APPLICATION MAPPING AND OPTIMIZATION FOR CMP BASED ARCHITECTURES

by

Betül Demiröz

B.S., Computer Engineering, Marmara University, 2002

M.S., Computer Engineering, Marmara University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering
Boğaziçi University
2011

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis supervisors Prof. Oğuz Tosun and Prof. Haluk Topçuoğlu for their support and guidance throughout my graduate study and completion of this thesis. I also want to thank Prof. Mahmut Kandemir (from Pennsylvania State University) for his feedback and support. I gratefully acknowledge Assoc. Prof. Can Özturan, Assoc. Prof. Arda Yurdakul and Assist. Prof. Esin Onbaşıoğlu for their participation to my thesis jury, their useful comments and feedback.

I gratefully acknowledge the financial support of The Scientific and Technological Research Council of Turkey (TUBITAK) with a research grant (Project Number: 108E035). Additionally a partial fund is provided by Bogazici University Research Fund -08HA101D. I also want to acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this thesis.

I want to thank my dear friends Fatma Ergin, Sanem Arslan, Işıl Öz and Berna Kiraz for their friendship and support over the years. Finally I want to thank my parents Fehamet and Şenol, my brother Dündar and my dear husband İsmail for their encouragement, patience and support.

ABSTRACT

APPLICATION MAPPING AND OPTIMIZATION FOR CMP BASED ARCHITECTURES

Chip Multiprocessors (CMPs) are becoming standard and primary building blocks for personal computers as well as large scale parallel machines, including supercomputers. In this thesis, our main focus is on performance-aware mapping and optimization of application threads onto multicore architectures. Specifically, we propose three different approaches, which are data-to-core mapping methodology, thread-to-core mapping methodology, and cache-centric data assignment methodology that includes data-to-thread mapping. For demonstrating data-to-core mapping methodology, we propose two novel parallel formulations for the Barnes-Hut method on the Cell Broadband Engine architecture by considering technical specifications and limitations of the Cell architecture. Our experimental evaluation indicates that the Barnes-Hut method performs much faster on the Cell architecture compared to the reference architecture, an Intel Xeon based system. To present thread-to-core mapping methodology, we propose a framework that uses helper threads running in parallel with application threads, which dynamically observe the behavior of application threads and their data access patterns. These helper threads calculate data sharing among application threads, cluster them to be mapped to available cores, use cache counters to calculate the efficiency of a mapping, and make the mapping decision after considering the execution needs. Our final methodology provides a locality-aware mapping algorithm, which targets to assign computations with similar data access patterns of an application to the same core. Our algorithm divides computations of the application into chunks to provide load balancing, and a set of chunks with high similarity is grouped into bins to provide data locality. We consider the sparse matrix-vector multiplication as the reference application.

ÖZET

YONGADA ÇOKLU-İŞLEMCİLİ MİMARİLER İÇİN UYGULAMA HARİTALAMASI VE ENİYİLEME

Yongada Çoklu-İşlemciler (CMP), kişisel bilgisayarların yanı sıra büyük ölçekli paralel makinaların ve süper bilgisayarların standart ve temel yapıtaşlarını oluşturmaya başlamıştır. Bu tezdeki temel amacımız, performansı arttıran haritalama ve optimizasyon yöntemleri geliştirerek, uygulama izleklerini çok çekirdekli mimarilere atamaktır. Bunu başarabilmek için üç farklı yöntem sunmaktayız, bunlar veri-çekirdek eşleme metodu, izlek-çekirdek eşleme metodu ve önbellek merkezli veri-izlek eşleme metodudur. Veri-çekirdek eşleme metodunda, Barnes-Hut algoritmasının bir CMP olan Cell Broadband Engine Mimarisinin teknik özelliklerini ve limitlerini göz önünde bulunduran iki özgün paralel formulasyonunu önermekteyiz. Yapılan deneysel değerlendirme, Barnes-Hut metodunun Cell mimarisi üzerindeki performansının karşılaştırma yapılan referans mimarisi olan Intel Xeon tabanlı sisteme göre belirgin oranda daha hızlı olduğunu göstermektedir. İzlek-çekirdek eşleme metodunu sunmak için, uygulama izlekleri ile paralel çalışan yardımcı izlekler kullanan ve dinamik olarak uygulama izleklerinin davranışlarını ve eriştikleri veri düzenini gözlemleyen bir sistem önerilmiştir. Yardımcı izlekler uygulama izleklerinin veri paylaşım miktarını hesaplayarak, bunları çekirdeklere eşlenmeleri için gruplara ayırır, eşlemelerin verimliliğini hesaplayabilmek için ön bellek sayaçları kullanır, ve çalışma zamanı ihtiyaçlarına bağlı olarak eşleme kararını alır. Önerilmiş olan son metodumuzda, benzer veri erişim şekline sahip olan hesaplamaları aynı çekirdeğe atamayı hedefleyen, veri yerelliğini sağlayan bir eşleme algoritması tasarlanmıştır. Önerilen algoritma verilen uygulamanın hesaplamalarını yükün eşit dağılımını sağlayabilmek için parçalara ayırır ve veri yerelliliğini sağlamak için yüksek benzerliğe sahip olan parçalar gruplandırılırlar. Metodun performansını ölçmek için, referans uygulama olarak seyrek matris-vektör çarpımı kullanılmıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES ix
LIST OF TABLES xiv
LIST OF SYMBOLS xv
LIST OF ACRONYMS/ABBREVIATIONS xvi
1. INTRODUCTION 1
1.1. Research Objectives and Contributions 2
1.2. Outline of the Thesis 5
2. SURVEY ON APPLICATION MAPPING AND SCHEDULING FOR MULTICORE ARCHITECTURES 6
2.1. Overview of Multicore Architectures 6
2.2. Application Mapping and Scheduling 11
3. DATA-TO-CORE ASSIGNMENT METHODOLOGY: A DEMONSTRATION ON IBM CELL ARCHITECTURE 14
3.1. N-body Problem and Barnes-Hut Method 15
3.2. IBM Cell Broadband Engine Architecture 17
3.3. Parallelization of Barnes-Hut Algorithm on the Cell Processor 20
3.3.1. Initial Parallelization of the Barnes-Hut Method 23
3.3.1.1. Domain Decomposition 23
3.3.1.2. Tree Construction and Force Calculation 23
3.3.1.3. Position and Velocity Updates 26
3.3.2. Enhanced Parallelization of the Barnes-Hut Method 26
3.3.2.1. Global Tree Construction 27
3.3.2.2. Tree Construction and Force Calculation 29
3.3.3. Experimental Work and Analysis of Results 30
4. THREAD-TO-CORE ASSIGNMENT METHODOLOGY: A DEMONSTRA-

TION ON GENERIC MULTICORE ARCHITECTURES 39
4.1. Proposed Framework for Dynamic Thread-to-Core Mapping 39
4.1.1. Clustering Thread 42
4.1.2. Cost and Benefit Analysis Thread 44
4.1.3. Counter Thread 46
4.1.4. Mapping Thread 48
4.2. Enhancements on Our Framework for Decreasing Overheads 48
4.2.1. Overhead of Thread Migration 48
4.2.2. Overhead of Helper Threads 50
4.3. Setup and Experimental Evaluation 51
4.3.1. Simulation Platform Simics 51
4.3.2. Applications 52
4.4. Results and Discussion 53
4.4.1. Scalability Results 54
4.4.2. Quantifying Overheads 56
4.4.3. Performance of Helper Threads 57
4.4.4. Sensitivity Analysis 59
5. A CACHE-CENTRIC DATA ASSIGNMENT METHODOLOGY ON GENERIC MULTICORE ARCHITECTURES 65
5.1. Proposed Algorithm for Locality-aware Dynamic Mapping 67
5.2. Setup and Experimental Evaluation 74
5.3. Results and Discussion 75
5.3.1. Performance of Dynamic Locality-aware Mapping 78
5.3.2. Sensitivity Analysis 81
6. CONCLUSIONS AND FUTURE WORK 85
APPENDIX A: IBM CELL BROADBAND ENGINE 88
A.1. IBM Cell Broadband Engine Architecture 88
A.1.1. Power Processing Element 88
A.1.2. Synergistic Processor Elements 89
A.1.3. Element Interconnect Bus 92
A.2. Application Development on IBM Cell Broadband Engine 92

A.2.1. Levels of Parallelism 94
A.2.2. Cell Programming Models 94
REFERENCES 98

LIST OF FIGURES

Figure 2.1. Conventional microprocessor. 8

Figure 2.2. A simple chip multiprocessor. 8

Figure 2.3. Shared-cache chip multiprocessors. 9

Figure 2.4. Multithreaded shared-cache chip multiprocessors. 9

Figure 3.1. Sequential version of Barnes-Hut algorithm. 18

Figure 3.2. Cell Broadband Engine Architecture [37]. 19

Figure 3.3. View within PPE for the initial parallelization. 21

Figure 3.4. View within SPE for the initial parallelization. 22

Figure 3.5. A typical step of the first parallel version with 14 particles using 4 SPEs. (a) Domain decomposition (b) Local tree construction in SPEs (c) Force calculation and position updates in SPEs. 25

Figure 3.6. View within PPE for the enhanced parallelization. 27

Figure 3.7. View within SPE for the enhanced parallelization. 28

Figure 3.8. A typical step of the second parallel version with 14 particles using 4 SPEs. (a) Global tree construction (b) Local tree construction, force calculation and position updates in SPEs. 30

Figure 3.9. Speedup with respect to different number of SPEs and PPE - initial parallelization. 31

Figure 3.10. Speedup with respect to different number of SPEs and PPE - enhanced parallelization. 32

Figure 3.11. Speedup of 8 SPEs relative to the Intel Xeon with respect to different number of particles. 33

Figure 3.12. Performance comparison with respect to number of particles given in the range [1024-4096]. 34

Figure 3.13. Performance comparison with respect to number of particles given in the range [8192-32768]. 34

Figure 3.14. Speedup of 8 SPEs relative to the Intel Xeon with respect to different λ values - initial parallelization. 36

Figure 3.15. Speedup of 8 SPEs relative to the Intel Xeon with respect to different λ values - enhanced parallelization. 38

Figure 4.1. The relationship among the four helper threads considered in our framework. 41

Figure 4.2. An example thread sharing and thread mapping scenario. (a) Weight sharing graph (b) Thread mapping for 2 cores (c) Thread mapping for 4 cores. 43

Figure 4.3. New mapping construction by the cost and benefit analysis thread. 45

Figure 4.4. Inline assembly code for Solaris running on serengeti architecture. 47

Figure 4.5. Gcc inline assembly code for Linux running on x86 Intel architecture. 47

Figure 4.6. Scalibility of Black-Scholes application with 16, 32 and 64 threads. 55

Figure 4.7. Scalibility of Jacobian application with 16, 32 and 64 threads. . . 55

Figure 4.8. Scalibility of SpMV application with 4 threads per core. 56

Figure 4.9. Overhead of running helper threads with 64 application threads on 16 cores for SpMV application. 57

Figure 4.10. Performance of helper threads running with application threads on 8 Cores for Black-Scholes application. 58

Figure 4.11. Performance of helper threads running with application threads on 16 cores for Black-Scholes application. 58

Figure 4.12. Performance of helper threads running with application threads for Jacobian Matrix Multiplication. 59

Figure 4.13. Performance of helper threads running with application threads for SpMV. 60

Figure 4.14. Performance of enhanced helper threads running with application threads for SpMV. 60

Figure 4.15. Performance of helper threads for different L1 cache configurations for Black-Scholes application. 61

Figure 4.16. Load distribution among cores for 64, 128, 256 application threads on 16 cores for SpMV application. 62

Figure 4.17. Normalized execution time of application threads on 8 cores for SpMV. 63

Figure 4.18. Normalized execution time of application threads on 16 cores for SpMV. 63

Figure 4.19. Dynamic distribution of 32 application threads to 8 cores for SpMV. 63

Figure 5.1. The layout of our framework including chunk and bin representations. 68

Figure 5.2. The algorithm for calculating similarities among chunks. 69

Figure 5.3. Chunk-to-bin assignment. 72

Figure 5.4. Calculating similarity values for the case of 4 chunks with 2 bins. . 73

Figure 5.5. Sparse matrix-vector multiplication algorithm with CSR storage. . 76

Figure 5.6. The shapes of input matrices used by SpVM application [78] (a)Boddy4, (b)Boddy5, (c)Boddy6, (d)Ncvxqp9, (e)Illc1033. 77

Figure 5.7. Performance of locality-aware dynamic mapping method and Linux scheduler for various input files. 80

Figure 5.8. Speedup values for different number of chunks. 81

Figure 5.9. Speedup values when 4K L1 cache / 16K L2 cache combination is considered. 82

Figure 5.10. Speedup values when 16K L1 cache / 64K L2 cache combination is considered. 82

Figure 5.11. Speedup values when 32K L1 cache / 128K L2 cache combination is considered. 83

Figure 5.12. Performance of locality-aware dynamic mapping method and Linux scheduler when maximum number of chunks are considered. 83

Figure A.1. SPE block diagram [55]. 89

Figure A.2. SPE Architecture [55]. 90

Figure A.3. Internal structure of Cell B./E. [57]. 92

Figure A.4. Example of eight concurrent data transfers [58]. 93

LIST OF TABLES

Table 3.1. Communication and computation times (in msecs.) for various phases of the enhanced parallelization of the Barnes-Hut method. . 35

Table 3.2. Execution time (in seconds) of the enhanced parallelization of Barnes-Hut method with compiler optimization flags for different number of SPEs and particles. 37

Table 4.1. Simulation parameters for performing experiments. 54

Table 5.1. Simulated architecture parameters of experimental study. 75

Table 5.2. Input matrices of SpMV application. 77

Table 5.3. L1 and L2 cache counts (in millions) for Boddy5 and Illc1033 input matrices. 79

Table 5.4. L1 and L2 cache counts (in millions) for the case of maximum number of chunks. 84

LIST OF SYMBOLS

$access_i$	Total number of data accessed by $thread_i$
$chunk_{ij}$	Bit vector representing data accessed by $chunk_i$
$sharing_{i,j}$	Total number of shared data between $thread_i$ and $thread_j$
W_{ij}	Similarity weight of $thread_i$ and $thread_j$
δ	Constant used in variation calculation of two mappings
$\Delta_{miss\ ratio}$	Miss ratio fraction of previous and current mapping
λ	Constant for MAC Criteria

LIST OF ACRONYMS/ABBREVIATIONS

Cell/B.E.	Cell Broadband Engine
CM	Current Mapping
CMP	Chip Multi-Processing / Chip Multi-Processor
CPI	Cycle Per Instruction
CPU	Central Processing Unit
CSR	Compressed Sparse Row
DMA	Direct Memory Access
DML	Device Modeling Language
EIB	Element Interconnect Bus
FFT	Fast Fourier Transform
GPU	Graphics Processing Unit
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
LS	Local Store
MAC	Multipole Acceptance Criteria
MFC	Memory Flow Control Unit
MM	Main Memory
MRc	Cache Miss Ratio of Current Mapping
MRp	Cache Miss Ratio of Previous Mapping
NM	New Mapping
NoC	Network-on-Chip
PPE	PowerPC Processing Element
SIMD	Single Instruction Multiple Data
SPE	Synergistic Processor Element
SpMV	Sparse Matrix-Vector Multiplication
SPU	Synergistic Control Unit
TLP	Thread Level Parallelism

1. INTRODUCTION

As the chip manufacturing technology moves toward deep sub-micron ranges, transistors become smaller, and operating frequencies keep getting faster. As a result, chip components are becoming increasingly unreliable. An architectural response to these critical challenges is the emerging Chip Multiprocessors (CMPs). A CMP contains multiple Central processing units (CPUs), an interconnection fabric, and some memory components, all packaged into a single chip. Almost all major chip manufacturers in the world are now considering CMP design and optimization, and CMPs have become the standard and primary building blocks for personal computers as well as large-scale parallel machines, including supercomputers. Nowadays, there are dual core chips on the market from major manufactures (Intel Montecito [1] and AMD Athlon [2]). Complex configurations with higher number of cores have also been delivered or prototyped, including Sun UltraSPARC T1 (formerly Niagara) [3], IBM's Cell [4], and Intel's 80-core TeraFlop [6], RAW [7], TRIPS [8], WaveScalar [9], and SmartMemories [10] in academia.

The focus of this thesis is to explore ways to map and adapt the execution of an application to the underlying hardware in CMPs. The assignment of application threads onto cores and the data they manipulate onto memory components is a challenging task. One option is to use static assignment of data and application threads to CMPs and keep this assignment throughout the execution of the application. With this static mapping, we are not able to capture data access patterns of application threads that are dynamically changing.

The other option is to use dynamic mapping strategies that can detect changes in data access patterns of application threads throughout the execution and can adjust the application threads to adapt dynamic data sharing patterns. Dynamic mapping strategies can be useful if their overheads can be kept low.

1.1. Research Objectives and Contributions

Our goal is the efficient exploitation of emerging CMPs using application mapping and adaptation techniques. The aim of our approach is to reduce execution latencies. To achieve this, we have designed mechanisms that take full advantage of the resources, especially the memory components available in CMPs. Our proposed work is demonstrated first using the IBM Cell chip multiprocessor, and then generic multicore processors that are simulated using an enhanced version of SIMICS.

We started our work by studying various static application and data mapping strategies for CMPs. We then decided on the mechanisms to collect runtime information about the state of the architecture and the state of the application. Our approach has both "static" and "dynamic" components for application mapping and optimization, targeting CMP architectures. These approaches can be listed as thread-to-core mapping and data-to-thread mapping. In our first approach, static thread-to-core mapping and dynamic data-to-thread mapping are considered. The second approach involves static data-to-thread mapping and dynamic thread-to-core mapping. In our final approach, we consider a cache-centric mapping that includes dynamic data-to-thread mapping.

Our first approach has three phases: code parallelization, thread-to-core mapping, and data-to-memory mapping. In the first phase, the behavior of the application is analyzed in detail, and two parallel implementations that consider the characteristics of the application are proposed. Additionally, both the CMP characteristics and the unique features of the target HW are considered in the implementation. These features include vector operations (most of the computations are SIMDized) and memory access operations that are overlapped with computation by using double buffering. Branches are also eliminated by using select bits, loop unrolling, and nonrecursive implementation. The second step maps threads to the available CPUs. In this phase, the number of threads created is equal to the number of cores used. In the third step, data is mapped to on-chip memory components. In this step, we exploit the affinity (data-computation relationship) as much as possible. Also, in this step, data is dynamically mapped to threads due to the size of the on-chip memory components and the dynamic nature of the application used.

In our second approach, we focus on the development of a framework that includes a dynamic thread-to-core assignment that aims to maximize data reuse of on-chip memory components. Because we use CMPs as our target architecture, effective thread-to-core assignment at runtime requires several adjustments. First of all, not every piece of information that could be collected through counters is valuable, so the data is processed before we can make adaptation decisions based on it. Second, inexpensive but effective analysis and decision algorithms are designed and implemented. An important question here is how to structure the decision mechanism. Performing effective adaptation decisions also requires predicting the application's data access pattern. Third, fast activation of optimizations at runtime is important. Costly optimizations cannot be implemented, as their overhead can offset the potential benefits.

Major challenges that are investigated as part of our second approach include the following: (i) given the runtime execution constraints and application characteristics, deciding what type of statistics we need to collect (statically and dynamically), (ii) determining mechanisms for collecting dynamic statistics during execution, and (iii) selecting the set of optimizations to apply. Our approach employs four helper threads to adapt the application execution to continuously changing runtime conditions. Dynamic mapping is triggered as a reaction to an interesting execution pattern, which presents opportunities for optimization. The goal of our framework is to eliminate performance bottlenecks of the given application by increasing the data reusability of on-chip memory components.

In our framework, four helper threads that run parallel with the application threads are used to achieve our goals. The responsibilities of the helper threads are the following:

- The clustering thread collects the data access pattern of all application threads and calculates their data sharing. This thread constructs clusters of application threads which have highest data sharing. These clusters are then assigned to cores to maximize data reuse of on-chip memory components.
- The counter thread makes use of performance counters and calculates the overall miss ratio of a specific interval. This helper thread uses the total hit and miss number of data read and write of on-chip memory components of all cores used.
- The cost and benefit analysis thread decides whether dynamic thread mapping is necessary, and if it is, this thread generates the next formation of threads. While making this decision, it predicts the behavior of the application in the near future based on the previous information collected through other helper threads. The metrics used in the decision process are the application threads' data sharing pattern, past execution history, and current counter data. The cost and benefit analysis thread is the main decision mechanism of the framework, so it coordinates all other helper threads.
- The mapping thread obtains the mapping decision from the cost and benefit analysis thread and uses this information to dynamically migrate threads among cores.

In our final approach, our aim is to obtain a general view of different parts of the application by using a locality analysis, which provides details of the cache requirements and data access patterns of application threads. In this work, our objective is to increase the probability of data reuse of on-chip memory components by assigning computations with similar data access patterns to same cores. It is built on a hierarchically data-to-thread and thread-to-core assignment methodology, which includes similarity calculation, data-to-bin, bin-to-thread, and thread-to-core assignment phases.

In similarity calculation, the input data is divided into smaller computations, and these computations are processed to find the total number of shared and distinct elements that will be used by application threads. Then data elements are clustered according to the highest data similarity into bins, which are dynamically assigned to threads. Similarity calculation and data-to-bin phases are completed before the execution of the application threads starts. As soon as threads begin their execution, bins are assigned to threads. This work improves the performance of the application by balancing the load among threads and increasing the cache reuse of on-chip memory components shared by application threads. To achieve load balance, data in bins are mapped to threads dynamically. We increase cache reuse by assigning the bins with maximum similarity to threads sharing the same on-chip memory component.

1.2. Outline of the Thesis

Chapter 2 provides a brief background of application scheduling and mapping for multicore architectures. This chapter also surveys various scheduling algorithms on parallel and distributed systems. Chapter 3 presents and evaluates our data-to-core assignment methodology. This chapter describes the N-Body problem and the Barnes-Hut Method, which is used as the base application for experimental evaluation. It also gives the details of the Cell Broadband Engine Architecture. Chapter 4 presents a novel thread-to-core mapping framework that considers four helper threads. It also discusses the methods used for cache optimization and gives details on three benchmarks used, namely the Black-Scholes application, Jacobian Matrix Calculation, and Sparse Matrix-Vector Multiplication. Chapter 5 focuses on a cache-centric data assignment methodology based on a locality analysis of data computations of applications. Finally Chapter 6 summarizes the contributions of the application mapping approaches proposed in this thesis.

2. SURVEY ON APPLICATION MAPPING AND SCHEDULING FOR MULTICORE ARCHITECTURES

In this chapter, we first give a brief overview of multicore architectures. This is followed by a survey on application mapping and optimization on CMP architectures.

