count = 0;
696     while(ii < No_of_Nodes)
697     {
698         //here a modification to the code is done my changing the
699         pacle of the counter to be inside the two if statment
700         //this update done by ALLAM on 16-march-2014
701         // count++; this counter moved to inside the if statments
702         r=AllNodes2[ii];
703         float dist2 = ((r->B->posx - B->posx) * (r->B->posx -
704 B->posx)
705 + (r->B->posy - B->posy) * (r->B->posy - B->posy)
706 + (r->B->posz - B->posz) * (r->B->posz - B-
707 >posz));
708         float dist = sqrtf(dist2);
709         //here I make a change to the code to compute how many times BH
710         has been applied
711         if ( (dist >= (r->d *2)) )
712         {
713             float den = dist2 + EPS2;
714             float F = (r->B->mass)/sqrtf(den*den*den);
715             B->fx += F * (r->B->posx - B->posx);
716             B->fy += F * (r->B->posy - B->posy);
717             B->fz += F * (r->B->posz - B->posz);
718             ii = r->index_of_next_neighbour;
719             bh_applied++; // this counter is added by ALLAM
720             to the code to count the times that BH is applied.
721             count++;
722         }
723         else if ( (r->is_internal_node==0))
724         {
725
726             float den = dist2 + EPS2;
727             float F = (r->B->mass)/sqrtf(den*den*den);

```

```

728         B->fx += F * (r->B->posx - B->posx);
729         B->fy += F * (r->B->posy - B->posy);
730         B->fz += F * (r->B->posz - B->posz);
731         ii = r->index_of_next_neighbour;
732         int_applied++;
733         count++;
734     }
735     else
736         ii++;
737 }
738 B->fx *= G * B->mass;
739 B->fy *= G * B->mass;
740 B->fz *= G * B->mass;
741
742 //printf("\nHow many times BH is applied=%d\n",bh_applied);
743 //printf("How many times on aleaf applied=%d\n",int_applied);
744 //printf("W:How many times force computed on a given
745 body=%d\n",count);
746
747     return count;
748 }
749
750 void Directory(char* a)
751 {
752     unsigned long ii = strlen(a) - 1;
753     while (ii>0 && a[ii]!='\\' && a[ii]!='/')
754         ii--;
755     a[ii+1] = '\\0';
756 }
757 //define MAX_THREADS 128
758 //unsigned long ThreadStart[MAX_THREADS];
759 //unsigned long ThreadEnd[MAX_THREADS];
760 void zone_cost_partitioning(unsigned long no_of_threads)
761 {
762     unsigned long long sum = 0;
763     unsigned long long W = AllNodes2[0]->B->zoneCost;
764     printf("\n The value of W work of the root node=%lu\n",W);
765     if (W == 0)
766     {
767         sum = MaxNBodies / no_of_threads;
768         for(unsigned long k = 0; k < no_of_threads; k++)
769         {
770             ThreadStart[k] = sum * k;
771             ThreadEnd[k] = sum * (k + 1) - 1;
772             ThreadCDF[k] = 0; // This is added by ALLAM to
773 insure that the CDF set to zero at the iteration 0
774         }
775     }
776     else
777     {
778         unsigned long i = 0;
779         unsigned long long sum = 0;

```



```

780         unsigned long long w = W / no_of_threads;
781         for(unsigned long k = 0; k < no_of_threads; k++)
782         {
783             W = (W - sum);
784             w = W / (no_of_threads - k);
785             ThreadStart[k] = i;
786             sum = 0;
787             while (sum < w)
788             {
789                 sum += Bodies[i]->zoneCost;
790                 i++;
791             }
792             ThreadEnd[k] = i - 1;
793             ThreadCDF[k] = sum;
794         }
795     }
796     ThreadEnd[no_of_threads - 1] = MaxNBodies - 1;
797     ThreadCDF[no_of_threads - 1] = (W - sum);
798 }
799 void Generate_GalaxyKing()
800 {
801     int ii = 0;
802     MaxNBodies = 100000;
803     loadBodies(filename_GalaxyKingModel_100);
804     for (int b = 0; b < 100000; b++)
805         for(int x = 0; x < 5; x++)
806             for(int y = 0; y < 5; y++)
807                 for(int z = 0; z < 5; z++)
808                 {
809                     ii = 100000 + z + y*5 + x*5*5 +
810 b*5*5*5;
811                     Bodies[ii] = new Body();
812                     Bodies[ii]->posx = Bodies[b]->posx + x
813 * 60;
814                     Bodies[ii]->posy = Bodies[b]->posy + y
815 * 60;
816                     Bodies[ii]->posz = Bodies[b]->posz + z
817 * 60;
818
819                     Bodies[ii]->mass = Bodies[b]->mass;
820                     Bodies[ii]->vx = Bodies[b]->vx;
821                     Bodies[ii]->vy = Bodies[b]->vy;
822                     Bodies[ii]->vz = Bodies[b]->vz;
823                 }
824                 MaxNBodies = ii + 1;
825
826     saveBodies(filename_GalaxyKingModel_1M_nonsorted);
827
828     std::sort(Bodies, &Bodies[MaxNBodies], Body_comparer_function
829 );
830     saveBodies(filename_GalaxyKingModel_1M);
831 }

```