2.1. Overview of Multicore Architectures

The performance of microprocessors has increased exponentially for two main reasons: the increased number of transistors available and increased parallelism exploited in software. The growing number of transistors has also increased the amount of power used. The processor designs should utilize the number of transistors available to enhance performance by minimizing design complexity and power usage.

By using the same number of transistors, we can obtain either a single high-performance processor running a single, high-performance thread or a group of simpler cores running multiple threads in parallel. If the same budget is used, multi-cores obtain higher performance compared to a monolithic processor; in addition, multi-core designs are simpler, and design and verification costs are lower. As a result, the new processor designs target CMPs.

Individual CPUs in a CMP architecture can be simpler than a single CPU-based system and can thus operate with lower frequencies and less aggressive supply voltages (as we have multiples of them), thereby helping with both the power and the reliability problem. Single-processor systems as shown in Figure 2.1 faced some major difficulties in further increasing processor performance, and CMPs overcome these problems [11]. These difficulties can be listed as follows:

- Memory Stall Latency: The speed of processor is increasing faster than the main memory (MM), and while data or instructions are fetched from memory, the

CPU waits for the memory operation to finish; as a result, memory stall latency determines a major part of total execution time. Previous designs proposed two solutions to solve this problem: larger caches and exploitation of instruction level parallelism (ILP). CMPs solve this problem by using thread level parallelism (TLP). While one thread uses the CPU, the other fetches data or instructions from memory, and for the entire core to be idle would mean that all threads have overlapping memory stalls, which has a very low possibility.

- Branch Prediction: Long pipelines are used in single-core systems, and when a branch miss-prediction occurs, the entire instruction is replaced with a new one, so it has a high penalty. CMPs support one pipeline in each core shared by multiple threads, so when a branch miss-prediction occurs during the execution of a thread, another thread uses the pipeline; meanwhile, the other thread fetches the new instruction from memory.
- Power Consumption: Power consumption increases exponentially as more transistors are used in core designs. CMP systems can yield the same performance by decreasing the amount of power used. Individual CPUs in a CMP architecture can be simpler than a single CPU-based system and thus can be operated with low frequencies and supply voltages, thereby helping with the power problem.
- Die Size: CMP cores are simple and small, and adding a new core to the current architecture results in minor increases in die size.
- Complexity: CMP cores are easy to design and verify, whereas single-core CPUs are much more complex and difficult to design.

There are different ways of implementing CMPs, but all CMPs contain multiple CPUs, an interconnection fabric, and some memory components, all packaged into a single chip. CMP implementation can vary according to shared components in the chip [12]. Some CMPs share only the system interface between cores as in Figure 2.2, so each core has its own L1 and L2 cache; while others (an example architecture is shown in Figure 2.3) may share an on-chip cache.

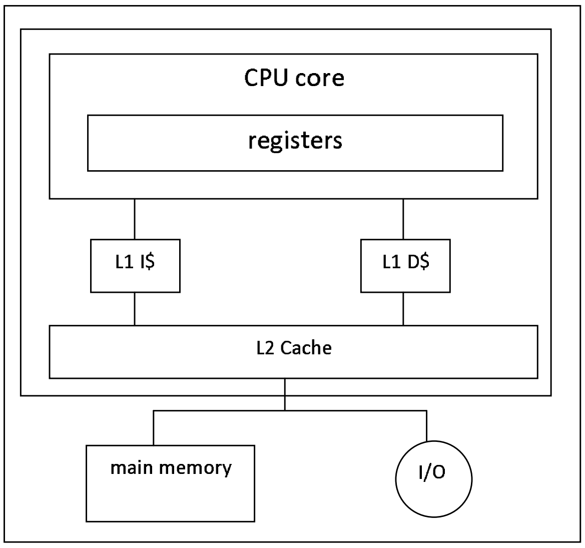


Figure 2.1. Conventional microprocessor.

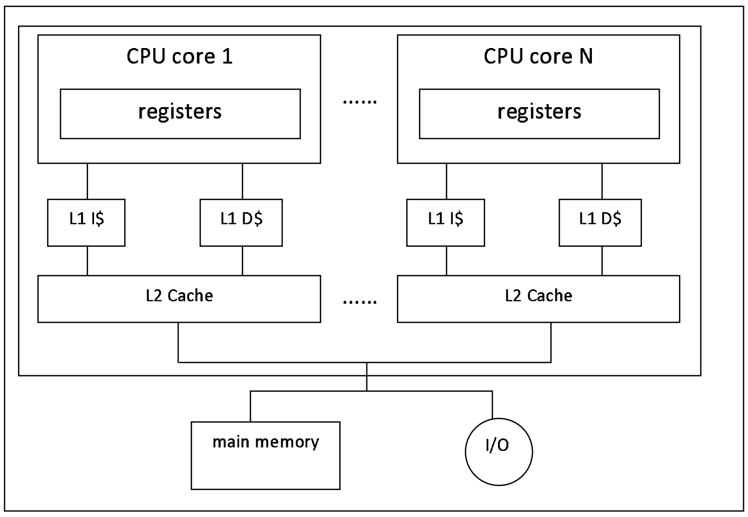


Figure 2.2. A simple chip multiprocessor.

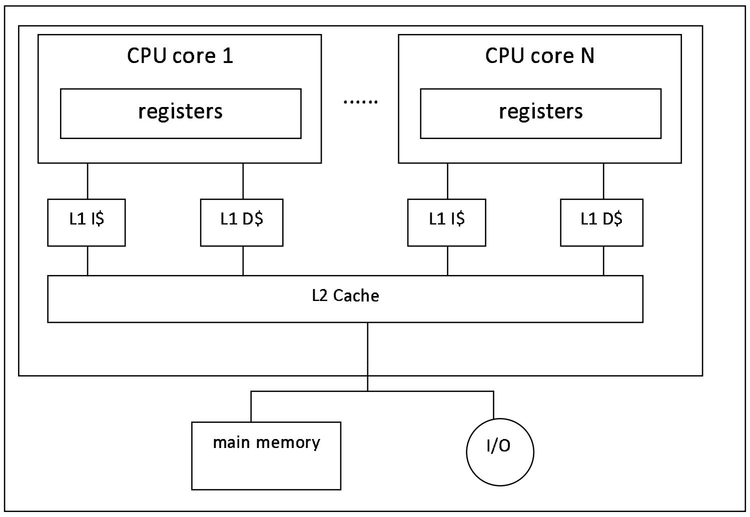


Figure 2.3. Shared-cache chip multiprocessors.

To distribute threads over the cores less in number then the cores, each core should support the execution of multiple threads, simultaneously. The CMP configuration in Figure 2.4 employs cores with multiple register files to provide simultaneous multithreading.

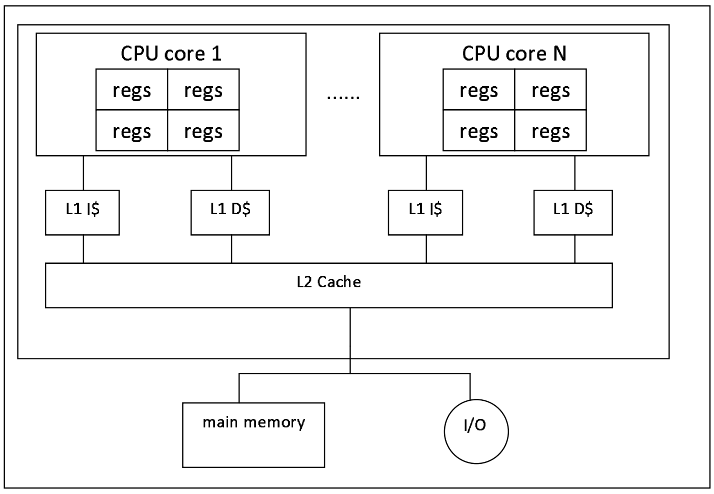


Figure 2.4. Multithreaded shared-cache chip multiprocessors.

The cores used in CMP architecture can be homogeneous or heterogeneous. Homogeneous CMPs are simple and easy to design, the same core is replicated several times, but heterogeneous CMPs contain high and low complexity cores. The size and strength of a replicated core varies according to application needs. Server applications focus on throughput per cost and power, so using many small low-power cores is attractive.In case applications require high performance, few large size cores may be preferred.

In reality, application's execution time requirements are not easily characterized; an application may have different phases (phases that can be executed on a simple core and phases that require a high-performance core), so different applications have different requirements, and these requirements may vary over time. Another core replication alternative is to use both high-performance and low-performance cores together to map application needs. Heterogeneous CMPs [13] may match the processor and system resources to each application's need efficiently. They may improve system throughput and reduce processor power. Heterogeneous CMPs may outperform homogeneous CMPs if a heterogeneous CMP can map each application to the core best suited to its performance demands and if it can answer to different workload demands, ranging from low TLP to high TLP.

Kumar et al. proposed a CMP architecture [14] that reduces processor power dissipation by using heterogeneous cores and mapping a given application to the most appropriate core to meet performance and power requirements. In this study, dynamic switch between cores is allowed, and the core selection depends on two metrics for power minimization: energy metric and energy-delay metric. In energy metric, the core that has the lowest energy consumption is chosen with the constraint that the performance should be within 10% of the fastest core in the chip. But for energy-delay metric, energy and response time have equal importance. Another study [15] examines policies for heterogeneous CMP architectures with and without multithreading cores.

In an efficient CMP design, the die size area and computational efficiency are two important factors. If topologically feasible resources can be shared, then the die area can be reduced, and overall computational efficiency can be improved as described in Kumar et al.'s study [16]. This work investigates the possible sharing of floating-point units, crossbar parts, instruction caches, and data caches between adjacent pairs of processors. The sharing of resources results in area savings if the resources shared are large enough that additional wiring overhead does not outweigh the area benefits.

2.2. Application Mapping and Scheduling

Efficient task scheduling and mapping is critical for achieving high performance in heterogeneous parallel and distributed systems. This problem has been shown to be NP-complete in general cases as well as in several restricted cases. Due to its key importance, the scheduling and mapping problem has been extensively studied, and various algorithms have been proposed in the literature. On the other hand, thread-level granularity is important for emerging architectures, including multicore systems, and efficient thread-to-core mapping mechanisms are crucial to improve the performance of multithreaded applications in multicore systems.

There are many scheduling algorithms for parallel and distributed systems in literature, and these algorithms are divided into two groups according to when scheduling action will be taken. In static scheduling, decision and mapping of tasks to processors is taken at compile time, whereas in dynamic scheduling, this decision is taken at runtime. Both static and dynamic assignment techniques will be considered in this work. Mapping of the algorithms in the literature directly, as they are to the CMP architecture is not suitable, because the communication cost in CMPs is much lower than that of conventional systems. Therefore, new scheduling algorithms considering CMP characteristics are proposed in the thesis.

Static scheduling techniques do not consider the runtime behavior of threads that vary throughout execution [17] and [18], so it is beneficial to use dynamic thread assignment techniques. Becchi et al. calculates the instructions per cycle (IPC) values on heterogeneous cores for each thread and uses the ratio of the IPC values to assign the application threads to cores at runtime [19].

Additionally, to efficiently utilize the underlying systems, it is important to observe the runtime behaviors of the running threads and to provide a dynamic thread-to-core mapping by exploiting thread migration between the cores. In Anderson et al.'s study [20] and Sondag et al.'s study [21], instead of using an applications execution trace, the similarity between programs' phases are used to make the thread assignment decision. This work contains two phases; the first phase uses a static analysis technique to group similar sections in a program, and the second phase deals with scheduling. Another dynamic scheduling method is given in Tam et al.'s study [22], which clusters the application threads that share the same cache access pattern to the same core. Cache access patterns are collected with performance monitoring units and are stored in a summary vector called shMap; and the similarity among threads' access patterns are measured using a similarity metric. If the similarity between two shMap vectors is greater than a given threshold, these two threads are assigned to the same cluster. After clusters are formed, each cluster is assigned to a core.

Process variation aware thread mapping algorithm is proposed in Hong et al.'s study [23]. This work focuses in the latency variations of identical processor cores and proposes a dynamic thread remapping algorithm which allows all processors to operate at their individual peak frequency. This algorithm has two phases, detection phase and stable phase. In detection phase the variations of execution latencies between application threads are detected. Based on the variations encountered, the application threads are remapped. As long as the variations of execution latencies stay the same, the algorithm remains in the stable phase. These two phases are repeated as long as the execution of the application continues. There are other studies that considers the effect of process variation on performance [24, 25].

For efficient use of heterogeneous multicore systems, runtime systems such as StarPU [26] are also developed. StarPU distributes parallel tasks to heterogeneous multicore architectures, and it provides an execution model that uses scheduling strategies to balance load and improve data locality among tasks.

Machine learning based approaches are also used for predicting the best thread-to-core mapping. A compiler based approach that maps a parallel program to a multi-core processor by predicting the number of threads and the scheduling policy, is proposed in Wang et al.'s study [27].

Data and computation mapping for multiprocessors are also widely studied. These algorithms propose data partitioning approaches applicable to a given multicore architecture. In Balasundaram et al.'s study [28] data partioning decisions are guided with a static performance estimator. Static and dynamic algorithms to partition the on-chip memory components are proposed in Kim et al.'s study [29]. Dynamic thread and data mapping approaches on CMP-based network-on-chip (NOC) architectures are also presented. One of these works [30] aims to reduce the distance between the location of the requested data and the core whose local memory contains it. In this study, a helper thread is used to make the thread-to-core assignment.

Cache performance is an important issue regarding the performance of CMPs and shared cache contention degrades performance of the application, so studies including cache behavior of the application parts have been introduced. One of these works introduces a locality model [31] which predicts the behavior of jobs and schedules them accordingly by using concurrent reuse distance [32]. To improve the management of the L2 Cache, CASC [33] schedules threads with low L2 miss rate and gives priority to threads that has low L2 space requirements.

Chen et al.'s study [34] analyze the performance of different thread schedulers to increase the hit ratio of off-chip memory components. In Chandra et al.'s study [35] several performance models are proposed to predict the effect of L2 cache sharing with multiprogram workloads. Pichel et al. [36] proposed a technique for increasing the locality of sparse matrix-vector multiplication (SpMV) application on multicore platforms.

3. DATA-TO-CORE ASSIGNMENT METHODOLOGY: A DEMONSTRATION ON IBM CELL ARCHITECTURE

IBM Cell Architecture [4,37,38] is an example of heterogeneous multi-core architectures, jointly designed by IBM, Toshiba, and Sony. The IBM Cell engine contains a traditional microprocessor called the power processing element (PPE), which deals with the orchestration and the coordination of the synergistic processing elements (SPEs). The tasks are performed by eight SPEs that work in parallel, and all components on the chip are connected via an element interconnect bus. IBM Cell is suitable for applications demanding high performance since it offers high computational power.

N-body is an important problem that can be applied to extensive applications from various domains in engineering and science, and it is one of the computation-intensive problems in the 13 dwarfs [46]. This problem simulates the motion of particles under pairwise interaction among n bodies for a predefined time period. At each time step, pairwise forces are calculated for all particles; this requires O(n^2) computations, which are not feasible when millions of particles are considered. Therefore, in order to reduce the complexity of the problem, many approximation algorithms are proposed. The Barnes-Hut method [47] is one of the most popular approximation algorithms, which reduces the computational complexity of the N-body problem to $O(n \log n)$. It is widely used due to its simplicity and easily programmable nature without requiring complicated data structures.

Many algorithms from a variety of domains have been parallelized and developed on the Cell architecture to take advantage of the performance and power utilization of the Cell processor. The potential of the Cell processor for several scientific computing kernels and the performance and power efficiency of the Cell architecture for these kernels have been presented [39]. The parallel implementation of fast fourier transform (FFT) on the Cell processor is also shown in the literature [40–42]. Additionally, a complexity model for algorithm designs on the Cell with an implementation of the list ranking problem is given in Bader et al.'s study [43] . The performance and design choices of the breadth-first search algorithm is explored in Villa et al.'s study [44]. Bio-informatics applications [45] also have been tested on the Cell processor. These studies highlight the performance and the architectural restrictions of the Cell processor for various applications.

N-body is an important problem that can be applied to extensive applications from various domains in engineering and science, and it is one of the computation intensive problems in the 13 dwarfs [46]. This problem simulates the motion of particles under pairwise interaction among n bodies for a predefined time period. At each time step, pairwise forces are calculated for all particles; this requires O(n^2) computations, which is not feasible when millions of particles are considered. Therefore, to reduce the complexity of the problem, many approximation algorithms are proposed. The Barnes-Hut method [47] is one of the most popular approximation algorithms, which reduces the computational complexity of the N-body problem to $O(n \log n)$. It is widely used due to its simplicity and easily programmable nature without requiring complicated data structures.

This work presents the dynamic data-to-core assignment methodology by mapping our parallel implementations of the Barnes-Hut algorithm onto the Cell BE Architecture [48]. Although Cell Architecture is suitable for computation-intensive parallel applications such as Barnes Hut algorithm, programming the Cell Architecture is quite difficult due to its architecture-specific limitations and empirical optimization schemes. While designing our parallel methods, these limitations are considered.

3.1. N-body Problem and Barnes-Hut Method

The N-body problem is used to simulate the evolution of n particles in a space by calculating the pairwise forces among them. Due to the interactions among particles, a net force is exerted on each particle, which causes them to move within a specified time. In each time step, with the movement of the particles, the forces are recalculated, because the force is dependent on the distance between particles. Some application areas of the problem are astrophysics, molecular dynamics, plasma physics, embedded SAR data processing, and protein folding.

At each time step, the N-body problem requires n^2 interactions; it calculates all pairwise forces, so the exact formulation of this problem requires O(n^2) computations. To reduce the complexity of the problem, many methods have been proposed. The Barnes-Hut method [47,49], which has a computational complexity of $O(n \log n)$, is a popular algorithm due to its simplicity. Another approximation algorithm is the Fast Multipole Method [50] with a complexity of $O(n)$, which has working principals similar to those of the Barnes-Hut method except that it uses complex data structures.

The Barnes-Hut method uses a hierarchical tree representation of space and reduces the number of interactions among particles by grouping relatively close particles under a single tree node. The sequential implementation of this method has two phases: *tree construction phase* and *force computation phase*. In the tree construction phase, the domain is represented with a tree in which the leaves are the particles themselves, and internal nodes are the clusters grouping nearby bodies (i.e., particles). This phase starts with an empty space, and particles are added to the domain one by one. For a 2D space, the domain is recursively divided into four equal sub-domains if the domain contains more than one particle. This process is repeated until each subdomain contains a single particle at each time step. The force acting on particles is calculated in the force computation phase. Instead of calculating pairwise interactions among the particles, the force between the nodes of the tree and particles is calculated using the distance between the particle and the center of mass of the node.

To cluster particles and represent the clusters with internal nodes, the center of mass and the total mass of the particles in the cluster are calculated, and this information is stored in each internal node. The net force on each particle is calculated by traversing all nodes of the tree, starting from the root. If the center of mass of an internal node is sufficiently far away from the particle, the particles contained in the internal node are approximated as a single particle, and the net force acting on the particle is calculated using internal nodes' mass and center of mass information. If the center of mass of an internal node is not sufficiently far away from the particle, then each of the sub-nodes of the internal node is explored and traversed recursively.

The multipole acceptance criteria (MAC) [47] determines whether a node is at a sufficient distance from a particle, and it is equal to the ratio of the dimension of the domain to the distance of the particle from the center of mass of the domain. If this ratio is less than the predefined constant λ, the node is at a sufficient distance from the particle and an interaction can be computed; otherwise, the node is expanded to its sub-nodes, and the force between the particle and center of mass of the sub-nodes is computed recursively. The speed and accuracy of the simulation is determined by setting a proper value for λ. The value of λ is in the given range $0 \leq \lambda \leq 1.0$; and if it is equal to 0, the Barnes-Hut application is turned into the N-Body problem in which all nodes are explored and the pairwise force among all particles is calculated. The increase in λ value decreases the number of nodes explored and also decreases the accuracy of the calculations performed.

Although there are various parallel implementations of the Barnes-Hut method in the literature [49,51], which are either for message-passing architectures [52] or shared-memory architectures [53], they can not be directly mapped to the Cell architecture. The sequential non-recursive Barnes-Hut algorithm given in Figure 3.1 is the basis of our parallel implementation running on the Cell architecture.

3.2. IBM Cell Broadband Engine Architecture

The Cell Broadband Engine (the Cell B./E.) [37, 38] is a high-performance architecture designed by Sony, Toshiba, and IBM that targets multimedia and gaming applications. The Cell is used in Sony's Play Station 3 gaming console, Mercury Computer System's dual Cell-based blade servers, IBM's QS20 - QS21 - QS22 Cell Blades, and Roadrunner, which has a speed of 1.06 Petaflops. The Cell is designed to increase

```
1. for time=0 to endTime do
2.     for i=0 to particleNumber do
3.         Insert particle i to the tree
4.         Update center of mass and total mass of each internal node on the way
5.     end for
6.     for i=0 to particleNumber do
7.         Add root node to visitList
8.         while visitList is not empty do
9.             Calculate distance between particle i and center of mass of node
10.            if center of mass of the node and particle i are distantthen
11.                Calculate force acting on particle i
12.            else
13.                Add children of node to visitList
14.            end if
15.        end while
16.    end for
17.    for i=0 to particleNumber do
18.        Update position and velocity of particle i
19.    end for
20. end for
```

Figure 3.1. Sequential version of Barnes-Hut algorithm.

microprocessor efficiency in terms of both power and performance. The clock speed of the Cell processor is 3.2 GHz and it has a single-precision peak performance of 204.8 Gflops/s and double-precision peak performance of 14.6 Gflops/s.

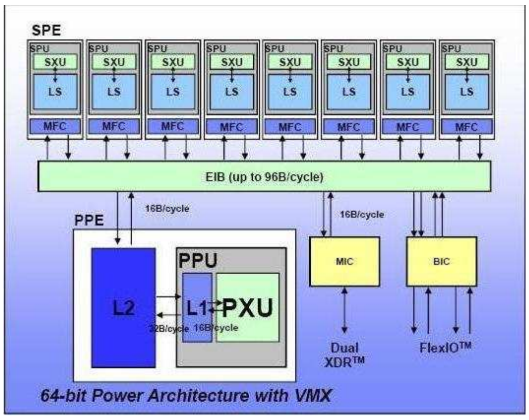


Figure 3.2. Cell Broadband Engine Architecture [37].

The Cell B./E. consists of a traditional microprocessor (power processing element-PPE), 8 smaller and simpler processors (SPEs) and an element interconnect bus (EIB) which connects the processors and provides access to main memory and I/O devices. General overview of the Cell Broadband Engine Architecture can be seen in Figure 3.2. The PPE is a dual-issue processor so it supports two-way simultaneous multithreading. Each SPE consists of a Synergistic Processor Unit (SPU), a Local Store (LS) and a Memory Flow Controller Unit (MFC). The instruction set of SPEs is designed to take advantage of 128-bit registers, and most of the instructions are single instruction multiple data (SIMD) instructions. Memory operations access 128-bits at a time even if the request data is 8-bits, so for an efficient implementation, the programmer should request 128-bits in each memory operation. SPEs are in-order processors with two instruction pipelines, namely the even pipeline and the odd pipeline. The even pipeline is responsible for arithmetic operations, and the odd pipeline deals with memory and branch instructions.

In a single clock cycle, SPEs can dispatch two instructions if these instructions have no data dependency. SPEs do not have a cache, but they have a 256 Kbyte LS which can be called private memory. Both the program and the data should be in LS to be executed. SPEs have no direct access to main memory and a direct memory access (DMA) controller is used to perform high bandwidth data transfers among the local store, main memory and other local stores. EIB connects all components of the Cell processor including the PPE, the SPEs, the main memory, and I/O. It supports a peak bandwidth of 204.8 Gbytes/s [56]. EIB is build of 16-byte wide four unidirectional rings, two in each direction.

The details of the Cell B.E. Architecture and Cell programming models are given in Appendix A.

3.3. Parallelization of Barnes-Hut Algorithm on the Cell Processor

This section presents common characteristics of our two parallel implementations of the sequential Barnes-Hut Algorithm (given in Figure 3.1) for the Cell Architecture. As part of parallelization, we aim to distribute data across SPEs. Since our workspace contains large number of particles and each SPE contains a 256KB Local Store, it is not possible to hold all data and the tree on each SPE. Therefore, the tree representing a part of the domain is gradually constructed in the SPEs and the force calculation is replicated for each local tree of the SPEs. In our initial parallelization, the number of particles in the domain is distributed equally among all SPEs to overcome both the limited size of the local stores and the time wasted on the synchronization barrier due to load imbalances among SPEs. The total number of nodes in the global tree is distributed among the SPEs in our enhanced parallelization.

To exploit the unique features of the Cell Architecture, a set of issues such as avoiding branches, vectorizing the corresponding code, and overlapping memory access with computation should be considered [61]. Since the Cell Engine has no branch prediction mechanism, we consider the non-recursive version of the Barnes-Hut algorithm as the starting point. Some of the if-then-else statements in the code are replaced with *select bits* instruction for eliminating branches. Additionally, the loops in the code are unrolled partially by decreasing the number of iterations and replicating the instructions in the loops. To reduce memory access latencies, DMA transfers and computations are overlapped through double buffering. The Barnes-Hut method performs the same computations on a large amount of 3D data repeating in each direction. In the force calculation phase (which is common in our two parallel implementations, as explained below), pairwise computations between particles and tree nodes are performed, where each computation is between a particle and the center of mass of a tree node. Each particle and the tree node is represented with a vector, and all operations performed are SIMDized. All data transferred between memory and the local store are stored contiguously in memory. Double-buffering usage is efficient for this type of implementation. After both the force calculation and the position and velocity updates are performed, the SPEs are synchronized using mailboxes that send and receive 32 bit messages.

```
1. Initialize data structures
2. Create SPEs
3. DMA: Initiate transfers to put blocks of particle coordinates to LS
4. Synchronization using mailbox
5. for time=0 to endTime do
6.     Arrange all particles in the sub-domains
7.     Use load information of the sub-domains and distribute load between SPEs
8.     Map sub-domains to SPEs
9.     Wait for SPEs to finish tree construction and force calculation
10.    Synchronization using mailbox
11.    Wait for SPEs to finish position and velocity updates
12.    Synchronization using mailbox
13. end for
```

Figure 3.3. View within PPE for the initial parallelization.

```
1. DMA: Initiate transfers to get blocks of particle coordinates
2. for i=0 to particleNumber do
3.     Determine which sub-domain particle i belongs to
4. end for
5. DMA: Put the data with its sub-domain information from LS back to MM
6. Synchronization using mailbox
7. for time=0 to endTime do
8.     while more sub-domains to visit
9.         while more blocks to process
10.            DMA: Load mass and coordinates of particles in the sub-domain from MM into LS
11.            for i=0 to subParticles do
12.                Insert particle i to the tree
13.                Update center of mass and total mass of each internal node on the way
14.            end for
15.        end while
16.        while more blocks to process
17.            DMA: Load mass and coordinates of all particles in the space
18.            DMA: Load force of all particles in the space
19.            for i=0 to allParticles do
20.                Add root node to visitList
21.                while visitList is not empty
22.                    Calculate distance between particle i and the center of mass of the node
23.                    if center of mass of the node and particle i are distant then
24.                        Calculate force acting on particle i
25.                    else
26.                        Add children of the node to visitList
27.                    end if
28.                end while
29.            end for
30.            DMA: Put force of all particles in the space to MM
31.        end while
32.        DMA: Put load information of sub-domain into MM
33.    end while
34.    Synchronization using mailbox
35.    while more blocks to process
36.        for i=0 to particleNumber do
37.            DMA: Get force exerted by other SPEs on particle i
38.            Calculate total force acting on particle i
39.            Update position and velocity of particle i
40.            Determine which sub-domain particle i belongs to
41.        end for
42.        DMA:Put new position, sub-domain and velocity of particles from LS to MM
43.    end while
44.    Synchronization using mailbox
45. end for
```

Figure 3.4. View within SPE for the initial parallelization.

3.3.1. Initial Parallelization of the Barnes-Hut Method

Our initial parallelization has three phases: *the domain decomposition phase*, *the tree construction and the force calculation phase*, and *the position and velocity updates phase*. The following subsections explain these phases in detail. The related pseudo-codes for the PPE-specific and SPE-specific parts are given in Figure 3.3 and Figure 3.4, respectively.

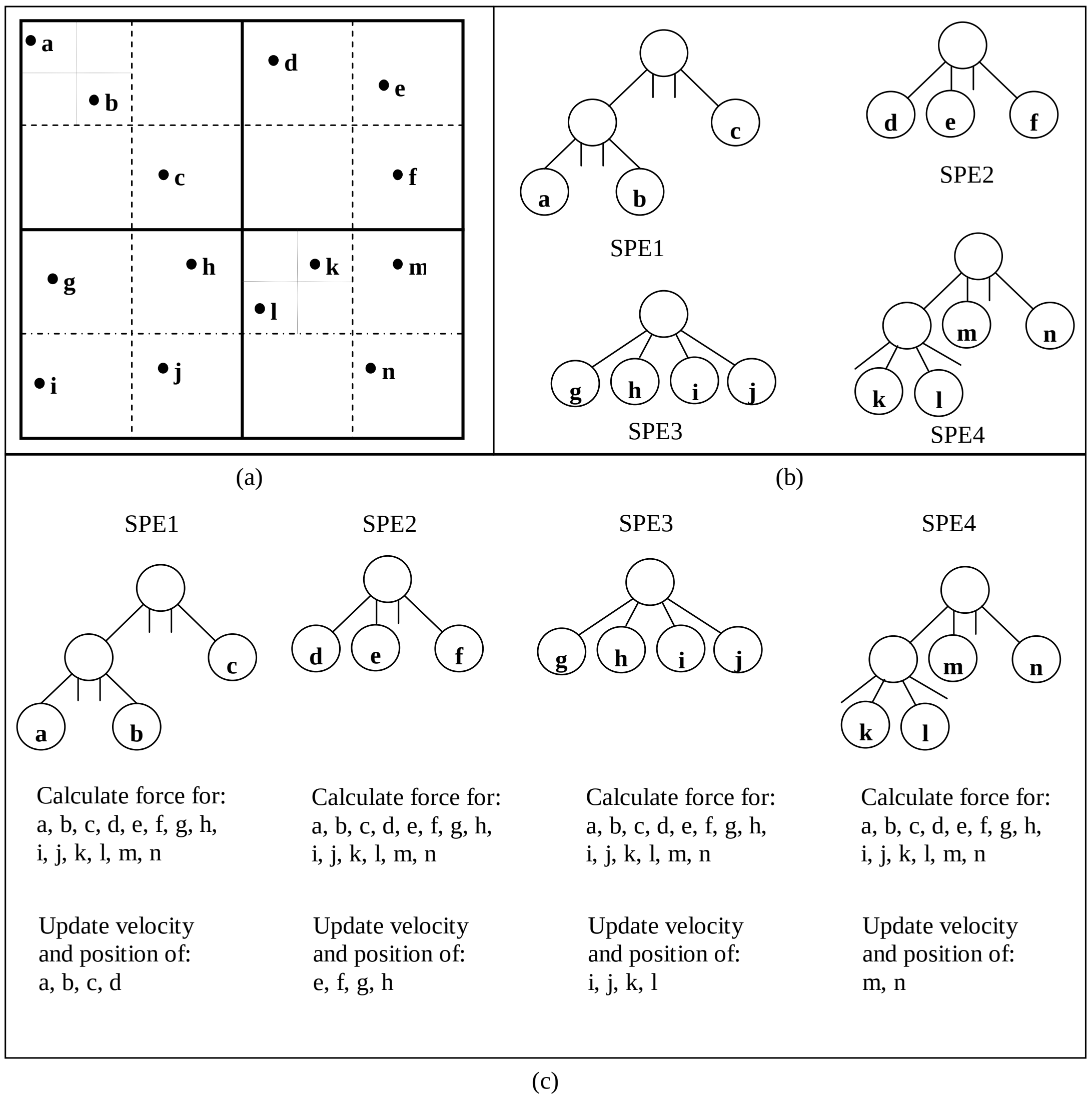
3.3.1.1. Domain Decomposition. The domain is decomposed into smaller parts (called $sub-domains$) in which each part contains a limited number of particles that can fit in the LS of each SPE. Initially, the PPE fetches the positions, velocities, and masses of particles, and it distributes the position of particles equally to the SPEs to obtain their sub-domain information. After the PPE obtains this information from the SPEs, it constructs sub-domains and uses the load information of sub-domains to give equal workload to each SPE. In the first iteration, the workload is divided into w equal units of data, where total number of particles in the workload is equal to $w * p$, and p is the number of SPEs used in execution. When the first iteration ends, the number of nodes explored by the local tree representing a sub-domain during force calculation in the previous iteration is used as the workload metric of the sub-domain for the current iteration. After the PPE completes sub-domain allocation, it sends the sub-domains to the SPEs on which they will operate. Finally, each SPE fetches its data from memory and starts its local tree construction.

3.3.1.2. Tree Construction and Force Calculation. In this phase, an octree representing the particles in a sub-domain is constructed by each SPE. Then, the force acting on all particles in the workload is calculated on all SPEs using their own local trees. Since the SPEs have local stores of limited size, they operate on many sub-domains by obtaining each sub-domain one by one. When the local tree representation of the sub-domain is finished, it starts the force calculation among the particles and the sub-nodes by traversing the tree starting from the root.

Interactions with the top nodes of the tree are calculated first, and then the sub-nodes are explored, if necessary. If the center of mass of an internal node is sufficiently far away from the particle, the particles contained in the internal node are approximated as a single particle, and the net force acting on the particle is calculated using the internal nodes' mass and center of mass information. To determine whether a node is sufficiently far away from a particle, MAC [47] is computed, which is the ratio of the dimension of the sub-domain to the distance of the particle from the center of mass of the given sub-domain. If a particle is too distant from a cluster of particles (where MAC is less than a predefined λ value), the particles contained in the cluster are represented with a single particle.

Each SPE is responsible for different sub-domains, and there will be more interactions between particles and nodes that are close to each other in the domain. As soon as the force calculation ends, each SPE writes back the calculated force values of all particles in its local tree to the memory and obtains the data of the new sub-domain. The tree construction and force calculation phase continues until all sub-domains assigned to the SPEs are processed.

Figure 3.5 shows the local trees that are constructed by each SPE for the given domain ((a) in Figure 3.5). There are 14 particles in this domain, which are processed by 4 SPEs. In the first phase, the domain information of particles is obtained by using their x and y coordinates, and each SPE is assigned an equal number of particles to work on. Therefore, SPE1, SPE2 and SPE3 receive 4 particles each, and SPE4 receives the remaining 2 particles. In the second phase, sub-domains are distributed among SPEs. There are 4 sub-domains, so each SPE handles one sub-domain. According to the number of particles in the sub-domains, SPE1 and SPE2 receive 3 particles, SPE3 and SPE4 receive 4 particles. Although SPE1 and SPE2 have the same number of particles, their tree construction times differ, as the depths of their trees are not the same.



Calculate force for:
a, b, c, d, e, f, g, h,
i, j, k, l, m, n

Update velocity
and position of:
a, b, c, d

Calculate force for:
a, b, c, d, e, f, g, h,
i, j, k, l, m, n

Update velocity
and position of:
e, f, g, h

Calculate force for:
a, b, c, d, e, f, g, h,
i, j, k, l, m, n

Update velocity
and position of:
i, j, k, l

Calculate force for:
a, b, c, d, e, f, g, h,
i, j, k, l, m, n

Update velocity
and position of:
m, n

Figure 3.5. A typical step of the first parallel version with 14 particles using 4 SPEs. (a) Domain decomposition (b) Local tree construction in SPEs (c) Force calculation and position updates in SPEs.

When the number of the nodes used to construct the local trees ((b) in Figure 3.5) are compared, SPE1 and SPE3 have 5 nodes, SPE2 has 4 nodes, and SPE4 has 6 nodes. Therefore, SPE2 has the lowest running time for tree construction, and SPE4 has the highest running time. It should be noted that SPE1 and SPE2 differ in terms of the running times for tree construction, although they have an equal number of particles in their sub-domains. In the force calculation phase, the local tree is explored for each particle ((c) in Figure 3.5). The running time of this phase is much longer than that of the tree construction phase. Additionally, the running time of this phase may vary among the SPEs due to the different number of nodes to be explored by themselves, which results in an unbalanced load among SPEs.

3.3.1.3. Position and Velocity Updates. After the force calculation phase is completed, the net force acting on each particle is calculated using all force values calculated on all local trees. Then, the particle's velocity and position values are updated accordingly. This process is repeated for a predefined number of iterations, and at the end of the simulation, each particle has a new position and velocity value.

3.3.2. Enhanced Parallelization of the Barnes-Hut Method

In our initial parallelization, implementation of the tree construction phase by using only the particles in the sub-domains causes loss of clusters of the domain. The tree construction phase is the heart of the Barnes-Hut algorithm. A good clustering of particles results in less computation for the force calculation phase. Therefore, our second parallel implementation is built on the idea of obtaining better clustering of the particles in the domain. Although the global tree is required in order to achieve this goal, it cannot be stored in the LS of the SPEs due to memory limitations. Therefore, the first phase of our enhanced parallelization is *the sequential tree construction phase*, where PPE constructs the global tree by using all particles in the domain, dividing global tree into sub-trees. The second phase of the enhanced parallelization is *the local tree construction and force calculation phase*, where SPEs use sub-trees to construct their local trees. The last phase, *the position and velocity updates phase*, is the same as the previous parallelization; therefore, it is not repeated in the following subsections. The related pseudo-codes are given in Figure 3.6 and Figure 3.7.

```
1. Initialize data structures
2. Generate global tree
3. Create sub-trees and store node information to tree array
4. Map sub-trees to SPEs
5. Create SPEs
6. for time=0 to endTime do
7.     Wait for SPEs to finish tree construction and force calculation
8.     Synchronization using mailbox
9.     Wait for SPEs to finish position and velocity updates
10.    Generate global tree
11.    Create sub-trees and store node information to tree array
12.    Map sub-trees to SPEs
13.    Synchronization using mailbox
14. end for
```

Figure 3.6. View within PPE for the enhanced parallelization.

3.3.2.1. Global Tree Construction. In this phase, PPE constructs a global tree that contains all the particles in the domain, and then it divides the global tree into sub-trees by using depth first search. The number of sub-trees depends on the total number of nodes generated in the global tree and the number of SPEs in use. In this parallelization, the total number of nodes are distributed equally among SPEs to have better load balancing. These sub-trees are stored as contiguous data in order to have a predictable memory access pattern. Each element in the array represents a node in the tree and contains the center, the total mass, the total number of children nodes, and the dimensions of the domain represented by the node. After PPE completes the generation of sub-trees, SPEs fetch their data from memory and start local calculations.

```
1.  for time=0 to endTime do
2.      while more trees to generate
3.          while more blocks to process
4.              DMA: Load tree nodes information from MM into LS
5.              for i=0 to nodeNumber do
6.                  Insert node i to the tree
7.              end for
8.          end while
9.          while more blocks to process
10.             DMA: Load mass and coordinates of all particles in the space
11.             DMA: Load force of all particles in the space
12.             for i=0 to particleNumber do
13.                 Add root node to visitList
14.                 while visitList is not empty
15.                     Calculate distance between particle i and the center of mass of the node
16.                     if center of mass of the node and particle i are distant then
17.                         Calculate force acting on particle i
18.                     else
19.                         Add children of the node to visitList
20.                     end if
21.                 end while
22.             end for
23.             DMA: Put force of all particles in the space to MM
24.         end while
25.     end while
26.     Synchronization using mailbox
27.     while more blocks to process
28.         for i=0 to particleNumber do
29.             DMA: Get force exerted by other SPEs on particle i
30.             Calculate total force acting on particle i
31.             Update position and velocity of particle i
32.         end for
33.         DMA:Put new position and velocity of particles from LS to MM
34.     end while
35.     Synchronization using mailbox
36. end for
```

Figure 3.7. View within SPE for the enhanced parallelization.

3.3.2.2. Tree Construction and Force Calculation. In this phase of the algorithm, each SPE obtains the number of sub-trees assigned to it and loads the nodes of the trees from memory to its local store one by one. The tree construction phase is simpler in this parallel implementation because the positions of the nodes are not searched in the whole local tree. Instead, this information is calculated by the PPE and is stored in the tree array. One disadvantage of this method is that additional variables are needed to store upper levels of the tree nodes, and this reduces the available space reserved for keeping local trees. On the other hand, this method has two advantages. It decreases the number of comparisons in finding the position of the particles in the trees thus reduces tree construction time. In addition, the upper levels of the local trees that contain the clustering information of the domain are included in the local trees generated. This reduces the total execution time of the force calculation phase by decreasing the number of comparisons in this phase. After local trees are constructed, the tree is traversed starting from the root, and the force acting on each particle in the domain is calculated. It should be noted that force calculation phase implementation is the same as described in Section 3.3.1.2. This phase continues until all local trees assigned to the SPEs are processed.

Figure 3.8 shows the global tree that is sequentially constructed by the PPE for the same domain used in initial parallelization ((a) in Figure 3.5). The global tree contains 20 nodes (excluding the root node), and the root node is replicated in all local trees assigned to 4 SPEs. The number of nodes distributed is 24, and each SPE should obtain approximately 6 nodes. While constructing the local trees, the PPE first considers keeping the clusters together; then, it takes the number of nodes into account. In this enhanced parallelization, the number of nodes is considered, which is different than the initial parallelization in which the number of particles is considered instead. This change aims to balance load among SPEs.

Figure 3.8 shows the local trees constructed by SPEs and the particles used in force calculation, as well as position and velocity update phases. The local tree includes upper level tree information ((b) in Figure 3.8). For a higher number of particles, this information is very valuable, because it decreases the number of nodes explored.

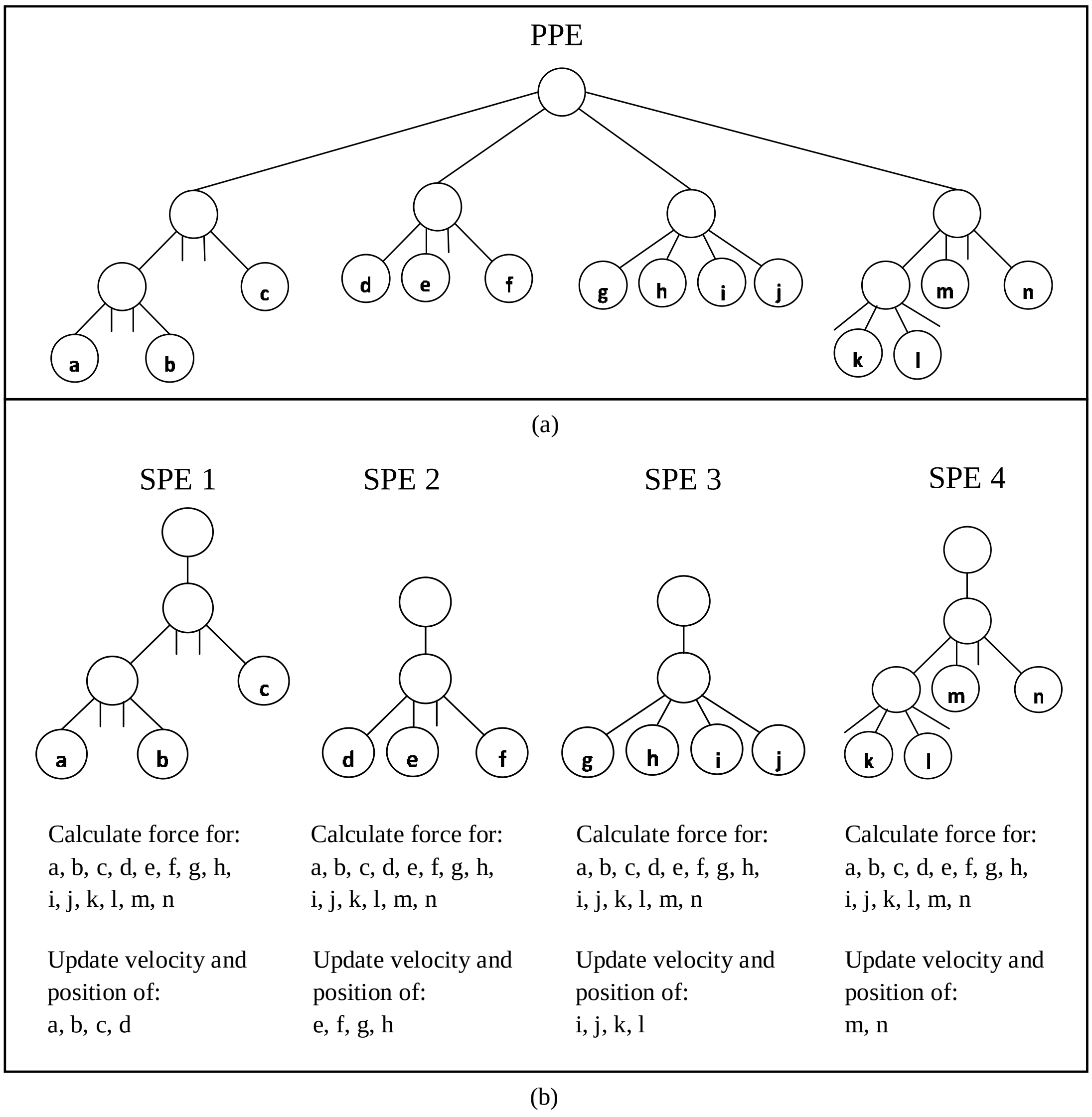

PPE
a
b
c
d
e
f
g
h
i
j
k
l
m
n
(a)
SPE 1
SPE 2
SPE 3
SPE 4
Calculate force for: a, b, c, d, e, f, g, h, i, j, k, l, m, n
Update velocity and position of: a, b, c, d
Calculate force for: a, b, c, d, e, f, g, h, i, j, k, l, m, n
Update velocity and position of: e, f, g, h
Calculate force for: a, b, c, d, e, f, g, h, i, j, k, l, m, n
Update velocity and position of: i, j, k, l
Calculate force for: a, b, c, d, e, f, g, h, i, j, k, l, m, n
Update velocity and position of: m, n
(b)

Figure 3.8. A typical step of the second parallel version with 14 particles using 4 SPEs. (a) Global tree construction (b) Local tree construction, force calculation and position updates in SPEs.