```

832
833 //allam add a function to print the bodies array to see whats
834 going on in the system
835 void printbodies(Body *x[], unsigned long MaxNBodies)
836 {
837     int i;
838     for(i=0; i<MaxNBodies; i++)
839     {
840         printf("\n%d %.15f %.15f %.15f %.15f %.15f %.15f\n", i, x[i]-
841 >posx, x[i]->posy, x[i]->posz, x[i]->vx, x[i]->vy, x[i]->vz);
842     }
843 }
844 //Barnes Hut Simulation algorithm
845 void BarnesHutSimulation(const char * fileName)
846 {
847     double end22=0;
848     double end33=0;
849     double endd=0;
850     double end44=0;
851     double end55=0;
852     double timer_all1=0;
853     double timer_all2=0;
854     double timer_all=0;
855
856     unsigned long problemsize[8] =
857 {100000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 1};
858     //unsigned long problemsize[10] = {3000000, 4000000, 5000000,
859 8000000, 12000000, 3000000, 4000000, 8000000}; this is modified by
860 ALLAM
861     //unsigned long problemsize[10] = {1000};
862     //fl=1;
863     if(generate==0)
864     {
865         //MaxNBodies = problemsize[4];
866         Generate_GalaxyKing();
867         /*generate_positions_poisson2();
868         saveBodies(filename_poisson2); */
869
870         //std::sort(Bodies, &Bodies[MaxNBodies], Body_comparer_functi
871 on);
872         //saveBodies(filename_poisson_sorted2);
873
874         printf("file generated\n");
875     }
876     for (unsigned long ps = 0; ps < 1; ps++)
877     {
878         omp_set_num_threads(threads);
879         unsigned long numThreads = omp_get_num_threads();
880         //unsigned long numThreads=1;
881         MaxNBodies = problemsize[ps];
882
883         if (loadBodies(fileName))

```

```

884         return;
885     struct node * root;
886     /*char outfile[255]= "";
887     sprintf(outfile,"data/frames3/frame%6d.txt",0);
888     saveBodiesLocations(outfile);*/
889     // printf("maxB=%lu, maxNodes = %lu,
890 expected_nodes=%lu\n", maxB, maxNodes, unsigned long(maxB*(
891 log(maxB)/log(8) ))+1);
892     //sprintf(outfile,"<Create
893 tree>,<Sort>,<Force>,<update>,<delete>\n");
894     for(int n = 0; n < steps; n++)
895     {
896     printf("\niteration#=%d\n",n);
897     timer_all1 = omp_get_wtime();
898     // printf("iteration: %lu .. ", n);
899     //01-Create Tree
900     double start2 = omp_get_wtime();
901     root = createOoctree(NULL);
902     //PrintTree_DFS2();
903     //unsigned long* NoOfNode=0;
904     //unsigned long* NoOfbodies=0;
905     //CountTree(root,NoOfNode,NoOfbodies);
906     //printf("\nNumber of Nodes in the OCT
907 tree=%lu\n",No_of_Nodes);
908     //printf("\nNumber of bodies in the oct
909 Tree=%f\n",NoOfbodies);
910
911     double end2 = omp_get_wtime();
912     // printf("%f\t", (end2-start2));
913     /*if(ps==3)
914     printf("OctTree is created. Node count = %lu\n",
915 No_of_Nodes);*/
916
917     //02-Compute Center mass and Total mass
918     double start3 = omp_get_wtime();
919     compute_indices_BreadthFirst(root);
920     //printf("\nNumber of Nodes in the OCT
921 tree=%lu\n",No_of_Nodes);
922     std::sort(&AllNodes2[0],&AllNodes2[No_of_Nodes],
923 compare_nodes);
924     std::sort(&Bodies[0],&Bodies[MaxNBodies],
925 compare_bodies);
926     //this is added to print for each node the
927 indices of it
928     //printf("\n");
929     int i;
930     /*for(i=0;i<No_of_Nodes;i++)
931     {
932 //printf("index_in_brothers=%lu\tindex_of_next_neighbour=%lu\tBre
933 adthFirstIndex=%lu\n",AllNodes2[i]-
934 >index_in_brothers,AllNodes2[i]-
935 >index_of_next_neighbour,AllNodes2[i]->BreadthFirstIndex);

```