3.3.3. Experimental Work and Analysis of Results

The performance results presented in this section are from actual runs on an IBM Blade Center QS20 with two 3.2GHz Cell BE processors, 512 KB Level 2 cache per processor and 1 GB memory. Our code is compiled using gcc compiler in the Cell SDK

3.1 with -O3 optimization flag for performance analysis. As mentioned before, loop unrolling, double-buffering, and vectorization are used to decrease execution time on the SPEs. The code running on the PPE is also optimized by loop unrolling, branch elimination, and vectorization. In all experiments performed, the number of particles used varies from 1024 to 32768, the number of buffers used is 2, λ is 0.5, and the number of iterations is 50, unless otherwise stated.

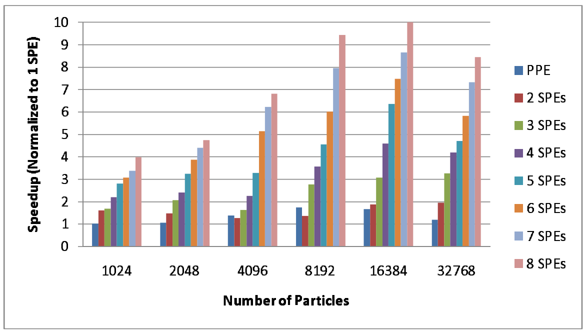


Figure 3.9. Speedup with respect to different number of SPEs and PPE - initial parallelization.

In the first set of tests, the performance of our algorithm is measured by varying the number of the SPEs used for different particle sizes. Figure 3.9 and Figure 3.10 show the speedup values that are normalized to a single SPE for the initial and enhanced parallelizations, respectively. The running time of the PPE is also included in the experiments. Considering that the Barnes-Hut algorithm is branchy, the PPE runs the code faster than a single SPE does. When 1024 particles are considered, 2 SPEs run 1.61, 4 SPEs run 2.2, and 8 SPEs run 3.98 times faster than 1 SPE in the initial parallelization. Similarly, in the enhanced parallelization, 2 SPEs run 1.93, 4 SPEs run 3.4, and 8 SPEs run 6.17 times faster than 1 SPE.

For the case of 32768 particles, 2 SPEs complete 1.95, 4 SPEs complete 4.18, and 8 SPEs complete 8.45 times faster in our initial parallelization. A speedup of 1.98 for 2 SPEs, 3.86 for 4 SPEs, and 7.32 for 8 SPEs is achieved in the enhanced parallelization. This shows that the application scales well as the number of the SPEs increases in both implementations, and load balancing is better in the enhanced parallelization.

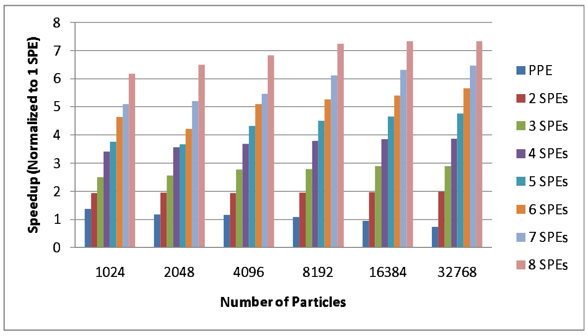


Figure 3.10. Speedup with respect to different number of SPEs and PPE - enhanced parallelization.

The performance of the SPEs with respect to cycle per Instruction (CPI) values is also measured using the IBM Cell Simulator. For the initial implementation of the method, the CPI values of the tree construction phase range between 1.99 and 2.03, and in the force calculation phase, they are between 1.72 and 1.79. When the second enhanced parallelization is considered, the CPI values of tree calculation and force calculation phases are 1.97 and 1.69, respectively. The percentage of total cycles stalled due to branch miss is equal to 25.7% for tree construction, and it is equal to 9.8% for force calculation. The CPI values are higher than the theoretical CPI value of 0.5 due to the branchy nature of the algorithm because in both phases the tree is traversed for each particle.

The second set of experiments presents a performance comparison of our implementations with that of the Intel Xeon 3.0 GHz processor running the Linux operating system. The Barnes-Hut code on Intel Xeon processor is compiled with gcc version 3.4.6 and -O5 optimization flag. The optimizations on the application code (except Cell specific optimizations including parallelization and vectorization) are applied to both architectures for a fair comparison. Additionally, the SPEs can construct trees with a limited number of particles since they have LS limitations, and this increases the number of nodes explored in the force calculation phase, whereas Intel Xeon has no such memory limitations, so all particles are considered simultaneously in the tree construction phase.

Due to the 256 KB LS size, each SPE constructs a tree consisting of at most 370 nodes, and a single sub-domain used for tree generation contains at most 208 particles in each LS in the initial parallelization. To ease workload distribution and load balancing, the workspace is initially divided into 512 sub-domains in each tree generation step. Then, the SPEs are assigned N sub-domains, where the total number of particles in each sub-domain is less than or equal to 208. As the number of particles increases, all nearby bodies may not be clustered in a single tree because of space limitations, and this increases the number of nodes explored in the force calculation step. As explained in Section 3.3.2, the bottleneck of this method is solved by distributing tree nodes containing cluster information in the enhanced parallelization. This approach decreases the number of nodes explored in the force calculation phase and improves the most compute-intensive phase of the algorithm, resulting in a speedup of 1.4 for 32768 particles.

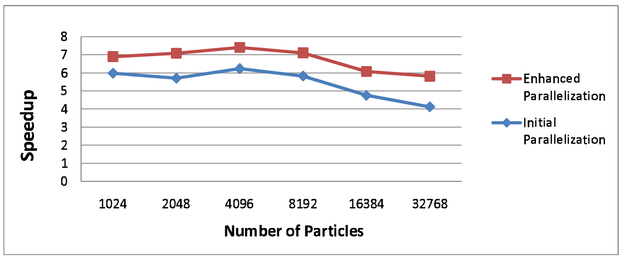


Figure 3.11. Speedup of 8 SPEs relative to the Intel Xeon with respect to different number of particles.

Figure 3.11 compares the speedup of the initial and enhanced parallelizations running on 8 SPEs with respect to Intel Xeon by varying the number of particles considered. When 4096 particles are considered, the Cell outperforms Intel Xeon with a speedup of 6.22 for our initial parallelization of the Barnes-Hut method, and it is equal to 7.4 for the enhanced parallelization. As the number of particles increases, we can no longer achieve an ideal speedup as there is an increase in the total number of local trees generated in a single step due to LS size restriction. The number of trees generated for 1024, 8192, and 32768 particles are 8, 64, and 200 respectively. As a result, all of the clusters of the domain cannot be stored in the LS; consequently, the number of clusters used in the force calculation phase decreases. When a lower number of clusters is considered, the number of nodes explored and the number of comparisons made increases, which decreases the performance of the system.

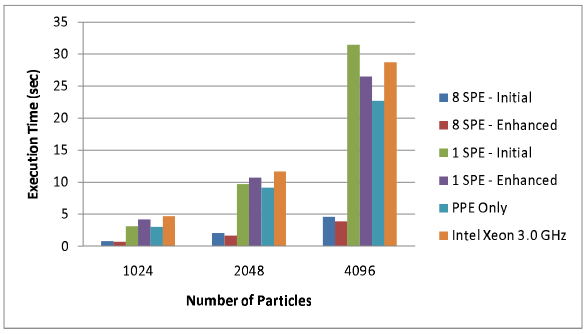


Figure 3.12. Performance comparison with respect to number of particles given in the range [1024-4096].

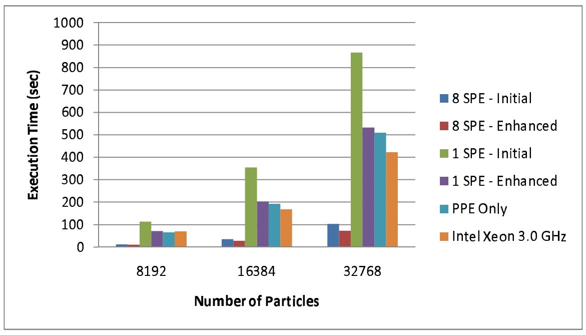


Figure 3.13. Performance comparison with respect to number of particles given in the range [8192-32768].

Figure 3.12 and Figure 3.13 compare the running time of parallel implementations on the Cell processor for 1 SPE, 8 SPEs, PPE only, and Intel Xeon cases by varying the number of particles in the domain. In Figure 3.12, the best performance is obtained when 8 SPEs of the enhanced parallelization are used. For 1024 and 2048 particles, Intel Xeon shows the worst performance. For the case of 4096 particles, 1 SPE shows the worst performance and the PPE only case outperforms both Intel Xeon and 1 SPE. As the number of particles increases, the performance of Intel Xeon increases, and it outperforms both PPE only and 1 SPE cases.

Table 3.1. Communication and computation times (in msecs.) for various phases of the enhanced parallelization of the Barnes-Hut method.

Phase Considered		**Number of Particles**					
		1024	**2048**	**4096**	**8192**	**16384**	**32768**
Tree Construction (SPE)	Computation	1.3	3.2	7.8	22.1	43.5	75.5
	DMA	0.04	0.1	0.2	0.6	1.1	2.1
Force Calculation (SPE)	Computation	18.3	68.1	198.3	953.6	3851.5	11573.9
	DMA	0.3	1.2	3.9	22.1	94.4	2854.7
Update Position (SPE)	Computation	1.1	2.2	5.4	7.3	20.3	41.6
	DMA	0.05	0.1	0.2	0.3	1.2	25.5
PPE-Specific	Computation	2.1	5.2	10.0	21.4	46.1	116.8
Synchronization (SPE)	Force	4.4	13.8	9.7	210.0	435.8	2052.7
	Update	1.7	2.2	3.2	5.1	9.7	26.2
Whole Application	Computation	22.8	78.8	221.5	1004.5	3961.6	11807.9
	Communication	6.5	17.5	17.4	238.3	542.2	2369.0

The average execution time of different phases of our enhanced parallelization for 8 SPEs is shown in Table 3.1, where the last two rows are for the overall computation and communication times. In this table, the "PPE-Specific" row denotes the time spent on global tree construction. As the number of particles in the workspace increases, the overhead of this phase compared to the total execution time decreases. Specifically, it takes 7.2% and 0.84% of the total execution time when 1024 and 32768 particles are considered. Similar behavior can be observed in the tree construction phase handled by SPEs; 4.4% and 0.54% of the total execution time is spent in this phase for 1024 and 32768 particles.

The update positions and velocities phase takes the least amount of time, as this phase includes SIMDized calculations and is straightforward. This phase provides good load balancing, so the synchronization barrier at the end of this phase takes only 0.1% of the total execution time for 32768 particles. The synchronization barrier at the end of the force calculation phase puts a higher overhead in running time, since loads are distributed to SPEs dynamically and it is harder to balance load in this phase. Specifically, 1.27% of the total execution time for 32768 particles is spent in this barrier. As the number of particles increases, nearly the whole computation of the algorithm is for the force calculation phase; namely this phase takes 88.3% to

98.9% of the total computation time of the method. The average communication to computation ratio for the given number of particles is equal to 0.19. Based on these results, the load balancing and communication of the application are acceptable.

As explained in Section 3.3.1.2, the MAC criteria calculates the ratio of the distance between the center of mass of the node and a particle to the dimension of the sub-domain. The term λ is used to decide whether to expand the corresponding node or to represent all particles that belong to the node with the center of mass of the node. The value of λ is in the given range $0 \leq \lambda \leq 1.0$; and if it is equal to 0, the Barnes-Hut application is turned into the N-Body problem in which all nodes are explored and the pairwise force among all particles is calculated. The increase in λ value decreases the number of nodes explored and also decreases the accuracy of the calculations performed. Since the application spends most of its time doing force calculation, the execution time decreases as the number of nodes explored decreases.

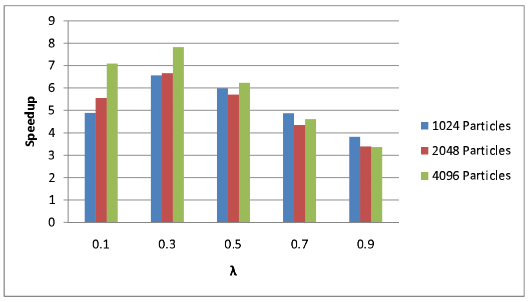


Figure 3.14. Speedup of 8 SPEs relative to the Intel Xeon with respect to different λ values - initial parallelization.

Figure 3.14 and Figure 3.15 compare the performance of the Cell processor of 8 SPEs with that of the Intel Xeon for different λ values. When λ value is close to 1, the algorithm does not consider the individual positions and mass values of the particles in the force calculation phase; rather, it uses cluster information regarding the upper level nodes. When the SPEs do not contain the upper level cluster information due to space limitations, they consider lower level cluster information, which increases the number of comparisons made. As a result, the speedup decreases when λ is 0.7 and

Table 3.2. Execution time (in seconds) of the enhanced parallelization of Barnes-Hut method with compiler optimization flags for different number of SPEs and particles.

Number of SPEs	**Number of Particles**	**Compiler Optimization Flag**			
		None	**O1**	**O2**	**O3**
1	1024	6.70	4.34	4.18	4.18
	2048	17.36	11.15	10.74	10.73
	4096	42.96	27.47	26.51	26,49
	8192	119.03	73.30	71.44	71.41
	16384	347.22	206.64	203.05	202.99
	32768	925.33	539.51	533.34	533.18
4	1024	2.04	1.27	1.23	1.23
	2048	5.01	3.12	3.01	3.01
	4096	12.00	7.45	7.20	7.18
	8192	32.03	19.35	18.89	18.85
	16384	91.62	53.68	52.79	52.71
	32768	242.61	139.65	138.22	138.06
8	1024	1.18	0.70	0.69	0.68
	2048	2.86	1.72	1.66	1.65
	4096	6.69	4.02	3.90	3.88
	8192	17.19	10.14	9.89	9.86
	16384	49.01	28.26	27.82	27.71
	32768	129.28	73.71	72.96	72.76

0.9. When the λ value is close to 0, cluster information of the lower level nodes is more frequently used, which is mostly available in the local trees of the SPEs. Therefore, the performance of the application increases. A test of 4096 particles with a value of 0.3 for λ outperforms other alternatives by providing a speedup of 7.83 for the initial parallelization and a speedup of 8.85 for the enhanced parallelization.

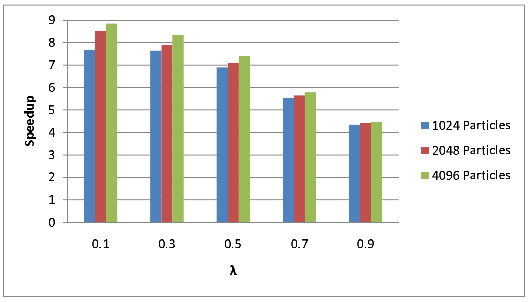


Figure 3.15. Speedup of 8 SPEs relative to the Intel Xeon with respect to different λ values - enhanced parallelization.

In the last set of experiments, the effect of compiler optimization is measured for 1, 4, and 8 SPEs for a variable number of particles, as shown in Table 3.2. Here, compiler flag -O1 is used to reduce the code size and execution time. The compiler increases code performance without compromising on code size by applying the -02 compilation flag. Additionally, the highest level of optimization is achieved using the -03 compiler flag, which is our standard choice in the previous experiments. The Barnes-Hut algorithm is branchy and with the usage of the -O1 flag, the compiler tries to optimize loops and if statements. It rearranges program code and minimizes branches. The Cell architecture has no branch mechanism, so the best performance improvement is achieved with this compiler flag. When compiler flag -O1 is used, a speedup of 1.75 is obtained; for compiler flags -O2 and -O3, a speedup of 1.77 is achieved, when 8 SPEs and 32768 particles are considered.

4. THREAD-TO-CORE ASSIGNMENT METHODOLOGY: A DEMONSTRATION ON GENERIC MULTICORE ARCHITECTURES

In our framework, we propose a dynamic thread-to-core mapping strategy based on the runtime characteristics of the threads. The policy focuses on the effective usage of caches by assigning threads that commonly share data to the same core. We also prevent thread migration unless it is highly beneficial for the performance of the system. In our thread-to-core mapping framework, four helper threads are used to monitor data access pattern of application threads and map threads with highest data sharing to the same core. Specifically, the first helper thread tracks data accesses of application threads and groups threads that share common data. The second helper thread collects cache statistics and calculates cache miss ratio between two consecutive mappings to see which mapping is more beneficial. The responsibility of the third helper thread is to migrate threads between cores. The final helper thread makes the dynamic mapping decision after considering cluster information, and the cache statistics gathered from other helper threads. Our simulation environment contains CMPs with 8 and 16 cores, each of which have a private L1 cache and L2 cache. For testing the performance of the dynamic thread-to-core mapping policy, the Black-Scholes application from the Parsec Benchmark Suite, Jacobian Matrix Calculation, and Sparse Matrix-Vector Multiplication are considered with 16, 32, 64, 128, and 256 threads.

4.1. Proposed Framework for Dynamic Thread-to-Core Mapping

Thread-to-core mapping is a challenging issue to effectively utilize multicore architectures for multithreaded applications. Additionally, the scheduling decisions of applications may change from one application to another. If the threads of an application share only a small amount of data except at certain communication points, they can be considered independent threads; so these threads can be distributed among different CPUs (or cores). On the other hand, if the threads of an application share data heavily, mapping these two threads to the same core (or nearby cores with common caches) improves the performance of the given application. Therefore, data sharing capabilities of threads can be considered for the task-to-core mapping of applications.

Theoretically, thread-to-core mapping can be considered as a clustering problem. If we have n threads and p cores, the aim is to construct p clusters (each of which contains $\lceil n/p \rceil$ threads). After these clusters are constructed, the second phase is to assign each cluster to a core (which core does not matter, but cluster-to-core mapping should be 1-to-1). As a static compiler point of view, a data sharing graph can be constructed as follows:

Given a graph $G(V, E)$, in which each vertex v in V corresponds to a thread and every edge e in EÂ indicates that the threads connected shares data, the sharing amount can be shown with a weight attached to each edge. After this graph is generated, any clustering algorithm can construct p clusters. The main problem is whether amount of data sharing between two nodes (threads) can be determined correctly at compile time, and in the most general way, this is a very hard problem. A method to ease the compiler's work is to profile the application program beforehand (by using sample input data), and calculate edge weights. If profile input is the same or very similar to the real input, this method can be highly effective.

On the other hand, it is important to observe the runtime behaviors of the running threads and provide a dynamic thread-to-core mapping by exploiting thread migration between the cores. We propose a dynamic thread-to-core mapping framework that is provided by the usage of helper threads. The strategy here is running helper threads parallel with the application threads, so that as soon as the program starts, the helper threads are spawned and can start working with the application threads.

These helper threads construct clusters that provides best data sharing by using the data sharing information of the application threads, and then by using the performance of the clusters, they map the application threads to cores. The helper threads can also change the thread-to-core mapping dynamically during program execution if it is beneficial to the application in terms of cache performance. While a program is in execution, it passes through different phases, and the ideal thread-to-core mapping may change. Helper threads intercept these changing points immediately and change current mapping to the desired thread-to-core mapping.

We consider four helper threads, which are *the clustering thread*, *the cost and benefit analysis thread*, *the counter thread*, and *the mapping thread*. The relationship among the helper threads is given in Figure 4.1. In the following subsections, the helper threads are explained in more detail.

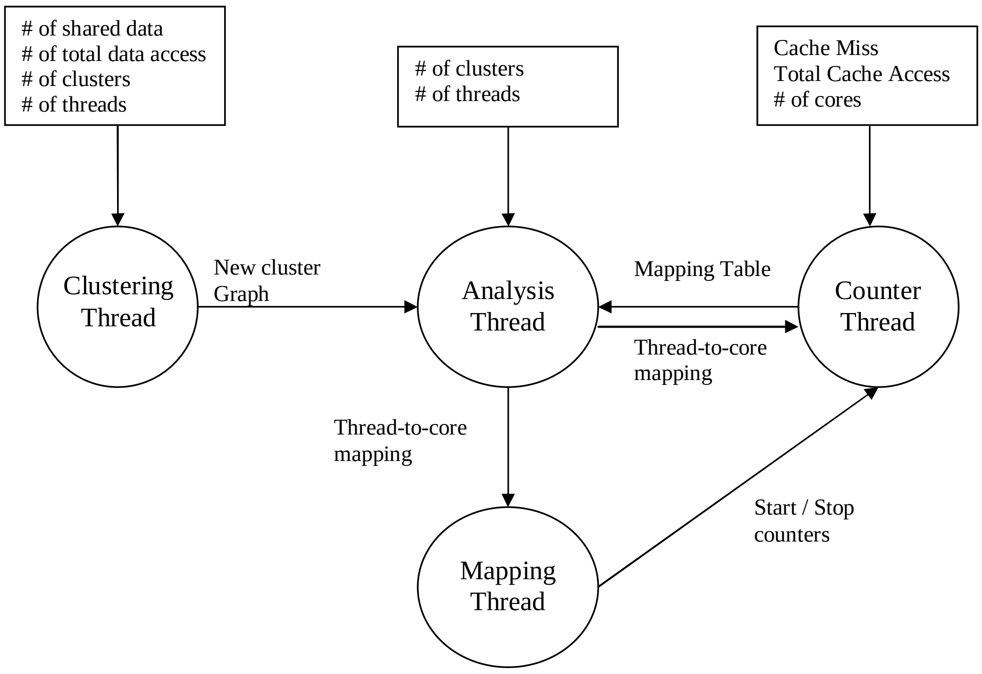


Figure 4.1. The relationship among the four helper threads considered in our framework.

4.1.1. Clustering Thread

The main job of the clustering threads is to group threads that will be mapped to each core by using the data sharing information that is collected throughout execution time among thread pairs and to construct clusters of threads. This thread calculates the weight among thread pairs by using the data sharing information among application threads as given in the following equation

$$W(i,j) = \frac{Sharing(i,j)}{Access(i) + Access(j)}, \tag{4.1}$$

where $Sharing(i,j)$ is the total number of accesses to the common shared data among thread i and thread j, $Access(i)$ is the total number of accesses done by thread i, and $Access(j)$ is the total number of accesses done by thread j. It should be noted that application threads with high number of data sharing, have higher weight values.

After the weights between threads are calculated, these weights are binned and represented with an integer number in the range [1..5]. If the weight is represented with 5, the thread pair is accessing common data extensively; conversely, a value of 1 for a weight value shows that the data sharing between thread pairs is very low. By using the binning information, data sharing graph is generated. Figure 4.2 shows an example of a data-sharing graph with 2 and 4 clusters (i.e., cores).

The data sharing graph is considered by both the clustering thread and the cost and benefit analysis thread. The clustering algorithm first selects the thread pairs that have a sharing weight of 5 and puts them in the same cluster. When the threads that have a weight information of 5 are all placed in a cluster, the binning value is decreased, and a suitable cluster is found for the threads that have not been assigned to a cluster. The cluster construction algorithm continues until all threads are assigned to a cluster.

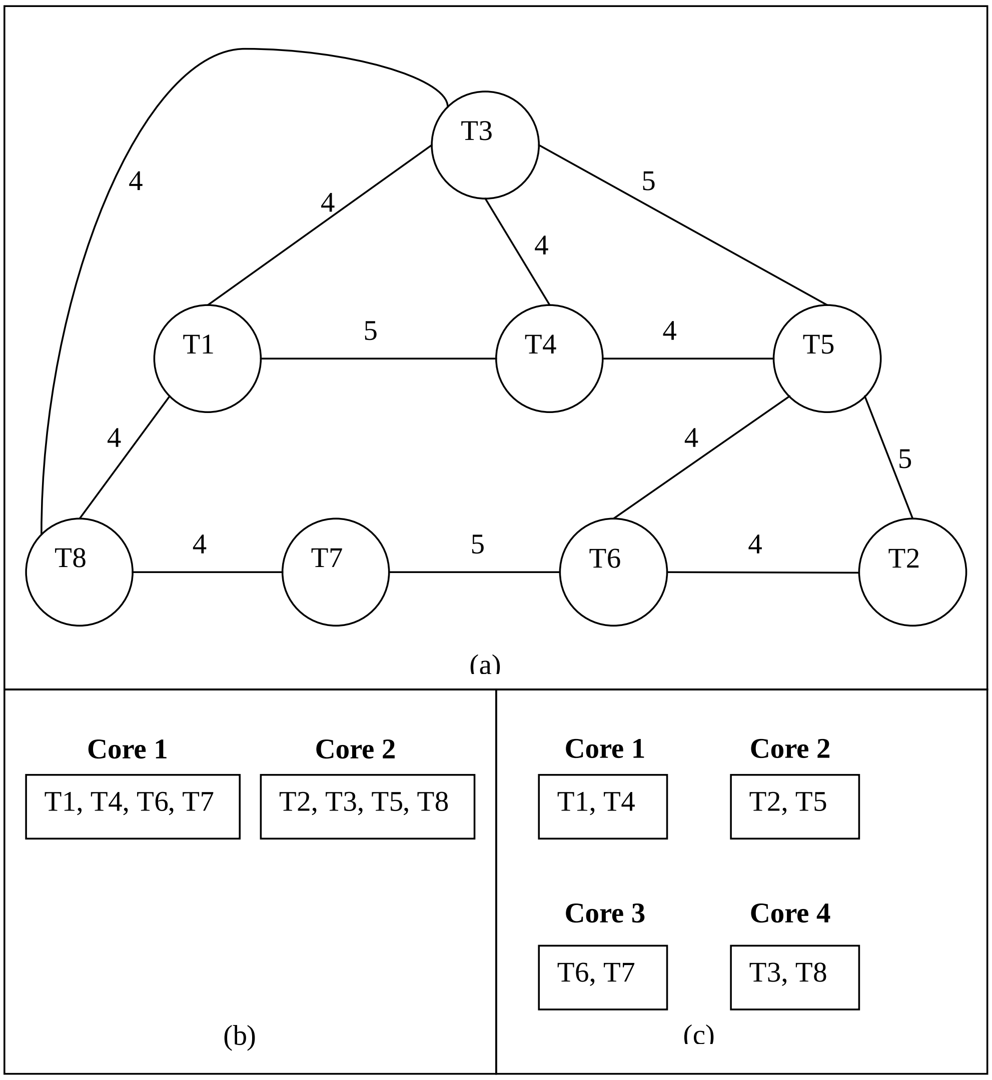


Figure 4.2. An example thread sharing and thread mapping scenario. (a) Weight sharing graph (b) Thread mapping for 2 cores (c) Thread mapping for 4 cores.

After the clusters are formed, the helper thread starts to collect new data sharing information and starts constructing the cluster that will be used in the next step, and this thread continues its execution until the end of the application.

4.1.2. Cost and Benefit Analysis Thread

The cost and benefit analysis thread makes the main decision for changing thread to core mapping dynamically. It uses the data sharing graph and the new cluster constructed by the clustering thread, and the mapping table formed by the counter thread. When it decides on the new thread-core mapping, it sends this information to the mapping thread.

This thread starts execution by using the new cluster information. If the new cluster information is different from that of the current one, the helper thread searches for the $< CM, NM >$ pair, where CM represents the current mapping and NM represents the new mapping in the mapping table. If it cannot find the <CM, NM> pair, the new mapping information is generated by using the new cluster information. If it is able to find the $< CM, NM >$ pair, the mapping has changed from current mapping to new mapping previously throughout execution, so the helper thread looks up for the benefit obtained from this change in the mapping table. If the benefit is greater than a given threshold value δ, changing the current mapping to the new mapping is beneficial, so the new mapping information is generated by using the new cluster information. If the variation between the two mappings is less than δ, changing the current mapping to the new mapping is disadvantageous, so the helper thread constructs a new mapping as shown in Figure 4.3.

In order to construct a new mapping, the new cluster formed by the clustering thread, current mapping and data sharing graph are used. While constructing new mapping, the criteria that should be considered is mapping threads that are extensively sharing data to the same core and decreasing extra overhead of thread migration. This algorithm starts with removing the thread pairs with the lowest data sharing from the

```
1. $counter = 5$
2. while ($counter > 5$) do
3.     Hash $< CM, NM >$ into table.
4.     if ($< CM, NM >$) entry does not exist then
5.         Put $NM$ into table.
6.         Use $NM$ for new thread-to-core assignment.
7.         Return $NM$.
8.     else
9.         Get variation percentage for entry $< CM, NM >$.
10.        if ($variation > \delta$)then
11.            Use $NM$ for new thread-to-core mapping.
12.            Return $NM$.
13.        else
14.            Reconstruct new cluster.
15.            $counter = counter - 1$
16.        endif
17.    endif
18. endwhile
```

Figure 4.3. New mapping construction by the cost and benefit analysis thread.

newly generated cluster, and maps these thread pairs to available cores by looking at the current mapping so that the new cluster resembles the current mapping as much as possible. This algorithm ensures that thread pairs with low data sharing are not migrated and thread pairs with high data sharing are mapped to the same core.

4.1.3. Counter Thread

The counter thread uses the thread-core mapping information generated by the cost and benefit analysis thread and cache statistics between the two mappings. The responsibility of this thread is to generate a mapping table by using the cache statistics between the two mappings. This mapping table has three rows, which are represented as previous mapping, current mapping, the variation in cache miss rate. This helper thread calculates the variation Δ between cache miss rates by using the following formula:

$$\Delta_{miss\ ratio} = 100 * \frac{MRp - MRc}{MRp} \tag{4.2}$$

where MRp and MRc are the (cache miss / total cache access) ratios for the previous and current mapping, respectively. This ratio is used when the decision of moving from one mapping to the other is made. Today's modern computer architectures contain various performance counters, but accessing them from the software (by using an OS command) is not permitted. The counter thread fulfills its function given above by using these performance counter values.

In order to access the cache statistics, the application should communicate with the Simics simulation environment. The application sets the o0 register on Serengeti architecture and ebx register on Intel x86 architecture, and calls a special NOP command, which is termed *magic instruction* by the simulator. As soon as the magic instruction is called, the application stops and the simulator is invoked by a hap callback. The simulator calculates the cache hits and misses of all caches defined and calculates overall miss ratio. The miss ratio is sent to the application by using either the o0 register or the ebx register, and the application continues its execution.

The inline assembly code called by the application for Serengeti architecture running Solaris operating system is given in Figure 4.4. On Intel x86 architecture, we used gcc inline assembly code as shown in Figure 4.5. Data passing to the Simics simulation environment by using the *ebx* register is shown on line 3. To stop the execution of the application and start cache statistics calculations, a special NOP command (*xchg*) given in line 4 is used. After Simics completes the cache statistics collection, it sends the miss ratio to the application through *ebx* register, and the miss ratio is accessed by the application using the *val* variable as in line 5.

```
1. Inline Function:
2. .inline magic_break,0
3. .volatile
4. sethi 0x40000, %g0
5. .nonvolatile
6. .end
```

Figure 4.4. Inline assembly code for Solaris running on serengeti architecture.

```
1. static inline unsigned int myMagic(unsigned int n)
2.       val = n
3.       asm volatile (”movl %0, %%ebx” : : ”g” (val) : ”ebx”)
4.       asm volatile (”xchg %bx,%bx”)
5.       asm volatile (”movl %%ebx, %0” : ”=g” (val) : )
6.       return (unsigned int) val
```

Figure 4.5. Gcc inline assembly code for Linux running on x86 Intel architecture.

4.1.4. Mapping Thread

The mapping thread uses the thread-core mapping generated by cost and benefit analysis thread and maps threads to cores. Before mapping threads to cores, it interrupts the counter thread to stop performance counters and restarts counter thread to collect cache statistics after the mapping of threads to cores is completed. For mapping threads to cores, we have used the *processor_bind* command on Solaris and *pthread_setaffinity_np* command on Linux.

4.2. Enhancements on Our Framework for Decreasing Overheads

The main idea behind the proposed framework is to increase performance of application threads by increasing data usage in the L1 and L2 cache. In order to achieve our goal, four helper threads that run parallel with application threads are introduced. Since four more threads that are much more complicated than the application threads are added to the system, the performance of the overall system decreases even though this framework is able to increase cache hit rate on both L1 and L2 caches. We have used three applications with numerous inputs and tested the performance of our framework with different core, L1 and L2 cache configurations. In this section, the overheads of our framework are given, and the solutions to decrease the overheads are proposed.

4.2.1. Overhead of Thread Migration

The most time-consuming overhead of our framework is the overhead of thread migration. In our initial implementation, we let the framework migrate threads even with small changes in data access patterns. Throughout execution, the order that threads will use CPU is known only at runtime, and as the threads' data access pattern is collected, some threads access more data than others. As a result, data sharing of threads varies throughout execution, and at some points of execution, we may take unnecessary mapping decisions. This increases the number of mappings done and thus increases the thread migration overheads. So the following constraints are added to the framework:

- We do not migrate threads unless it is necessary. If a number of mappings generated consecutively are the same, then it is beneficial to use that mapping. This helps to eliminate those mappings that are not really beneficial but are taken due to missing sharing information.
- The workload of cores changes throughout execution. Even though load is distributed equally among threads, the execution time of threads varies, so when one thread finishes its execution, other threads assigned to the same core can get more CPU cycles than threads running on other cores and this increases the probability of finishing their execution. So as soon as threads finish their execution, we migrate threads to balance load among cores.
- We perform a computational study to compare the performance of our framework and the Linux scheduler, and for a fair comparison, we have changed our initial mapping. We have proposed our initial mapping to be random and changed it to the exact mapping that Linux scheduler has performed.
- We have used different L2 sharing schemes among cores. Simics does not support simultaneous multithreading so each time a context switch occurs, the L1 cache is flushed; therefore only the data that resides in L2 cache is shared among threads. If each core has its private L2 cache, then this cache is used by only the threads assigned to that core. But if L2 cache is shared by a number of cores, then the data is also shared by all the threads assigned to those cores. So it is much more logical to construct L2 cache number of cluster of threads. For instance, if two cores share L2 caches, then it is much more beneficial to construct $coreNum/2$ clusters. This approach has two advantages. It presents effective usage of data by increasing data sharing among threads. It also decreases the overhead of thread migrations. Threads are only migrated between cores sharing the same L2 cache unless another core becomes idle.

4.2.2. Overhead of Helper Threads

In order to dynamically monitor data accesses of application threads, we have to use additional threads and this normally adds an overhead to the application. We have revised our helper threads and implemented them as simple as possible to decrease their overheads. We have also changed some features of our framework based on the characteristics of the applications used.

The clustering thread should work very fast to detect changes in data access patterns and to reflect these changes to the system. If this thread is not quick enough, then clusters generated can no longer be beneficial and we cannot achieve improvements in execution time. The clustering thread starts by calculating sharing among threads. Initially, we have used a data structure to store data accesses of threads and processed it to calculate the number of shared data accessed by thread pairs. To simplify this process, we have defined bit vectors for each thread. Each bit in the bit vectors represents data that can be accessed by more than one thread. Initially all bit vectors are filled with zeros. If data j is accessed by thread i, then bitVector(i,j) is set to 1. With the help of this representation, we have simplified computations by using simple bitwise operations. For instance, we have calculated the number of shared data among thread i and thread j by using AND operation on bitVector(i) and bitVector(j). By counting the number of 1's in the resulting vector, we obtained the total amount of shared data accessed by thread i and thread j. Similarly, the number of total data accessed by application threads is calculated by counting 1's in their bit vector.

The benchmark applications considered are working on loop operations and the data that application threads access in one iteration is repeated in all other iterations, so the data access pattern obtained in the first iteration is repeated throughout execution. So we have clustered threads according to the initial data sharing values, and this cluster does not change until some threads finish their execution. As some threads are completed, new clusters are constructed with the available application threads in the system for load balancing purposes. The cost and benefit analysis thread is the main decision mechanism and it changes the given mapping if necessary by resembling the given cluster with the current cluster. Since the most useful mapping is our initial mapping, we no longer need to resemble the given cluster, so we omit this part. Also the mappings are done to balance load except the initial mapping, so we do not need to collect cache statistics, so we can omit counter thread. Therefore, the responsibility of the cost and benefit analysis thread is to get the cluster information from the cluster thread and make the mapping thread migrate threads between cores; as a result it acts as a bridge between two helper threads. These two helper threads can communicate directly, so we can omit the cost and benefit analysis thread, too. Our enhanced framework contains two helper threads, the clustering thread and the mapping thread. Even with these modifications, we could not outperform the Linux scheduler. The details of the results are given in Section 4.4.4.

4.3. Setup and Experimental Evaluation

In this section, we first give a brief background on the multicore simulation platform Simics that is used in our experimental study. This is followed by the description of the applications used to test the performance of our framework.

4.3.1. Simulation Platform Simics

Simics [62] is a full system simulation platform from Virtutech [63], it supports the design, development, and testing of computer hardware and software. It achieves balance between accuracy and performance; it is accurate enough to run commercial workloads and is fast enough to run real workloads, interactive desktop applications and games. It can also model embedded systems, multiprocessor systems, clusters and networks.

Simics simulates processors at the instruction-set level, currently it supports models for UltraSparc, Alpha, x86, x86-64, PowerPC, IPF, MIPS and ARM. The hosts Simics supports are Linux, Windows and Solaris. Multiple instances of Simics can simulate different processor architectures running different operating systems on a single machine.

Simics is scalable, it can support single board, multi board, multi core, multi processor, heterogeneous, distributed, networked, system-on-chip systems, and it uses device modeling language (DML), Python and C/C++ as the modeling languages for various architectures. Simics models targets with objects. Processors, devices or virtual objects like disk images and memory mappings, are all modeled by using objects.

There are several debug tools in Simics, breakpoints can be added on a device access or read from memory components to monitor the system. The simulation can be run in reverse order, so that we can go over errors and see the problems occurred during the simulation.

Virtutech has a tool called SimGen that generates an interpreter for the instruction set of the target processor. To do this, first the high level description of the instruction set should be written. Patterns for instruction decoding are given to SimGen; it takes the description and generates the interpreter. SimGen can be used if a target processor which has not been implemented in Simics, should be used.

Simics is used as our simulation platform, since our purpose is to design and implement multicore systems with different attributes. These attributes are varying core types and numbers, varying on-chip memory components and variable sharing of these memory components among cores.

4.3.2. Applications

For testing the performance of the thread-to-core mapping algorithm, the following applications are used.

- Black-Scholes application is from PARSEC Benchmark Suite [64]. We used OpenMP version of this algorithm. This application [65] uses Black-Scholes partial differential equation to calculate the prices for a portfolio of European options. The program is parallelized by dividing the number of portfolios into a number of work units and assigning work units to application threads dynamically. Each thread iteratively calculates the price of the given portfolio for a predefined time interval.
- Jacobian Matrix Calculation performs addition operations on the elements of a given matrix A and stores the result in another matrix B. For the calculation of each data element of matrix B, row-wise and column-wise neighbors data elements are accessed in matrix A.
- Sparse matrix-vector multiplication (SpMV) is an important kernel in scientific computing which has irregular data access patterns due to matrix sparsity. SpMV works on a sparse matrix A and a vector X and calculates $AX=B$.

4.4. Results and Discussion

For performing our experiments, a multiprocessor simulation environment, Simics toolset is used. For Serengeti Architecture, each configuration in the experiments runs Solaris 10 operating system and Sun Studio [66] compilers are installed for OpenMP support. For Intel x86 Architecture, experiments are conducted on Linux operating system using gcc compiler. We have first started to conduct our experiments on Serengeti Clusters running Solaris operating system. After threads are migrated, the cache read and write accesses are inconsistent with non-helper version and the values we have obtained were 20% - 30% higher than they should be. This increase in the number of cache accesses is due to the OS disturbance. So we have switched to Linux operating system for much reliable results on pthreads implementations. The simulation parameters for the CMP architectures used is given in Table 4.1.

In order to represent the assignment of helper threads to cores, <x, y, z> representation is used, where x shows the number of cores used only for application threads,

Table 4.1. Simulation parameters for performing experiments.

$Parameter$	$Value$
$Numberofcores(n)$	8, 16
$Numberofthreads(p)$	16, 32, 64, 128, 256
$CPUFrequency$	1 GHz
$L1Cache$	4K, 8K, 16K, 64K, 2-way
$L2Cache$	64K, 128K, 256K, 4-way

y shows the number of cores that are used by both helper and application threads and z represents the number of cores that are only used by the helper threads.

4.4.1. Scalability Results

In this part of the experiments, the speedup curves of the applications are given. These curves are very important for our framework, since we are working with multicores, the applications may not take full advantage of all the cores in the system. For instance, using n cores and $n + 1$ cores may have the same performance, so using this additional core would be inessential for the application. Instead of reserving this additional core/cores to application threads, these cores can be used by the helper threads. Allocating these cores to helper threads would not reflect application threads' performance and the overhead of the helper threads would no longer effect the overall system. Additionally some cores would be reserved to helper threads and these threads would not share caches with application threads, thus they would not invalidate the shared data among application threads.

In order to achieve these goals, we need speedup curves that are not increasing linearly, instead after a certain core number, these curves should bend and should end up with minor speedup values. But unfortunately, the desired scalability results can not be obtained, instead all applications' speedup curves continue to increase linearly as more cores are added to the system.

The speedup of Black-Scholes application and Jacobian Matrix Multiplication for 16, 32 and 64 threads running on 1 to 16 cores are given in Figure 4.6 and Figure 4.7 respectively. Both applications are data parallel and they obtain full advantage of all resources in the system.

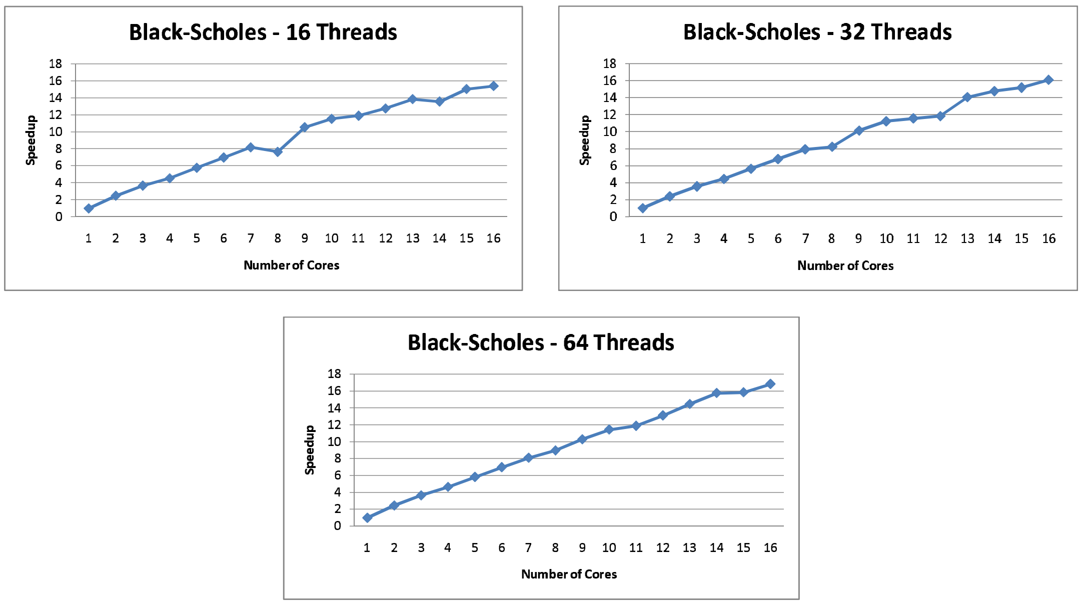


Figure 4.6. Scalibility of Black-Scholes application with 16, 32 and 64 threads.