```

936 printf("BreadthFirstIndex=%lu\tindex_in_brothers=%lu\tindex_of_ne
937 xt_neighbour=%lu\n",AllNodes2[i]->BreadthFirstIndex,
938 AllNodes2[i]->index_in_brothers, AllNodes2[i]-
939 >index_of_next_neighbour);
940     */
941     double end3 = omp_get_wtime();
942     //printf("Time taken sort : %f\n", (end3-
943 start3));
944
945
946     //unsigned long no_of_threads =
947 omp_get_num_threads();
948     //zone_cost_partitioning(threads);
949
950     Body *B;
951     //03-Compute Forces
952     double start =0;
953     double end =0;
954     if(zone==1)
955     {
956
957         printf("\n#####
958 #####\n");
959         printf("The Result for the COST ZONE");
960
961         printf("\n#####
962 #####\n");
963         zone_cost_partitioning(threads);
964         end3 = omp_get_wtime();
965         unsigned long i=0;
966         unsigned long threadID=0;
967         start = omp_get_wtime();
968
969 #pragma omp parallel private(B,i,threadID)
970     {
971         numThreads = omp_get_num_threads();
972
973         threadID = omp_get_thread_num();
974         ThreadStartTime[threadID] =
975 omp_get_wtime();
976         //printf("threadstart= %lu
977 %f\n",threadID,ThreadStartTime[threadID]);
978         for(i = ThreadStart[threadID]; i <=
979 ThreadEnd[threadID]; i++)
980     {
981         B = Bodies[i];
982         B->fx = B->fy = B->fz = 0.0;
983         Bodies[i]->zoneCost =
984 compute_body_force(B);
985     }
986         ThreadEndTime[threadID] =
987 omp_get_wtime();

```

```

988     printf("thread#= %d thread time= %f work=%lu\n"
989     ,threadID, (ThreadEndTime[threadID]-
990     ThreadStartTime[threadID]),ThreadCDF[threadID]); // print EXEC
991     time per thread
992     }
993     for (unsigned long i = 0; i < numThreads;
994     i++)
995     {
996         ThreadAvgTime[i] += (ThreadEndTime[i]-
997     ThreadStartTime[i]);
998         //printf("thread time=    %f, %f,
999     %f\n",ThreadAvgTime[i],ThreadStartTime[i],ThreadEndTime[i]);
1000    }
1001    end = omp_get_wtime();
1002    //printbodies(Bodies,MaxNBodies); // here I add it to print
1003    the location of the bodies after the iterations of the cost zone
1004    parallel
1005    }
1006    if(seq==1)
1007    {
1008
1009        printf("\n#####
1010    #####\n");
1011        printf("The Result for the sequential");
1012
1013        printf("\n#####
1014    #####\n");
1015
1016        start = omp_get_wtime();
1017
1018        for(unsigned long i = 0; i < MaxNBodies;
1019        i++)
1020        {
1021            B = Bodies[i];
1022            B->fx = B->fy = B->fz = 0.0;
1023            //printf("\nCompute force on a given
1024        body=%d\n",i);
1025            Bodies[i]->zoneCost =
1026        compute_body_force(B);
1027        }
1028
1029        end = omp_get_wtime();
1030        printf("\nsequential time=%f\n",end-start);
1031
1032        //printbodies(Bodies,MaxNBodies); // here I add it to
1033        print the location of the bodies after the sequential execution
1034        of the cost zone
1035        }
1036        if(normal==1)
1037        {

```

```

1038
1039     printf("\n#####
1040 #####\n");
1041         printf("The Result for the normal");
1042
1043     printf("\n#####
1044 #####\n");
1045         unsigned long i=0;
1046         unsigned long threadID=0;
1047         start = omp_get_wtime();
1048
1049 #pragma omp parallel private(B,i,threadID)
1050     {
1051
1052         numThreads = omp_get_num_threads();
1053
1054         threadID = omp_get_thread_num();
1055         ThreadStartTime[threadID] =
1056 omp_get_wtime();
1057         //printf("threadstart= %lu
1058 %f\n",threadID,ThreadStartTime[threadID]);
1059         for(i = threadID* (MaxNBodies /
1060 numThreads); i < (threadID+1)*( MaxNBodies / numThreads); i++)
1061     {
1062         B = Bodies[i];
1063         B->fx = B->fy = B->fz = 0.0;
1064         Bodies[i]->zoneCost =
1065 compute_body_force(B);
1066     }
1067         ThreadEndTime[threadID] =
1068 omp_get_wtime();
1069         //printf("threadend= %lu
1070 %f\n",threadID,ThreadEndTime[threadID]);
1071         printf("thread#= %d thread time= %f work=%lu\n"
1072 ,threadID,(ThreadEndTime[threadID]-
1073 ThreadStartTime[threadID]),ThreadCDF[threadID]); // print EXEC
1074 time per thread
1075     }
1076         for (unsigned long i = 0; i < numThreads;
1077 i++)
1078     {
1079         ThreadAvgTime[i] += (ThreadEndTime[i]-
1080 ThreadStartTime[i]);
1081         //printf("thread time= %f, %f,
1082 %f\n",ThreadAvgTime[i],ThreadStartTime[i],ThreadEndTime[i]);
1083     }
1084         end = omp_get_wtime();
1085     }
1086     if(dynamic==1)
1087     {
1088         unsigned long i=0;
1089         unsigned long threadID=0;

```

```

1090
1091
1092                                     /*#pragma omp parallel private(threadID,B,
1093 i)
1094                                     {*/
1095                                     omp_set_num_threads(threads);
1096                                     start = omp_get_wtime();
1097                                     //numThreads = omp_get_num_threads();
1098
1099                                     /*threadID = omp_get_thread_num();
1100                                     ThreadStartTime[threadID] =
1101 omp_get_wtime();*/
1102                                     //printf("threadstart= %lu
1103 %f\n",threadID,ThreadStartTime[threadID]);
1104 #pragma omp parallel for schedule(dynamic) private(threadID,B,
1105 i)
1106                                     for(i = 0; i < MaxNBodies; i++)
1107                                     {
1108                                         B = Bodies[i];
1109                                         B->fx = B->fy = B->fz = 0.0;
1110                                         Bodies[i]->zoneCost =
1111 compute_body_force(B);
1112                                     }
1113                                     //ThreadEndTime[threadID] =
1114 omp_get_wtime();
1115                                     /* */
1116                                     /*ThreadEndTime[threadID] =
1117 omp_get_wtime();*/
1118                                     //printf("threadend= %lu
1119 %f\n",threadID,ThreadEndTime[threadID]);
1120                                     //for (unsigned long i = 0; i < numThreads;
1121 i++)
1122                                     //{
1123                                     // ThreadAvgTime[i] += (ThreadEndTime[i]-
1124 ThreadStartTime[i]);
1125                                     // //printf("thread time= %f, %f,
1126 %f\n",ThreadAvgTime[i],ThreadStartTime[i],ThreadEndTime[i]);
1127                                     //}
1128                                     end = omp_get_wtime();
1129                                     }
1130                                     //04-update velocity position
1131                                     double start5 = omp_get_wtime();
1132                                     for(unsigned long i = 0; i < MaxNBodies; i++)
1133                                         update_velocity_and_position(Bodies[i]);
1134                                     double end5 = omp_get_wtime();
1135                                     //05-Delete the oct tree and delete far bodies
1136                                     /*double start4 = omp_get_wtime();
1137                                     delete_far_bodies();
1138                                     delete_node_tree();
1139                                     double end4 = omp_get_wtime();*/
1140                                     //printf(" %f %f %f %f %f\n", (end2-start2), (end3-
1141 start3), (end-start), (end5-start5), (end4-start4));

```

```

1142
1143
1144
1145         //06-store current iteration
1146         /* sprintf(outfile, "data/frames3/frame%6d.txt",
1147 n+1);
1148         saveBodiesLocations(outfile);*/
1149         double start4 = omp_get_wtime();
1150         // delete_far_bodies();
1151         delete_node_tree();
1152         double end4 = omp_get_wtime();
1153
1154         timer_all += (omp_get_wtime()-timer_all1);
1155         end22 +=(end2-start2);
1156         end33 +=(end3-start3);
1157         endd +=(end-start);
1158         end55 += (end5-start5);
1159         end44 += (end4-start4);
1160     }
1161
1162         //printf(" \n");
1163         if((seq==1)||(dynamic==1))
1164         {
1165             printf("%lu %lu %f %f %f %f %f %f\n", ps,
1166 problemsize[ps] ,end22/steps,end33/steps,
1167 endd/steps,end55/steps,end44/steps, timer_all/steps );
1168             report1[ps][0] = endd/steps;
1169         }
1170         if((zone==1)||(normal==1))
1171         {
1172             printf("%lu %lu %f %f %f %f %f %f\n", ps,
1173 problemsize[ps], end22/steps,end33/steps,
1174 endd/steps,end55/steps,end44/steps , timer_all/steps);
1175             report1[ps][1] = endd/steps;
1176
1177             printf("<Thread no> <assignment> <Work> <time>
1178 \n");
1179             for (unsigned long i = 0; i < numThreads; i++)
1180             {
1181                 printf("%lu %lu %llu %f \n", i,
1182 ThreadEnd[i] - ThreadStart[i] + 1, ThreadCDF[i],
1183 ThreadAvgTime[i]/steps);
1184                 ThreadEndTime[i]=0;
1185                 ThreadStartTime[i]=0;
1186                 ThreadAvgTime[i]=0;
1187             }
1188         }
1189         printf("\n");
1190
1191         end22=0;
1192         end33=0;
1193         endd=0;

```