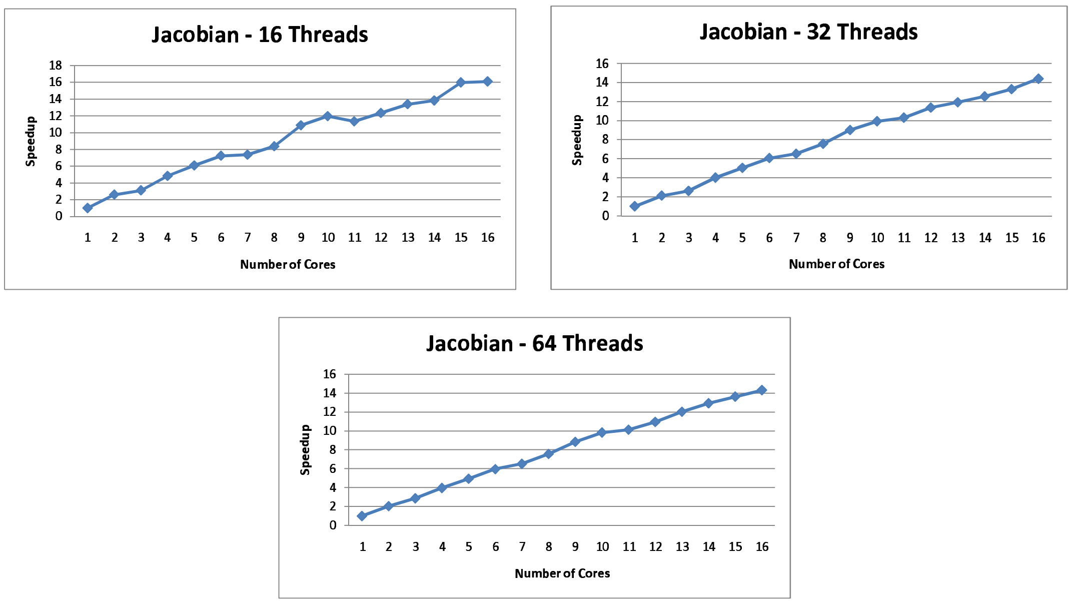


Figure 4.7. Scalibility of Jacobian application with 16, 32 and 64 threads.

We conducted the same experiments on SpMV with three different inputs namely, Boddy4, Boddy5 and Boddy6; and obtained the same results. So we extended the tests by using 64, 128 and 256 application threads running on 1 to 32 cores and obtained the results given in Figure 4.8. This application also uses all available resources in the system. One interesting behavior of these scalability tests is to obtain a speedup of 32.88, 33.53 and 32.92 for Boddy4, Boddy5 and Boddy6 inputs respectively, for 32 cores when 256 threads are in use. Normally, we can achieve a maximum speedup of 32 but the additional resources such as L1 and L2 caches increases the speedup values. The scalability tests showed that reserving cores to helper threads and decreasing

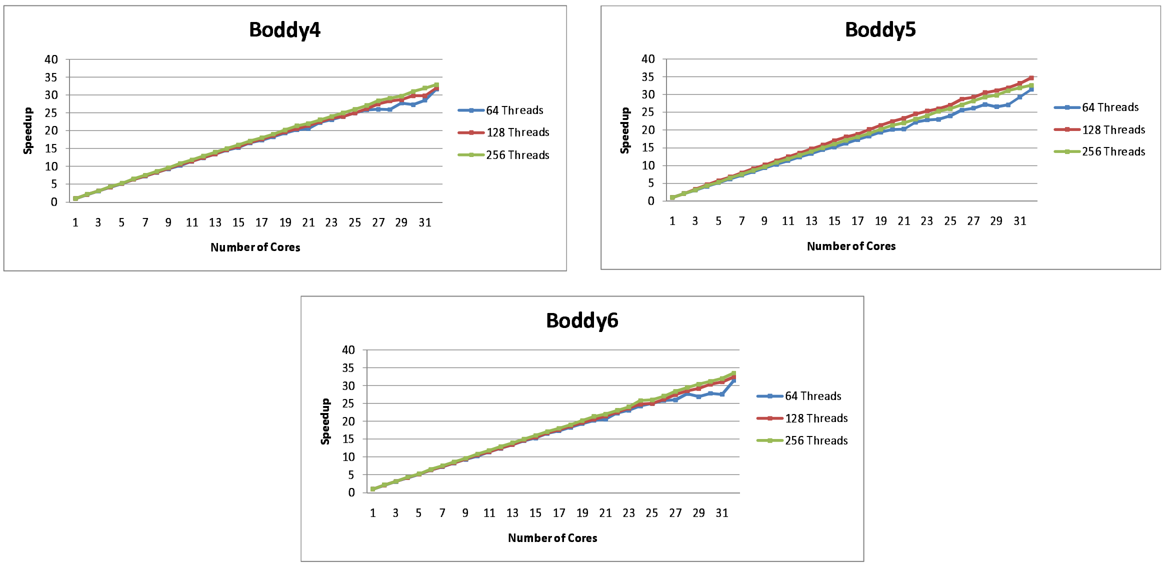


Figure 4.8. Scalibility of SpMV application with 4 threads per core.

the number of cores application threads use, is not beneficial and would yield to performance degradations. So we should run both application threads and helper threads on the same cores.

4.4.2. Quantifying Overheads

In this part of the experimental study, the overhead of helper threads is measured by running them parallel with application threads. The cluster, the cost and benefit analysis and the counter thread's code remain as they are, whereas the mapping thread makes the decision of mapping application threads to cores but does not actually map threads to cores. For all applications, the overhead values we obtained are similar. The helper threads' overhead increase when they are mapped to only one core because the cache usage of the helper threads increases and they steal more CPU cycles from application threads, and this increases the execution time of application threads running on the same core with helper threads. Additionally when each helper thread is mapped to a different core, the helper thread load is distributed among cores and as a result, the overhead of the framework decreases. The overhead values for the SpMV application running with three inputs is given in Figure 4.9. For all inputs considered, the helper threads adds an overhead of 5% to 15% to overall execution time.

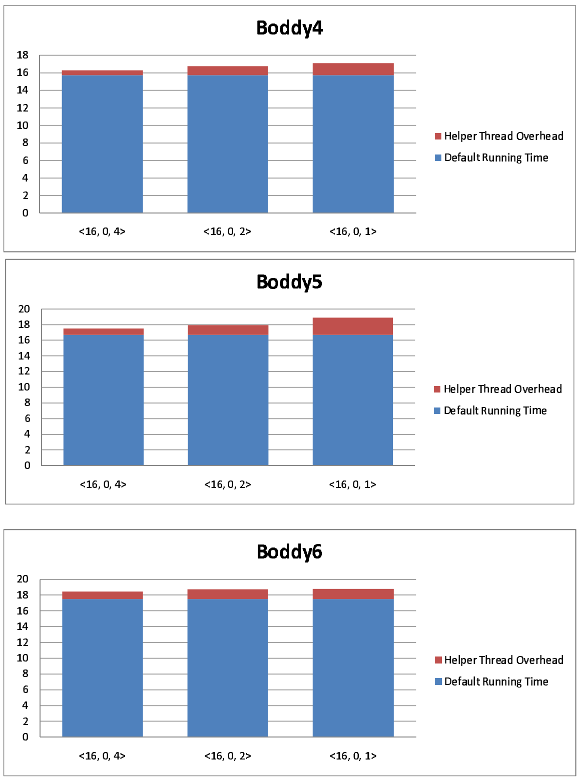


Figure 4.9. Overhead of running helper threads with 64 application threads on 16 cores for SpMV application.

4.4.3. Performance of Helper Threads

The performance of helper threads is measured either by assigning them to a core with an application thread or by assigning them to a dedicated core. When 8 cores are considered, helper threads are either assigned to 1 individual core and 7 remaining cores are used for running application threads or helper threads and application threads run parallel on one core and 7 cores are reserved for application threads. The comparison of running times for Black-Scholes application is given in Figure 4.10.

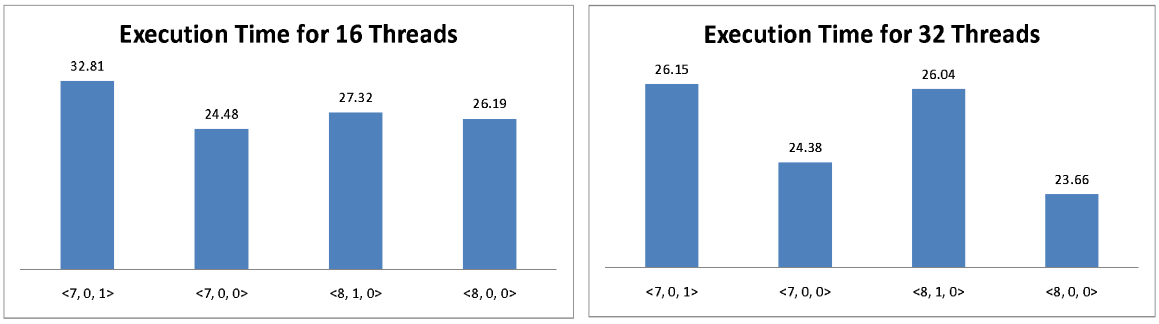


Figure 4.10. Performance of helper threads running with application threads on 8 Cores for Black-Scholes application.

When 16 cores are considered, 1 to 4 cores are assigned to helper threads, and remaining cores are used to run application threads. The comparison of execution times is given in Figure 4.11. We achieved an improvement of 15% when 12 cores and an improvement of 1% when 13 cores are considered for running 64 application threads. When 32 application threads are considered, we achieved a speedup of 19% with 12 cores running application threads. But when application threads use all available resources, we are not able to obtain improvement in execution time.

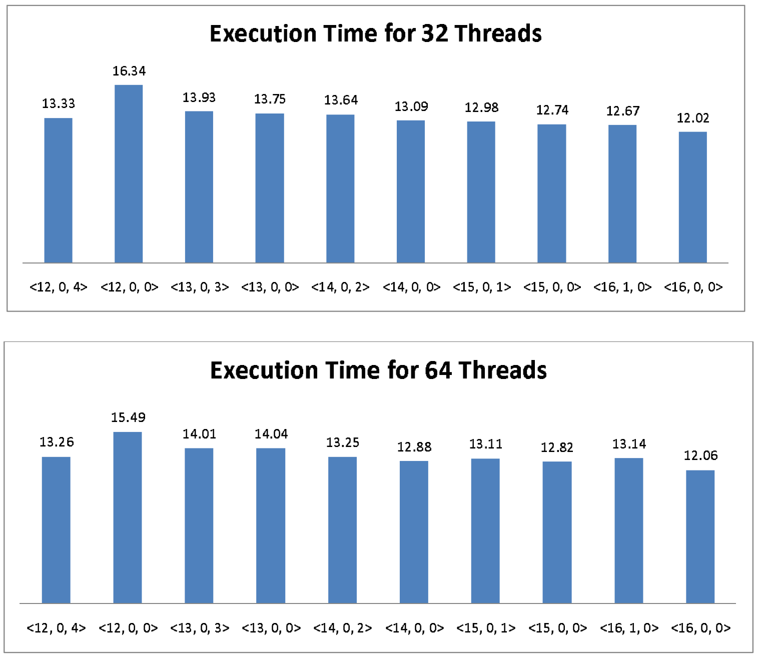


Figure 4.11. Performance of helper threads running with application threads on 16 cores for Black-Scholes application.

The same performance tests are repeated for Jacobian Matrix Multiplication on 8 cores with 32 threads and 16 cores with 64 threads given in Figure 4.12, and SpMV application on 16 cores with 64 threads given in Figure 4.13. When individual cores are reserved for helper threads, they do not disturb the shared data used by application threads on caches, so the overall system performance increases. Even with this performance improvement, we are not able to outperform the application-only version which is using all available cores. The execution times of the enhanced framework using two helper threads running parallel with 32 and 64 application threads for the SpMV application is given in Figure 4.14. The results are similar with our initial framework due to the overhead of the thread migration.

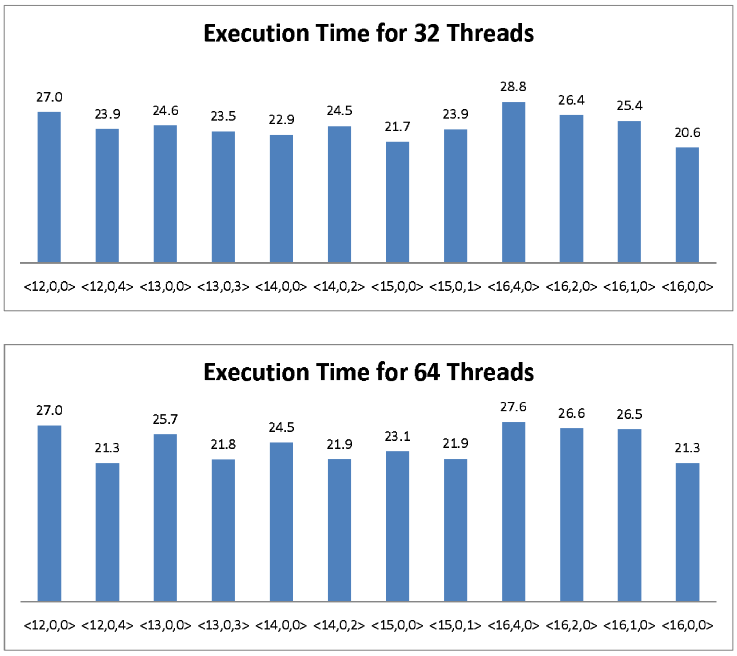


Figure 4.12. Performance of helper threads running with application threads for Jacobian Matrix Multiplication.

4.4.4. Sensitivity Analysis

This section gives detailed analysis on how the size of on-chip memory components affect performance. Then the performance of the best mapping is presented by statically using it as the initial mapping. The overhead of thread migration is also explained in detail. Finally to clarify the importance of load balancing, execution time diversity of application threads and the idleness of cores are measured.

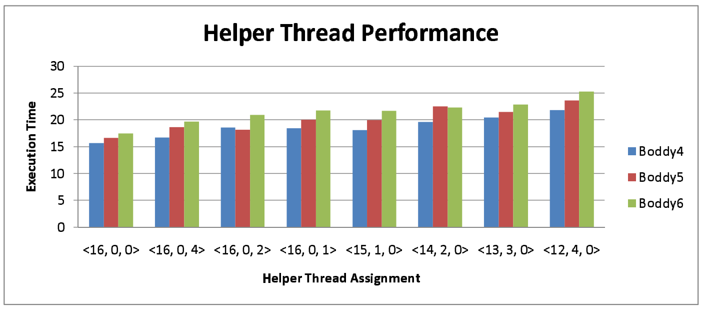


Figure 4.13. Performance of helper threads running with application threads for SpMV.

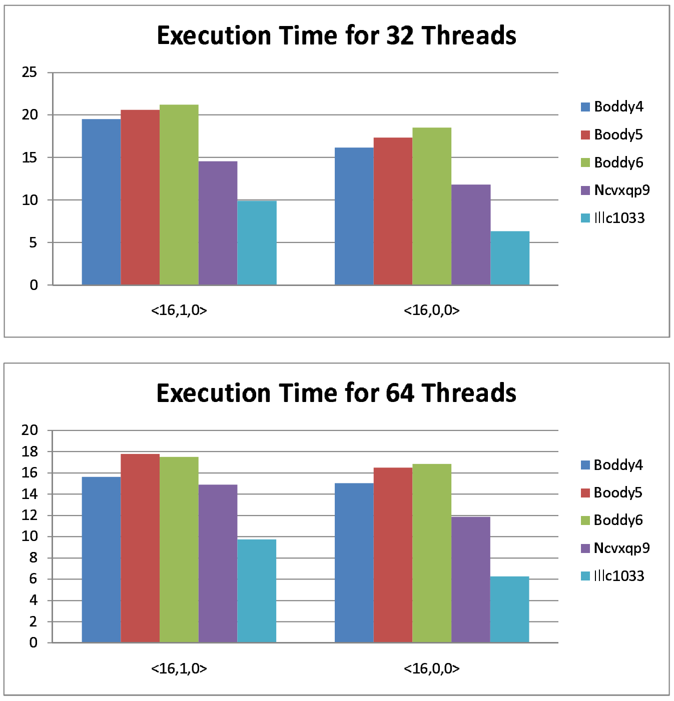


Figure 4.14. Performance of enhanced helper threads running with application threads for SpMV.

In the first part, the performance of the helper threads is measured by varying the L1 cache and L2 cache sizes. The effect of L1 cache size is measured by using Black-Scholes application and the results are shown in Figure 4.15. When L1 cache size is decreased to 32K, the performance is decreased by 3% when 8 cores and 16 application threads are considered. When 16 cores are used, the performance is decreased by 5% for 32 application threads and 3% for 64 application threads.

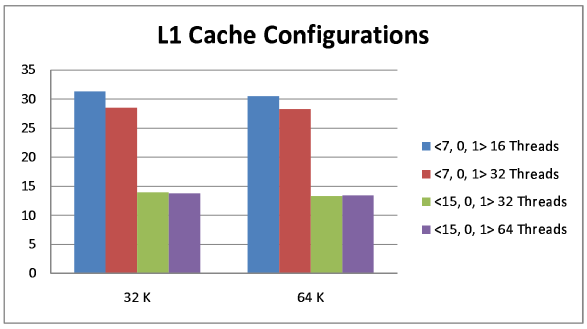


Figure 4.15. Performance of helper threads for different L1 cache configurations for Black-Scholes application.

We have measured the effect of both L1 cache and L2 cache size on the performance of SpMV application. In these tests, for all input matrices, the application threads are mapped statically to cores by using the best thread mapping scheme that maximize data sharing. In this test, helper threads are not included to see the pure benefit that can be obtained by using the best thread-to-core mapping. The migration overhead of application threads is also neglected. The results show that we gain a performance improvement of at most 7% when the cache sizes are 4K for L1 cache and 256K for L2 cache for Boddy4 input. As L2 cache size is increased, the number of data elements in the cache increases and this increases the possibility of a thread reusing the data fetched by another thread, and this increases the performance of the static mapping. When the overhead of thread migrations for the same static mapping is included, for 64 and 128 threads, the performance gain is equal to the overhead of thread migrations and as a result, we no longer can obtain performance improvements. When the number of threads in use is increased to 256, the overhead of thread migrations also increases, which degrades performance of the static mapping.

Migration overhead for only the initial mapping is equal to or greater than the performance gain obtained by data sharing, and when the helper thread overhead is added to the system, the total overheads are much greater than the performance improvements. We tried to find additional methods to improve these performance gains and investigated the performance gains that can be obtained if load balancing is added to the system. Two dynamic tests are used to see the effect of load balancing in detail. The first test includes the load distribution of cores for 64, 128 and 256 threads for the SpMV application as shown in Figure 4.16.

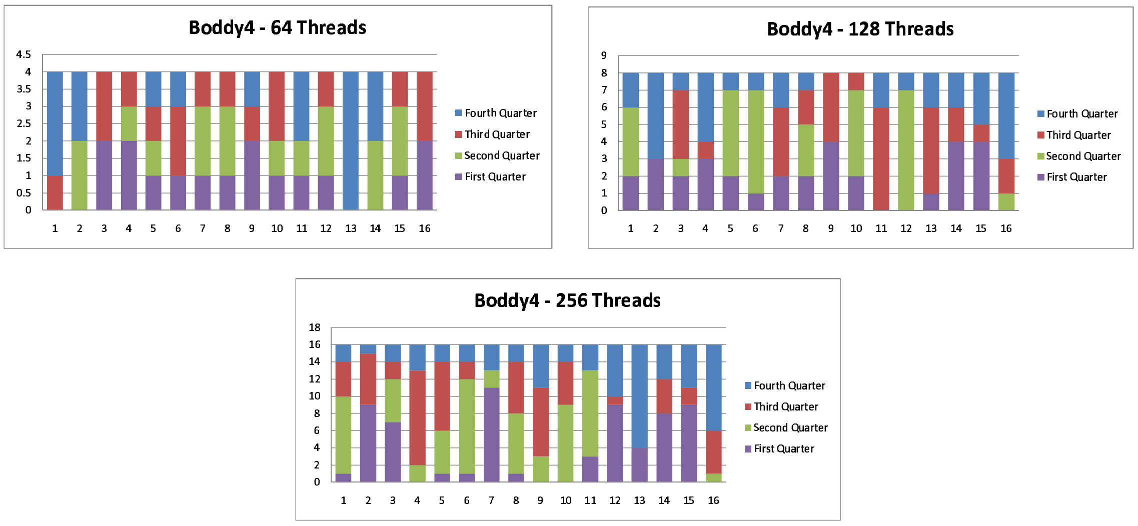
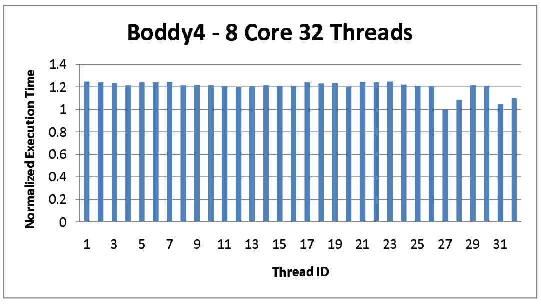


Figure 4.16. Load distribution among cores for 64, 128, 256 application threads on 16 cores for SpMV application.

The second test explores the differentiation among execution times of threads for 8 cores and 16 cores using 32 and 64 application threads. Figure 4.17 shows the normalized execution time of threads running on 8 cores. The normalized execution times of application threads on 16 cores is given in Figure 4.18

Finally, a case study on Boddy4 input running with 32 application threads on 8 cores is given in Figure 4.19. This study includes eight dynamic mappings. The first mapping taken at the beginning of the program is the initial mapping which is



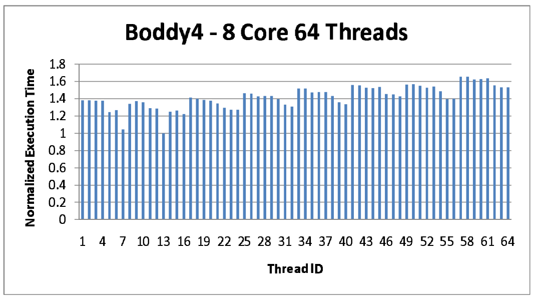
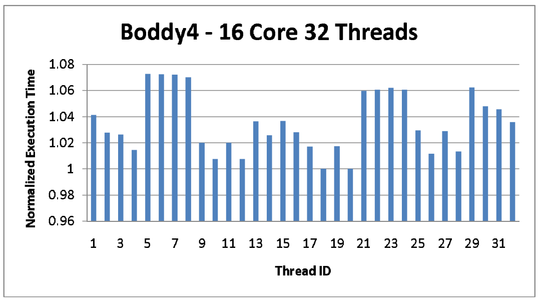


Figure 4.17. Normalized execution time of application threads on 8 cores for SpMV.



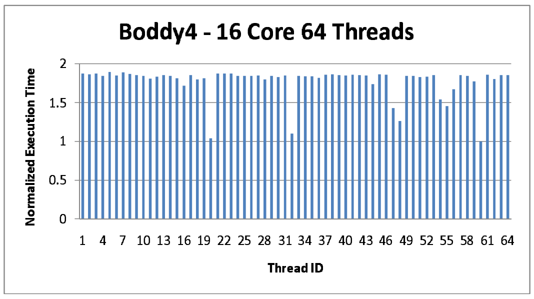


Figure 4.18. Normalized execution time of application threads on 16 cores for SpMV.

identical to that of Linux scheduler. After the initial mapping, the thread-to-core mapping based on the best data sharing pattern is performed. In this step, 23 out of 32 threads are migrated. As the first thread finishes its execution in the 19th second, the framework starts to balance load. The following mappings include one or two migrations, and they all aim to balance load.

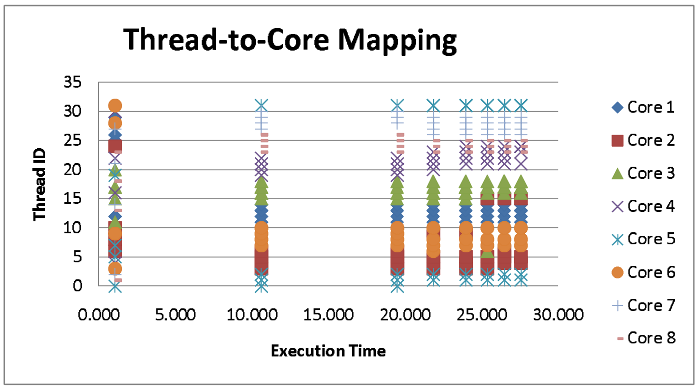


Figure 4.19. Dynamic distribution of 32 application threads to 8 cores for SpMV.

In spite of optimizations on data sharing and load balancing, due to the overheads of thread migration and helper threads, we are not able to gain performance improvements on this study. We have changed architectural resources and their connections, benchmark applications and their inputs but in no case, we are able to outperform the Linux scheduler.

5. A CACHE-CENTRIC DATA ASSIGNMENT METHODOLOGY ON GENERIC MULTICORE ARCHITECTURES

In this part of the thesis, we propose a locality-aware mapping algorithm which uses a cache-centric data mapping methodology for data-to-thread assignment. To demonstrate the performance of our locality-aware mapping algorithm, sparse matrix-vector multiplication is used. SpMV effects the performance of numerous applications in economic modeling, scientific and engineering computing. Sparse kernels suffer from irregular and indirect memory access patterns. The properties of the sparse matrices can only be known at runtime so code transformations as well as dynamic optimizations and tuning are needed for higher performance. Some optimization techniques including column-reordering, row-reordering and pipelining, are proposed to increase the performance of the algorithm [67–70]. There are many studies on optimization and tuning on SpMV in the literature [71, 72]. The performance of SpMV is also studied on multicore architectures [73] and graphics processing units (GPUs) [74, 75]. In Strout et al.'s study [76] rescheduling of computation for increasing data locality in sparse matrices is explored and Gauss-Seidel is used for testing the performance of their algorithm which is called serial sparse tiling.

The management of the shared on-chip space is a critical issue on CMPs. A data element on the cache may be replaced by a data access of one core. When the displaced data element is requested again, a cache miss occurs. The overall performance is significantly affected by the conflicts on the shared cache. In order to obtain high performance from CMPs, the reuse of data elements on the cache should be increased. Shared cache conflicts can occur by threads of different applications, and these conflicts are more frequent when threads of the same application are mapped to different cores [77].

The same data element may be accessed through different parts of the program more than once. In the ideal case, the data element is brought to the shared cache space in its initial access, and during all other accesses, the data element resides in the cache and is reused. This ideal case may not be achieved due to the limited capacity of the shared cache space and the access patterns of different threads. Our goal is to reduce the number of conflict misses on the shared cache by dynamically considering the runtime behavior of threads and their data usage.

In this work, a locality-aware mapping policy that observes the runtime characteristics of an application is proposed. The application (i.e., the input of the given application) is partitioned into chunks based on computations with similar access patterns and the chunks with similar data access patterns are grouped into bins. Similarity calculations among chunks consider both the shared and distinct data. This policy focuses on effective usage of caches by assigning data that will be reused to the same cores. In this method, we make the following contributions:

- The locality-aware dynamic mapping algorithm proposed in this study considers the data access patterns of chunks and determines both the shared and distinct data used by different chunks in order to calculate the similarities among them. Using only shared data for similarity calculation would neglect the amount of distinct data, which would decrease the data reuse probability on the cache. The locality-aware mapping algorithm can be applied to any multithreaded application and would yield high performance improvements, especially for those with irregular data access patterns.
- In order to validate the effectiveness of our proposed algorithm, we consider the sparse matrix-vector multiplication by using the classical blocked version of the compressed sparse row (CSR) format. SpMV is a critical application that effects the performance of numerous applications in different domains, including economy, science, and engineering. Unlike other scientific kernels, such as dense linear algebra kernel, SpMV requires many indirect and irregular memory accesses; therefore efficient data distribution is crucial for SpMV [73]. The

properties of the sparse matrices can only be known at runtime, so code transformations as well as dynamic optimizations and tuning are needed for higher performance.

- We perform a simulation study by considering various inputs with different shapes and characteristics in order to evaluate our proposed locality-aware mapping algorithm. Our algorithm increases data reuse on the shared cache for all cases, and our proposed mapping strategy outperforms the Linux scheduler for different distributions of nonzero data elements.
- Our simulation environment contains CMPs with 16 cores, each of which have a private L1 and private L2 cache. The input matrices used by SpMV differ in nonzero element distribution, as well as in size and in the number of nonzero elements. The size of the matrices takes up to 1.5 GB, and the maximum number of nonzero elements used in the matrices is 77057.

5.1. Proposed Algorithm for Locality-aware Dynamic Mapping

Data reuse for multithreaded applications is a critical issue, as it increases the probability of requested data elements to be located in the cache, which improves performance by decreasing the number of data misses. Especially if the application's data distribution is obtained in runtime, data-to-thread distribution should be done dynamically. Therefore, locality-aware dynamic mapping can be considered to increase data reuse on the shared cache. Data-intensive, multithreaded applications operating on any type of data could be the target application domain for locality-aware dynamic mapping.

The main idea underlying this algorithm is to assign computations with similar data access patterns to same cores. An application program may access different data and access pattern may vary throughout execution. In the ideal case, a data element is accessed consecutively; therefore, it can be found on the shared cache. On the other hand, data element accesses are unordered in the worst case. In most of the applications, we do not observe the ideal case; therefore, the probability of data element reuse on the shared cache is low.

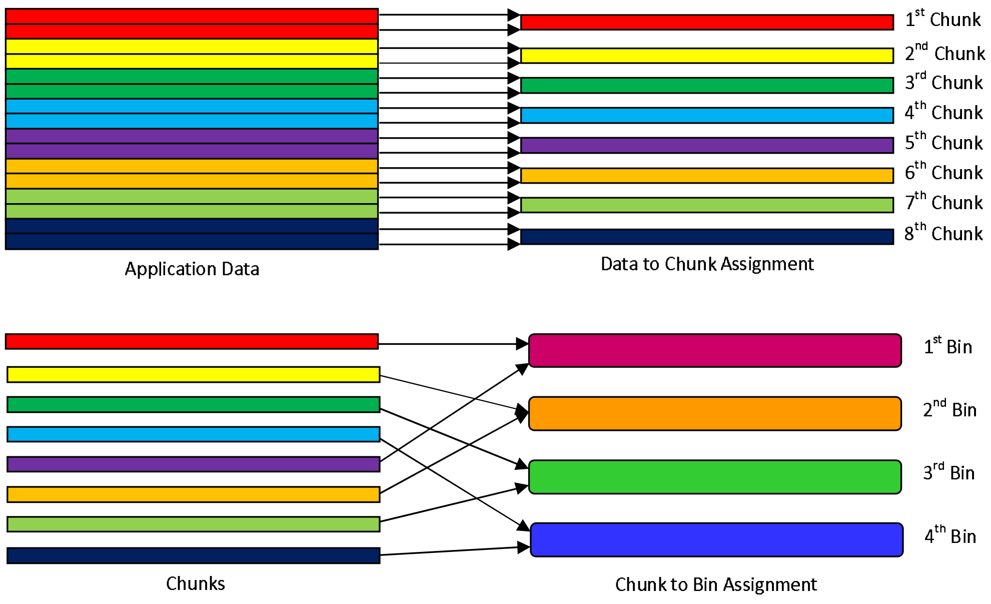


Figure 5.1. The layout of our framework including chunk and bin representations.

Many applications obtain their program data at runtime. In order to find the similarities between data elements, a dynamic strategy should be considered. As soon as the application acquires its input data, our approach begins with the similarity calculations and forms groups of data elements that have the maximum amount of sharing. These groups are then assigned to application threads, and these threads start their execution. Thus our proposed algorithm starts execution after the input data is read, and then it divides the program into computations with similar data access patterns, called *chunks*. By using the data elements accessed in each chunk, the locality-aware mapping algorithm groups chunks using similar data elements together to *bins*, and it finishes its execution. Chunk and bin representations in our framework are given in Figure 5.1.

Figure 5.2 shows how similarity calculations are performed in our algorithm. While calculating similarities in our scheme, we try to find two chunks with the maximum number of shared data and minimum number of distinct data. Each chunk is represented with a bit vector to store the data that is accessed by the computation it is

Input: Number of bins ($binNumber$), Number of chunks per bin($chunkPerBin$), Total data elements to be used by all threads (TDE)

Output: Pairwise similarity values of chunks

```
1. $totalChunks = binNumber * chunkPerBin$
2. **for** i=0 to totalChunks **do**
3.    **for** j=0 to TDE **do**
4.      **if** $TDE_j$ is accessed by $Chunk_i$ **then** $bitVector_{i,j} = 1$.
5.    **end for**
6. **end for**
7. **for** i=0 to totalChunks **do**
8.    **for** j=i+1 to totalChunks **do**
9.      $shared = bitVector_i\ AND\ bitVector_j$
10.     $distinct = bitVector_i\ XOR\ bitVector_j$
11.     mask=1
12.     **while** $shared > 1$ **do**
13.       $totalShared_{i,j} = totalShared_{i,j} + shared\ AND\ mask$
14.       $shared = shared >> 1$
15.     **end while**
16.     mask=1
17.     **while** $distinct > 1$ **do**
18.       $totalDistinct_{i,j} = totalDistinct_{i,j} + distinct\ AND\ mask$
19.       $distinct = distinct >> 1$
20.     **end while**
21.   **end for**
22. **end for**
23. **for** i=0 to totalChunks **do**
24.   **for** j=i+1 to totalChunks **do**
25.     $similarity_{i,j} = totalShared_{i,j}/(totalDistinct_{i,j}+ totalShared_{i,j})$
26.   **end for**
27. **end for**
```

Figure 5.2. The algorithm for calculating similarities among chunks.

representing as given in Figure 5.2, lines 2-6. The bit vector of each chunk, considering data accesses, is defined as follows:

$$chunk(i,j) = \begin{cases} 1, & \text{if } data(j) \text{ is accessed by } chunk_i \\ 0, & \text{otherwise.} \end{cases}$$

If two bit vectors are the same, this shows that data accesses of these two chunks are the same; if the resulting vector obtained by using AND operation on two bit vectors is 0, two chunks share no common data at all. In general, to maximize cache hit ratio and improve performance, we should group chunks that can reuse the highest amount of data.

The similarity between two chunks is computed by performing AND and XOR operations on two-bit vectors as shown in Figure 5.2, lines 7-22. AND operation on two bit vectors calculates the total number of shared data between two chunks; whereas XOR operation calculates the total number of distinct data between two chunks. The similarity between two chunks is calculated by the following formula:

$$similarity(i,j) = \frac{totalShared(i,j)}{totalDistinct(i,j) + totalShared(i,j)}, \tag{5.1}$$

where $totalShared_{i,j}$ represents the total number of shared data points, and $totalDistinct_{i,j}$ is the total number of distinct data accessed by both chunks. It is obvious that two chunks with high data sharing and low data difference have higher similarity, so there is a high data reuse between them. We want to show that simply using data sharing or data differences would not capture the most similar chunk pair. If only the number of shared data points is used as the similarity metric, we would be neglecting the data differences, which could be high enough to destroy the sharing pattern. The number of data points used by different chunks can vary; therefore, the ratio of data sharing and total accessed data would be more appropriate to capture both of the constraints.

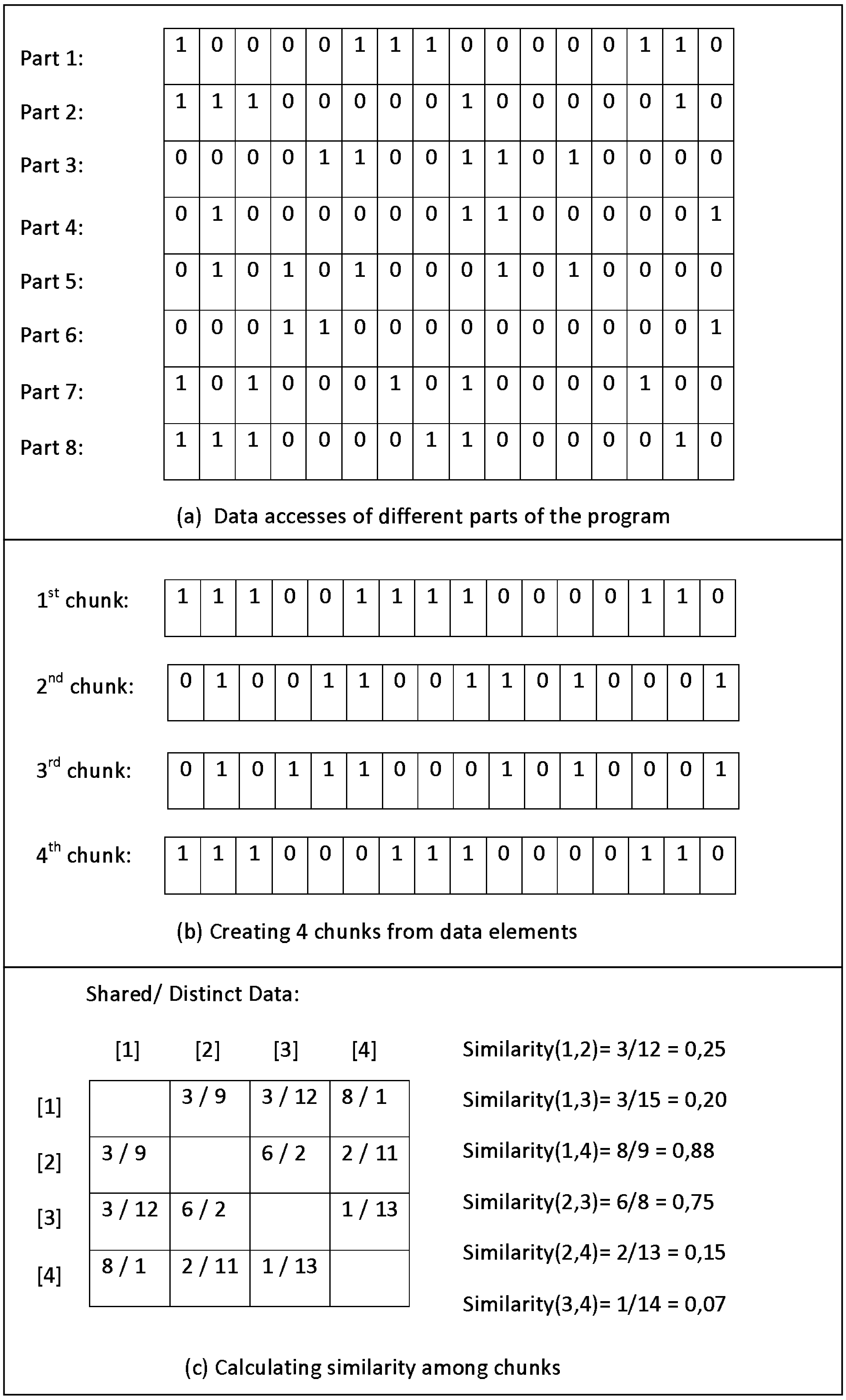
Then, similarities between bit vectors are sorted, and chunks with highest similarity values are grouped as shown in Figure 5.3. These groups are represented with *bins*. The number of chunks in each bin may vary according to the size of the data elements used in the application. Each bin contains equal number of chunks for a fair distribution of data elements. Our aim is to maximize data reuse on cache, so we group all chunks with high similarity into one bin. When assigning chunks to bins, we take into account both pairwise similarity between chunks and the similarity obtained from the chunks that are already assigned to bins to find the best bin for the given chunk pair.

In general, CMPs may have different cache structures, i.e. each core can have its private L1 and private L2 cache, or cores may have their private L1 cache and share L2 cache. Since we aim to increase data reuse on caches for multithreaded applications, the number of *bins* used is directly related to the number of caches used. Assuming the application has N shared caches, the locality-aware mapping algorithm should construct at least N bins. As a result, each bin should be assigned to threads working on cores accessing the same shared cache space.

To explain similarity calculation between chunks and our chunk assignment policy, an example application running 2 threads on 2 cores accessing different shared caches is given in Figure 5.4. In this example, we consider 2 bins. Assuming that the application consists of 8 computations, (a) in Figure 5.4 shows the data elements accessed in each computation. Consider that 4 chunks are used, 2 contiguous application parts are assigned to a chunk, and the bit vectors of the chunks will be formed as shown in (b) in Figure 5.4. If we take a closer look at the first chunk, all application data accessed in the first and second parts of the program should be included in this chunk, so we can obtain the bit vector of the first chunk by using the OR operation on the data accessed in the first and second parts of the application.

```
Input: Similarity values of ($chunks$),
       Number of chunks($totalChunks$)
Output: Chunk assignments to bins
1. Sort $similarity$
2. $chunksAssigned = 0$
3. while $chunksAssigned < totalChunks$ do
4.   Get next $similarity_{x,y}$
5.   if $chunksBin_x$ and $chunksBin_y$ not assigned then
6.     Assign $chunksBin_x$ and $chunksBin_y$ to $availableBin$
7.     $chunksAssigned += 2$
8.   else if $chunksBin_x$ is not assigned to a bin then
9.     Assign $chunksBin_x$ to $chunksBin_y$
10.    $chunksAssigned += 1$
11.  else if $chunksBin_y$ is not assigned to a chunk then
12.    Assign $chunksBin_y$ to $chunksBin_x$
13.    $chunksAssigned += 1$
14.  end if
15. end while
```

Figure 5.3. Chunk-to-bin assignment.



Part 1:	1	0	0	0	0	1	1	1	0	0	0	0	0	1	1	0
Part 2:	1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0
Part 3:	0	0	0	0	1	1	0	0	1	1	0	1	0	0	0	0
Part 4:	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1
Part 5:	0	1	0	1	0	1	0	0	0	1	0	1	0	0	0	0
Part 6:	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1
Part 7:	1	0	1	0	0	0	1	0	1	0	0	0	0	1	0	0
Part 8:	1	1	1	0	0	0	0	1	1	0	0	0	0	0	1	0

(a) Data accesses of different parts of the program

1st chunk:	1	1	1	0	0	1	1	1	1	0	0	0	0	1	1	0
2nd chunk:	0	1	0	0	1	1	0	0	1	1	0	1	0	0	0	1
3rd chunk:	0	1	0	1	1	1	0	0	0	1	0	1	0	0	0	1
4th chunk:	1	1	1	0	0	0	1	1	1	0	0	0	0	1	1	0

(b) Creating 4 chunks from data elements

Shared/ Distinct Data:

	[1]	[2]	[3]	[4]
[1]		3 / 9	3 / 12	8 / 1
[2]	3 / 9		6 / 2	2 / 11
[3]	3 / 12	6 / 2		1 / 13
[4]	8 / 1	2 / 11	1 / 13	

Similarity(1,2)= 3/12 = 0,25

Similarity(1,3)= 3/15 = 0,20

Similarity(1,4)= 8/9 = 0,88

Similarity(2,3)= 6/8 = 0,75

Similarity(2,4)= 2/13 = 0,15

Similarity(3,4)= 1/14 = 0,07

(c) Calculating similarity among chunks

Figure 5.4. Calculating similarity values for the case of 4 chunks with 2 bins.

After generating all the other bit vectors in the same way, we calculate similarities between chunk pairs and assign the most similar chunks to the same bin. (c) in Figure 5.4 shows the similarity calculations between chunk pairs by using shared and distinct data between chunks. In order the calculate the number of shared and distinct data points, AND and XOR operations are used, respectively. When the similarities between chunks are considered, chunk 1 and chunk 4 have the highest similarity value of 0.88 with 8 shared data points and 1 distinct data point, so chunk 1 and chunk 4 are assigned to the same bin. As for chunk 2 and chunk 3, similarity value of 0.75 is obtained with 6 shared and 2 distinct data points, so these chunks are assigned to the second bin. If the application uses 4 threads running on different cores accessing distinct shared caches on the target architecture, the number of bins used should be at least 4, and the number of chunks used should be at least 8 in order to have pairwise chunk groups. In this case, each computation will be considered as a chunk, and will be represented with a bit vector.

5.2. Setup and Experimental Evaluation

For performing our experiments, multicore simulation platform Simics is used. Each configuration in the experiments runs Fedora 5 operating system. We modeled our target hardware with 16 cores, and each core has a private L1 cache and a private L2 cache. Also the number of application threads used is equal to the number of cores in the target architecture, and each application thread is assigned to a single core. Our main configuration parameters and their default values are listed in Table 5.1. After presenting the primary results, we change the L1 cache and L2 cache sizes for sensitivity analysis.

To validate the applicability of our locality-aware dynamic mapping algorithm, we consider the sparse matrix-vector multiplication (whose code is given in Figure 5.5) as our benchmark. SpMV is an important kernel in scientific computing which has irregular data access patterns due to matrix sparsity. The data that SpMV application will use is not known at compile time, so to obtain a sharing-aware data distribution to

Table 5.1. Simulated architecture parameters of experimental study.

Parameter	*Value*
Number of cores (n)	16
Number of threads (p)	16
System	*PCI* Based X86 System
Processor Type	*Pentium* 4
CPU Frequency	3.5 GiB
Main Memory	1 GB
Private L1 Instruction Cache	8K, 2-way
Private L1 Data Cache	8K, 2-way
Private L2 Cache	32K, 4-way

threads, data should either be preprocessed or dynamically mapped to threads. So this application is suitable for measuring the performance of our algorithm. SpMV works on a sparse matrix A and a vector X and calculates $AX{=}B$. SpMV loads and stores the values of sparse matrix A and B only once; however, vector X is accessed indirectly many times. Our algorithm uses the access pattern of vector X and calculates the number of shared elements on this vector.

Five selected inputs of SpMV application from the University of Florida Sparse Matrix Collection [78] are considered for computational experiments. The properties and the shapes of those inputs are given in Table 5.2 and Figure 5.6, respectively.