```

1194         end44=0;
1195         end55=0;
1196     }
1197     delete_AllBodies();
1198 }
1199
1200
1201
1202 int main(unsigned long argc, char* argv[])
1203 {
1204
1205     printf("Number Of steps: %lu \n",steps);
1206     printf("<step><PrbSize><NoofBodies><NoofNodes><Tree><Sort><
1207 Force><update><delete>\n");
1208     filepath = argv[0];
1209     Directory(filepath);
1210     /*seq=0;
1211     normal=0;
1212     zone=0;
1213     threads=1;
1214     generate=0;
1215     BarnesHutSimulation (filename_GalaxyKingModel_1M);
1216     printf("Normal\n");
1217     generate=1;
1218     seq=1;
1219     normal=1;
1220     dynamic=1;
1221     zone=0;
1222     //unsigned long ThreadingCount[10] = {8,32, 64,128, 240};
1223     unsigned long ThreadingCount[10] = {8};
1224     //signed long ThreadingCount[10] = {2,3,4,5,6,7}; //used in
1225 hpc as a back up
1226     for (unsigned long tc = 0; tc <1; tc++)
1227     {
1228         printf("Thread: %lu \n",ThreadingCount[tc]);
1229         threads = ThreadingCount[tc];
1230         BarnesHutSimulation (filename_GalaxyKingModel_1M);
1231         generate=1;
1232     }
1233
1234     //getchar();
1235
1236     printf("Dynamic\n");
1237     seq=1;
1238     normal=0;
1239     dynamic=1;
1240     zone=0;
1241     for (unsigned long tc = 0; tc <1; tc++)
1242     {
1243         printf("Thread: %lu \n",ThreadingCount[tc]);
1244         threads = ThreadingCount[tc];
1245         BarnesHutSimulation (filename_GalaxyKingModel_1M);

```

```

1246     */
1247     int ThreadingCount[1] = {16}; // here to add the number of
1248 threads that will be used in our case
1249     generate=1;
1250     seq=0;
1251     normal=0;
1252     dynamic=0;
1253     zone=1;
1254
1255     //BarnesHutSimulation (filename_GalaxyKingModel_1M);
1256     for (int tc = 0; tc < 1; tc++)
1257     {
1258         printf("Thread: %d \n",ThreadingCount[tc]);
1259         threads = ThreadingCount[tc];
1260         BarnesHutSimulation (filename_GalaxyKingModel_1M);
1261     }
1262     /*printf("N Sequential ZoneCost");
1263     for (unsigned long i = 0; i <2; i++)
1264     {
1265         for (unsigned long j = 0; j <2; j++)
1266         {
1267             printf("%lu ", report1[i][j]);
1268         }
1269         printf("\n");
1270     }*/
1271     return 0;

```

## References

- [1] Openshaw, S. *1 GeoComputation*. in *Geocomputation*. 2014. CRC Press.
- [2] Singh, A.K., et al. *Mapping on multi/many-core systems: survey of current and emerging trends*. in *Proceedings of the 50th Annual Design Automation Conference*. 2013. ACM.
- [3] Chen, L., *Exploring novel many-core architectures for scientific computing*. 2010, University of Delaware.
- [4] Albers, S., F. Müller, and S. Schmelzer, *Speed scaling on parallel processors*. *Algorithmica*, 2014. **68**(2): p. 404-425.
- [5] Borkar, S. and A.A. Chien, *The future of microprocessors*. *Communications of the ACM*, 2011. **54**(5): p. 67-77.
- [6] Ubar, R., J. Raik, and H.T. Vierhaus, *Design and Test Technology for Dependable Systems-on-chip*. 2010: IGI Global.
- [7] Salihundam, P., et al., *A 2 tb/s 6 4 mesh network for a single-chip cloud computer with dyfs in 45 nm cmos*. *Solid-State Circuits, IEEE Journal of*, 2011. **46**(4): p. 757-766.
- [8] Diaz, J., C. Munoz-Caro, and A. Nino, *A survey of parallel programming models and tools in the multi and many-core era*. *Parallel and Distributed Systems, IEEE Transactions on*, 2012. **23**(8): p. 1369-1386.
- [9] Han, L., et al. *A survey on cache coherence for tiled many-core processor*. in *Signal Processing, Communication and Computing (ICSPCC), 2012 IEEE International Conference on*. 2012. IEEE.
- [10] Vajda, A., M. Brorsson, and D. Corcoran, *Programming many-core chips*. 2011: Springer.
- [11] Hwu, W.-m., *What's ahead for parallel computing*. *Journal of Parallel and Distributed Computing*, 2014.
- [12] Gropp, W., E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. 1999: MIT press.
- [13] Chapman, B., G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. Vol. 10. 2008: MIT press.
- [14] Cramer, T., et al., *OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison*. 2012.
- [15] Williams, S., et al. *Optimization of geometric multigrid for emerging multi-and manycore processors*. in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012. IEEE Computer Society Press.
- [16] Pennycook, S.J., et al. *Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors*. in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 2013. IEEE.
- [17] Wu, Q., et al. *MIC acceleration of short-range molecular dynamics simulations*. in *Proceedings of the First International Workshop on Code Optimisation for Multi and many Cores*. 2013. ACM.

- [18] Park, J., et al. *Efficient backprojection-based synthetic aperture radar computation with many-core processors*. in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 2012. IEEE.
- [19] Haack, J., *A hybrid OpenMP and MPI implementation of a conservative spectral method for the Boltzmann equation*. arXiv preprint arXiv:1301.4195, 2013.
- [20] Saule, E., K. Kaya, and Ü.V. Çatalyürek, *Performance evaluation of sparse matrix multiplication kernels on intel xeon phi*, in *Parallel Processing and Applied Mathematics*. 2014, Springer. p. 559-570.
- [21] Dokulil, J., et al., *Efficient Hybrid Execution of C++ Applications using Intel (R) Xeon Phi (TM) Coprocessor*. arXiv preprint arXiv:1211.5530, 2012.
- [22] Hwu, W.-m.W., *Using hybrid shared and distributed caching for mixed-coherency GPU workloads*. 2013.
- [23] Abhishek Das<sup>1</sup>, D.M., Shraddha Desai<sup>3</sup>, Shweta Das<sup>4</sup>, Nisha Kurkure<sup>5</sup>, Goldi Misra<sup>6</sup>, Prasad Wadlakondawar<sup>7</sup>, *Analysis of Molecular Dynamics (MD\_OPENMP) on Intel® Many Integrated Core Architecture*. HPCS Group, Centre for Development of Advanced Computing, Agriculture College Campus, District Industries Centre, Shivajinagar, Pune 411005-India, 2012.
- [24] Intel Corporation, James Reinders, "An Overview Of programming for Intel Xeon Prcossor and Intel Xeon Phi coprocessor," 2012.
- [25] Intel Corporation, "Intel Xeon Phi", *Coprocessor Instruction Set Architecture Reference Manual, Vols. reference number 327364-001, Intel Corporation, September, 2012. Intel*. 2012.
- [26] Intel® C++ Compiler XE 13.1 User and Reference Guide , Document number: 323273-131US , Intel® C++ Composer XE 2013- Linux\* OS.
- [27] Intel, Knights Corner, *Software Developer Guide, Intel Corporation, 27, April, 2012*.
- [28] *U-pipe , V-pipe*. 2013; Available from: [http://www.dauniv.ac.in/downloads/CArch\\_PPTs/](http://www.dauniv.ac.in/downloads/CArch_PPTs/).
- [29] Garea, S.R. and T. Hoefler, *Modelling Communications in Cache Coherent Systems*. 2013.
- [30] Sanchez, L.M., et al. *A Comparative Evaluation of Parallel Programming Models for Shared-Memory Architectures*. in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. 2012. IEEE.
- [31] Strassen, V., *Gaussian elimination is not optimal*. *Numerische Mathematik*, 1969. **13**(4): p. 354-356.
- [32] Blair-Chappell, S. and A. Stokes, *Parallel Programming with Intel Parallel Studio XE*. 2012: John Wiley & Sons.
- [33] Li, J., S. Ranka, and S. Sahni. *Strassen's matrix multiplication on gpus*. in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. 2011. IEEE.
- [34] Badin, M., et al. *Improving numerical accuracy for non-negative matrix multiplication on GPUs using recursive algorithms*. in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 2013. ACM.

- [35] Müller, M.S., et al., *SPEC OMP2012—An Application Benchmark Suite for Parallel Systems Using OpenMP*, in *OpenMP in a Heterogeneous World*. 2012, Springer. p. 223-236.
- [36] Karniadakis, G., *Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation*. 2003: Cambridge University Press.
- [37] Appel, A.W., *An efficient program for many-body simulation*. SIAM Journal on Scientific and Statistical Computing, 1985. **6**(1): p. 85-103.
- [38] Barnes, J. and P. Hut, *A hierarchical  $O(N \log N)$  force-calculation algorithm*. 1986.
- [39] Greengard, L. and V. Rokhlin, *A fast algorithm for particle simulations*. Journal of computational physics, 1987. **73**(2): p. 325-348.
- [40] Feng, H., et al., *A Parallel Adaptive Treecode Algorithm for Evolution of Elastically Stressed Solids*. 2014.
- [41] Overman, R.E., et al. *Dynamic Load Balancing of the Adaptive Fast Multipole Method in Heterogeneous Systems*. in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. 2013. IEEE.
- [42] Tooto, P. and H.W. Loidl, *Parallel Haskell implementations of the N-body problem*. Concurrency and Computation: Practice and Experience, 2013.
- [43] Xu, T.C., et al. *Evaluate and optimize parallel Barnes-Hut algorithm for emerging many-core architectures*. in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. 2013. IEEE.
- [44] Zhang, J., B. Behzad, and M. Snir, *Design of a multithreaded Barnes-Hut algorithm for multicore clusters*. 2013, Tech. Rep. ANL/MCS-P4055-0313, MCS, Argonne National Laboratory.
- [45] Dekate, C., et al., *Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model*. International Journal of High Performance Computing Applications, 2012. **26**(3): p. 319-332.
- [46] Duy, T.V.T., et al., *Hybrid MPI-OpenMP Paradigm on SMP clusters: MPEG-2 Encoder and n-body Simulation*. arXiv preprint arXiv:1211.2292, 2012.
- [47] Lashuk, I., et al., *A massively parallel adaptive fast multipole method on heterogeneous architectures*. CommuNiCatioNs of the ACm, 2012. **55**(5): p. 101-109.
- [48] Winkel, M., et al., *A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale  $N$ -body simulations*. Computer Physics Communications, 2012. **183**(4): p. 880-889.
- [49] Zhang, J., B. Behzad, and M. Snir. *Optimizing the Barnes-Hut algorithm in UPC*. in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011. ACM.
- [50] Sundar, H., R.S. Sampath, and G. Biros, *Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel*. SIAM Journal on Scientific Computing, 2008. **30**(5): p. 2675-2708.
- [51] Küpper, A.H., et al., *Mass segregation and fractal substructure in young massive clusters—I. The McLuster code and method calibration*. Monthly Notices of the Royal Astronomical Society, 2011. **417**(3): p. 2300-2317.

## VITAE

**NABLUS, PALESTINE**

**MOBIL-PALESTINE: 00970-595324742**

**MOBILE-KSA: 00966-565409222**

**E-Mail: [allam\\_fatayer@yahoo.com](mailto:allam_fatayer@yahoo.com), [allam.fatayer123@gmail.com](mailto:allam.fatayer123@gmail.com),  
[g201003720@kfupm.edu.sa](mailto:g201003720@kfupm.edu.sa)**

**Nationality: PALESTINIAN**

**BOD: 17/6/1983**

### *Allam Abd Elghany Fatayer*

#### **Education:**

- Tawjihi Certificate in Science Field, GPA 90.1%
- 2001–2006 An-Najah National University, Nablus, Palestine  
Bachelor of Computer Engineering (CE) , Faculty of Engineering , Nablus , Palestine.
- 2011-2015 – King Fahed for Petroleum and Minerals (KFUPM) , Master Science of Engineering . Computer Engineering Department, Saudi Arabia, Dhahran.

#### **Experience:**

- 2 year experience working on Intel Xeon Phi cluster ( KAUST) and Multicore Cluster (KFUPM), During Master thesis. The following duties is accomplished
  - Install Servers.
  - Managing Linux OS ( NFS, SSH, Firewall, and LVM).
  - Installing Software and complex tool configuration
  - Complex Parallel Programming using OpenMP model.
  - Shell scripting.
- 4 year experience ( Dammam and Qatif Technical College)
  - Computer Network Trainer ( CCNA1, CCNA2, CCNA3, CCNA4, Network security, Wireless networks, Server 2003, Server 2008 and Linux).
  - Hardware Trainer ( Computer maintenance, Computer component and Computer security).
  - Technical computer department timetable leader.
  - Technical Computer Scientific committee coordinator.
- 3 year experience, Trainer (Jouf Technical college). Accomplished the following tasks
  - Technical Computer Department Timetable Coordinator.
  - Technical Computer Scientific committee coordinator.
  - Team Leader of the group which participated in the “ 4<sup>th</sup> technical conference and exhibition in Saudi Arabia and took the second Rank.
  - Lab Technical Support.

- Training Software ( JAVA, ORACLE, WEBPAGE DESIN)
- Training Hardware( Computer Components, Computer Maintenance, Network)

**Skills:**

- High ability to work under stress conditions.
- Good oral communication.
- Dynamic personality with motivation & initiative.
- Ability to work with team.
- Self-learner and pro-active working.

**Educational and training courses:**

- 5th of June – 24th of August. 2004  
Amra information technology, Nablus/Palestine. Cisco course (CCNA).
- 5th of June – 20th of June. 2005  
Communication and computer system (CCS), Amman/Jordan.  
Oracle certified professional for 9i (SQL).
- 21st of June – 10th of July. 2005  
Communication and computer system (CCS), Amman/Jordan.  
Oracle certified professional for 9i (PL\SQL).
- 11th of July – 11th August. 2005  
Communication and computer system (CCS), Amman/Jordan.  
Oracle certified professional for 9i (Forms), build internet application.
- 1 th of August – 15th of August. 2006  
General Organization for Technical education and Vocational Training  
Aljouf/Saudi Arabia  
Optical Fiber Applications And Installation.
- 20 hour training course  
General Organization for Technical education and Vocational Training  
Aljouf/Saudi Arabia  
Advanced java programming .
- 20 hour Training Course  
General Organization for Technical education and Vocational Training  
Aljouf/Saudi Arabia  
Different web page design concepts.
- 20 hours Training Course  
General Organization for Technical education and Vocational Training  
Aljouf/Saudi Arabia  
Management networks using (Windows Server 2003 )
- 160 hours Training Course  
General Organization for Technical education and Vocational Training  
Damam Technical College/Saudi Arabia  
Cisco Academy courses ( CCNA 1 , CCNA 2 , CCNA 3 , CCNA 4)
- 20 hours Training course, 2014  
King Fahed University for Petroleum and minerals  
Dahran/Saudi Arabia

## Penetration Testing (Ethical Hacking).

### **Programming language:**

- C solid programming skills , OpenMP parallel programming. ( Expert Level)
- GPU Programming (CUDA). (Beginner level)
- C++, Java (J2SE), Servlet.
- SQL, PL/SQL.
- Php, Action Script, Html, Java Script , CSS .
- Assembly language for "Intel Machine" and "MIPS machine" and Microcontroller.

### **Operating System:**

- Linux ( RHEL (solid understanding) , Ubuntu , SL)
- Windows Server 2003 , 2008.
- Windows XP , Vista , 7.
- VMware Cloud Computing.

### **Certified:**

- Cisco company , CCNA , Cisco certified Network Associative , 2009  
Exam no : 640-802 .
- Certified from Oracle University (Introduction to SQL)  
EXAM NO: 1Z0-007 ( Oracle 9i)
- Certified from Oracle University (Introduction to PL\SQL)  
EXAM NO: 1Z0-147 (Oracle 9i)  
These tow exams give me OCA (Oracle Certified Association 9i )

### **Publications**

#### **Journals:**

- Padding Free Bank Conflict Resolution for CUDA-Based Matrix Transpose Algorithm, Ayaz Ul Hassan Khan, Mayez Al-Mouhamed, Allam Fatayer, Anas Almousa, Abdulrahman Baqais and Mohammed Assayony, International Journal of Networked and Distributed Computing (IJNDC) (2014).

#### **Conferences:**

- Padding Free Bank Conflict Resolution for CUDA-Based Matrix Transpose Algorithm, Ayaz Ul Hassan Khan, Mayez Al-Mouhamed, Allam Fatayer, Anas Almousa, Abdulrahman Baqais and Mohammed Assayony, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2014).



- AES-128 ECB Encryption on GPUs and Effects of Input Plaintext Patterns on Performance, Ayaz Ul Hassan Khan, Mayez Al-Mouhamed, Anas Almousa, Allam Fatayar, Abdul Rahman Ibrahim and Abdul Jabbar Siddiqui, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2014).
  
- On the Scalability of Some Parallel Applications on Many-Core, Saudi Arabian High performance computing users group conference, [http://www.hpcsaudi.com/wp-content/uploads/2013/12/KFUPM\\_Fourth-Saudi-HPC.pdf](http://www.hpcsaudi.com/wp-content/uploads/2013/12/KFUPM_Fourth-Saudi-HPC.pdf). ( 2013)