5.3. Results and Discussion

In this section, the performance of our dynamic locality-aware mapping is evaluated and compared with standard Linux scheduler for various test cases considered.

```
Input: Bins assigned to each thread ($myBin$),
Chunk indexes of bins($globalIndex$),
First data element of each chunk ($startChunk$),
Last data element of each chunk ($endChunk$)
1. chunkIndex=$globalIndex_{myBin}$
2. $globalIndex_{myBin}$++
3. while chunkIndex $< maxChunk_{myBin}$ do
4.   startPoint = $startChunk_{chunkIndex}$
5.   endPoint = $endChunk_{chunkIndex}$
6.   while startPoint < endPoint do
7.       for j = $start_{startPoint}$ to $start_{startPoint+1}$ do
8.           total += $val_j * x_{cols_j}$
9.       end for
10.      $b_{startPoint}$ = total
11.      total = 0
12.      startPoint++
13.  end while
14.  chunkIndex=$globalIndex_{myBin}$
15.  $globalIndex_{myBin}$++
16. end while
```

Figure 5.5. Sparse matrix-vector multiplication algorithm with CSR storage.

Table 5.2. Input matrices of SpMV application.

Input Name	Number of rows	Number of columns	Number of nonzeros
Boddy4	17546	17546	69742
Boddy5	18589	18589	73935
Boddy6	19366	19366	77057
Illc1033	1033	320	4732
Ncvxqp9	16554	16554	31547

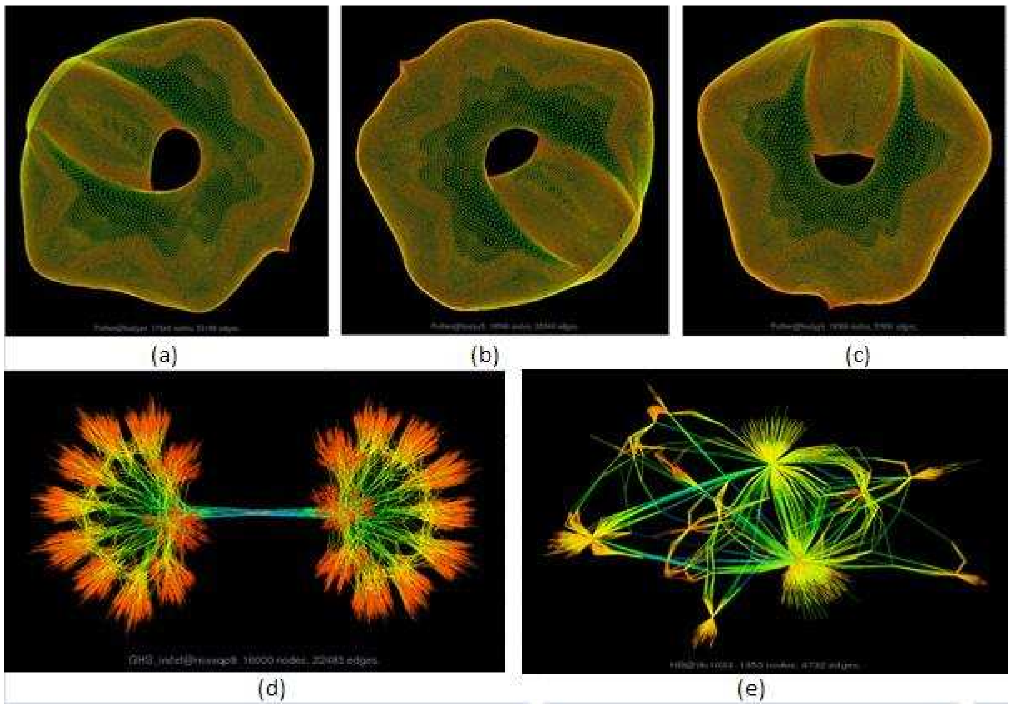


Figure 5.6. The shapes of input matrices used by SpVM application [78] (a)Boddy4, (b)Boddy5, (c)Boddy6, (d)Ncvxqp9, (e)Illc1033.

5.3.1. Performance of Dynamic Locality-aware Mapping

In our algorithm, the main thread obtains the input data and calculates similarities of the data elements accessed in different parts of the program. It groups chunks accessing similar data elements to bins and then begins computation as all other application threads do. In all of the tests reported in this part, the number of application threads is equal to the number of cores in the target architecture, whereas the number of chunks per core are 2, 5, 10, 20, and 50.

The performance of our algorithm is compared with that of standard Linux scheduler. The code running on standard Linux scheduler assigns consecutive blocks of chunks to bins, and each application thread accesses the same amount of data. The standard Linux scheduler places threads in the least loaded processor and performs reactive and proactive dynamic load balancing. When a core becomes idle, a thread from a remote processor is migrated to the idle core in reactive load balancing; in addition, the processor times used by threads are balanced in proactive load balancing. During thread scheduling and migration, data sharing is not considered by the Linux scheduler [22].

The execution times of locality-aware mapping and Linux scheduler for 5 different inputs are given in Figure 5.7. When the number of chunks per bin is equal to 2, data used is divided into 32 parts and all chunks include large amount of program data. Even if 2 chunks have some sharing, the distinct data between them is higher so the similarity between chunks are close to each other. Therefore, we are not able to obtain high similarity between chunks. As a result, both our algorithm and the Linux scheduler have the same performance for all inputs used in this case.

Based on the characteristics of the input data used, different trends between the performance and chunk per bin number are observed. For inputs Boddy4 and Ncvxqp9, we observe the best performance when the chunk number is 50; whereas Boddy5, Boddy6, and Illc1033 inputs have the lowest execution time when 10 chunks are used. When chunk per bin number is changed, the data included in the chunks change, and the similarity between chunks is revised. For inputs Boddy4 and Ncvxqp9, we can obtain the highest similarity among chunks when 50 chunks are considered, but for other inputs 10 chunks are necessary to obtain the highest similarity. As a result, both the performance of our algorithm and Linux scheduler is affected by the varying number of chunks in use.

If we examine Figure 5.7, we see that both the Linux scheduler and our algorithm show similar performance trends for varying number of chunks. For all of the inputs, our algorithm outperforms the Linux scheduler, and this is due to increasing data reuse in the cache as shown in Table 5.3.

Table 5.3. L1 and L2 cache counts (in millions) for Boddy5 and Illc1033 input matrices.

			L1 Cache			**L2 Cache**		
Method	*Input*	Chunk #	Miss #	Access #	Hit %	Miss #	Access #	Hit %
Locality-aware Mapping	Boddy5	2	69.7	2516	97.2	53.3	487	89.1
		5	70.4	2514	97.2	94.6	487	80.6
		10	70.3	2519	97.2	111.9	488	77.1
		20	72.8	2528	97.1	78.7	493	84.0
		50	77.9	2551	97.0	85.4	503	83.1
	Illc1033	2	13.4	789	98.3	26.1	144	81.9
		5	13.8	803	98.3	33.0	148	77.8
		10	15.8	822	98.1	17.4	154	88.8
		20	24.5	867	97.2	20.9	174	88.0
		50	50.9	1040	95.1	28.1	242	88.4
Linux Scheduler	Boddy5	2	69.6	2517	97.2	89.2	487	81.6
		5	70.1	2520	97.2	104.2	488	78.6
		10	70.1	2523	97.2	154.2	489	68.4
		20	70.1	2532	97.2	154.4	491	68.5
		50	70.3	2554	97.2	95.1	497	80.8
	Illc1033	2	13.1	789	98.3	32.3	144	77.6
		5	16.2	991	98.4	35.7	162	77.9
		10	16.3	1016	98.4	36.8	188	80.4
		20	16.9	1070	98.4	44.8	201	77.8
		50	19.2	1288	98.5	46.8	257	81.8

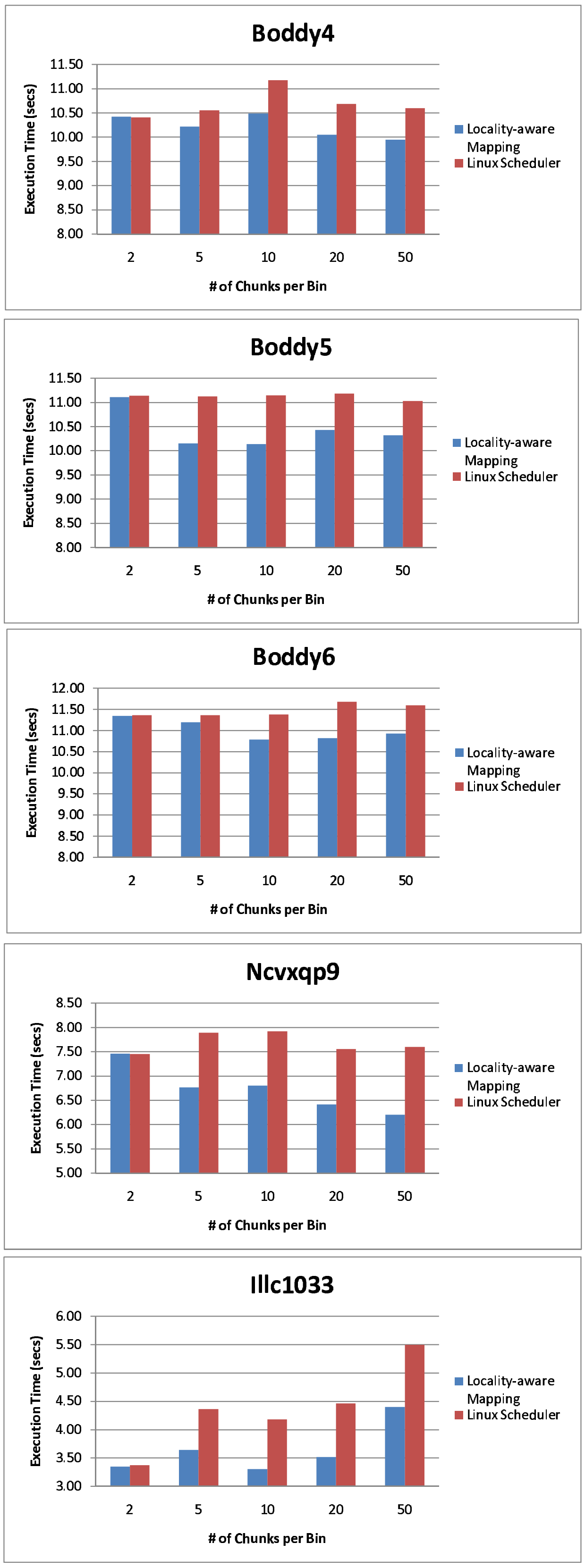


Figure 5.7. Performance of locality-aware dynamic mapping method and Linux scheduler for various input files.

Figure 5.8 shows the speedup values of our algorithm in comparison to the Linux scheduler with respect to different number of chunks for different inputs. The speedup values obtained when Boddy4 input used vary between 3.3% and 6.6%, whereas the average speedup value is 5.6%. For Boddy5 input, the speedup values range between 6.8% and 9.9%, and the average speedup value is 8.3%. The minimum, maximum, and average speedup values for Boddy6 input are 1.4%, 7.9%, and 5.3%. The data elements of Ncvxqp9 and Illc1033 inputs have irregular access pattern and as a result, these inputs show the best performance with average speedup values of 18.3% and 24.5%, respectively. The minimum and maximum speedup values for input Ncvxqp9 are 16.4% and 22.4%, meanwhile the values are 19.8% and 26.8% for Illc1033 input. According to these values, our algorithm shows an average speedup of 12.44% for inputs with different characteristics.

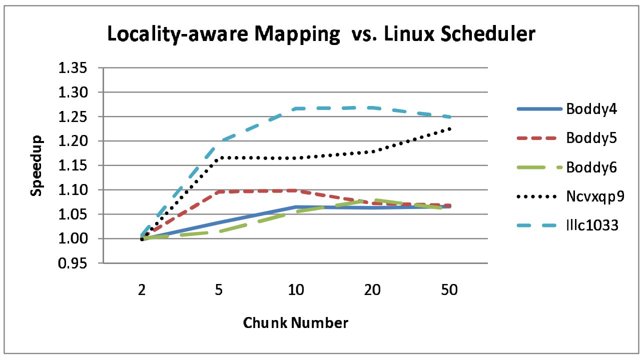


Figure 5.8. Speedup values for different number of chunks.

5.3.2. Sensitivity Analysis

In this section, we conduct sensitivity analysis by changing the default values of cache sizes and chunk numbers. In each experiment below, the value of only one parameter is changed and its effect is observed.

In the first set of tests, to observe the effect of L1 and L2 cache sizes on the algorithm, three tests are conducted. In the first one, L1 and L2 cache sizes are decreased to 4K and 16K, respectively. Both L1 and L2 cache sizes are increased to

16K and 32K in the second test, and to 64K and 128K in the last test. The speedup values obtained are given in Figure 5.9, Figure 5.10, and Figure 5.11. Our algorithm is built on the idea of reusing data on the cache, so when the cache sizes are decreased, the probability of finding the requested data on the cache decreases, which negatively affects the performance. Specifically, the average speedup value when all inputs are considered is 11.94%; whereas when the cache sizes are increased, 12.16% and 12.56% are obtained as the average speedup values. Increasing the cache size decreases the number of cache misses on the cache for both our algorithm and the Linux scheduler.

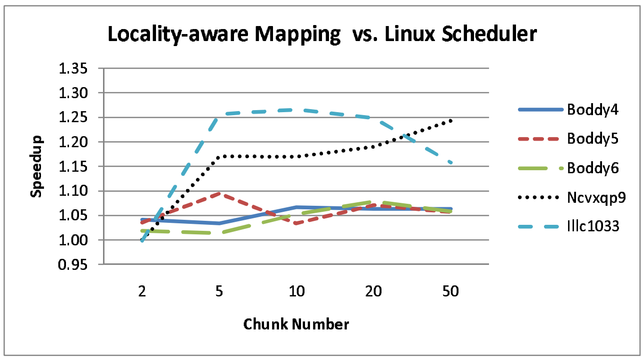


Figure 5.9. Speedup values when 4K L1 cache / 16K L2 cache combination is considered.

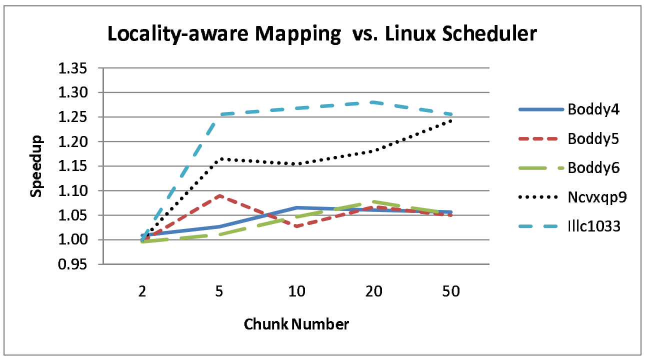


Figure 5.10. Speedup values when 16K L1 cache / 64K L2 cache combination is considered.

In order to observe the performance of our algorithm for an extreme number of chunks, each input row is represented with a chunk and equal number of chunks are distributed to bins. In this case, the chunks contain limited number of data elements

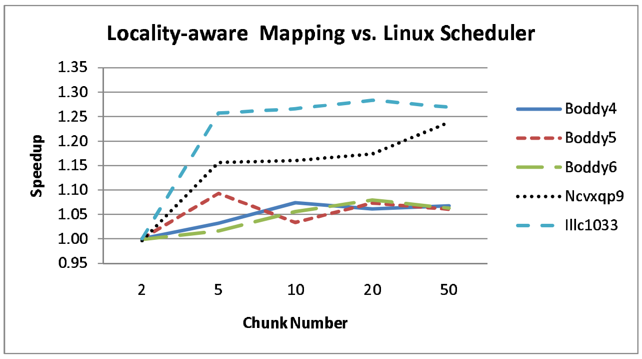


Figure 5.11. Speedup values when 32K L1 cache / 128K L2 cache combination is considered.

and the sharing among them is limited. The similarities between chunks have the highest values where they share all of the data and do not include a single distinct data. As a result, the cache hit ratio and performance decrease in both locality-aware mapping and the Linux scheduler, as shown in Figure 5.12. The cache characteristics of these tests are given in Table 5.4.

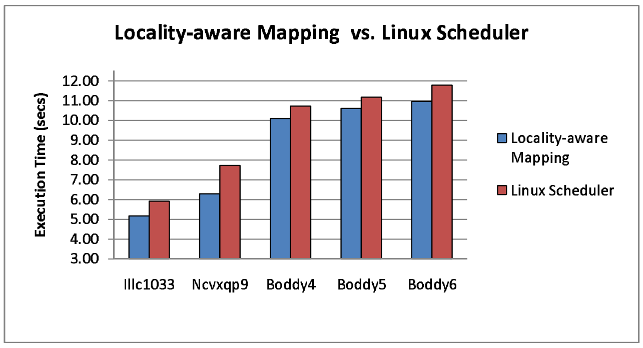


Figure 5.12. Performance of locality-aware dynamic mapping method and Linux scheduler when maximum number of chunks are considered.

Table 5.4. L1 and L2 cache counts (in millions) for the case of maximum number of chunks.

		L1 Cache			L2 Cache		
Method	*Input*	Miss #	Access #	Hit %	Miss #	Access #	Hit %
Locality-aware Mapping	Boddy4	78.9	2431	96.8	67.0	486	86.2
	Boddy5	83.6	2574	96.8	132.4	514	74.3
	Boddy6	86.6	2679	96.8	111.4	487	79.2
	Ncvxqp9	63.5	1395	95.5	69.0	487	77.6
	Illc1033	50.9	1040	95.1	28.1	242	88.4
Linux Scheduler	Boddy4	66.5	2433	97.3	96.2	474	79.7
	Boddy5	70.5	2577	97.3	163.4	502	67.5
	Boddy6	73.9	2684	97.2	143.3	524	72.6
	Ncvxqp9	56.6	1404	96.0	98.2	304	67.7
	Illc1033	19.2	1288	98.5	46.8	257	81.8

6. CONCLUSIONS AND FUTURE WORK

The focus of this thesis is to explore the ways to adapt the execution of an application to the underlying hardware to improve the performance of applications running on CMPs. To do this, we consider three steps for application mapping namely (i) analyze application behavior and partition application code into multiple threads for execution; (ii) assign threads onto parallel processors on the chip and the data they manipulate onto memory components; and (iii) re-adjust thread and data mapping, as necessary, at runtime to reduce application execution latency. Thus, the thesis work considers both static data and thread mapping at compile time as well as dynamic data and thread re-mapping at runtime to satisfy the goals. By incorporating both static and dynamic components, our approach aims to collect the benefits of code analysis (which provides critical information about the behavior of the entire application) and runtime adaptation (which can exploit execution time information about resource availability and application behavior).

We have proposed and evaluated three application mapping approaches in this thesis. In our first approach, we present and evaluate two parallel implementations of the Barnes-Hut method on the Cell BE architecture. The memory requirement of this application is very high, as both the local tree representing the domain and the positions and masses of the particles should be simultaneously stored in the Local Stores of the SPEs. Consequently, an efficient workload distribution on the SPEs is needed. In our first parallel version, the domain is divided into sub-domains, and the sub-domains are assigned to SPEs to overcome memory needs and to achieve load balance. The method contains three phases. In the first phase, the domain is decomposed into smaller sub-domains and the sub-domains are assigned to the SPEs. In the second phase, tree construction and force calculation in each sub-domain are performed for sub-domains. Finally in the third phase, the velocities and positions of the particles are updated. In this approach only the particles in the assigned sub-domains are used in the local tree construction phase, so they do not contain many of the important clusters that are stored in the higher levels of the tree.

In our enhanced version of parallelization and mapping, a global tree representing the whole workspace is constructed by the PPE, and nodes including all clusters of the workspace are assigned to SPEs. This increases the effort in the tree construction phase, but decreases the execution time of the most time-consuming phase (namely force calculation) significantly. After the tree construction phase is completed, the algorithm follows the same steps as in the first parallel version. Although programming the Cell architecture is more difficult than programming the Intel Xeon (due to explicit on-chip memory management), our results show that the required extra effort pays off. As part of the experimental study, the performances of the two proposed versions are measured by using different number of SPEs, different λ values, and different number of particles. Our experimental evaluation indicates that this application performs much faster on the Cell BE compared to the Intel Xeon based system. Specifically, our first and second methods on the Cell BE outperform Intel Xeon by a factor of 5.8 and 7.1 for 8192 particles, respectively.

In our thread-to-core mapping methodology, we proposed a framework that uses hardware and software counters to dynamically tune the execution of the application. HW counters are collected through our simulation environment Simics; whereas SW counters are implemented as four helper threads running parallel with an application code. We have developed an effective and inexpensive framework and increased the data reuse of on-chip memory components. Despite of optimizations on data sharing and load balancing, due to the overheads of thread migration and helper threads, we were not able to gain performance improvements in this study. We have changed architectural resources and their connections as well as benchmark applications and the input data, but in none of the cases, we could manage to outperform Linux scheduler. If need of resources change dynamically during execution, then this approach may pay off.

In the third application mapping approach, we present a cache-centric mapping which dynamically assigns data to threads. The objective of this is to assign data with similar access patterns to the same on-chip memory components. Our algorithm partitions data of a given application into chunks in order to provide better load balancing; and a set of chunks with similar access patterns is grouped into bins and mapped to caches to provide data locality. Our goal is to decrease cache contention by increasing data reuse in the L2 cache. We validate applicability of our algorithm via Sparse matrix-vector multiplication and validation is performed by considering five input matrices with different shapes and characteristics. Our experimental evaluation indicates that our algorithm outperforms the Linux scheduler by an average of 12.5% on the basis of execution time.

There will be three directions for the future work of our hierarchical data and thread assignment methodology. We can use our methodology to map other multi-threaded applications with irregular data access patterns such as Lattice-Boltzmann application to further validate the approach. We use the ratio of total number of shared data elements to the total number of accessed data elements as the similarity metric in this approach. This metric can be replaced with the total number of shared elements between chunks. Finally, we can use different architectural structures for evaluating the proposed method. In our initial work, we have used private L1 and L2 caches connected to each core; thus only the application threads running on the same core can use the advantage of locality analysis. If the L2 caches are shared among multiple cores, then the impact of data sharing among multiple threads running on different cores can also be measured. For a multicore architecture containing 16 cores, L2 caches can be shared by 2 or 4 cores and the number of application threads can be increased to 32, 64 or 128.

APPENDIX A: IBM CELL BROADBAND ENGINE

Cell B./E. processor is different compared to traditional architectures, it contains many smaller processors that work in parallel rather than having a single large processor. The architecture is designed to exploit parallelism. The individual processing elements are also much simpler than most modern CPUs. They don't have many properties most processors have, such as cache management, and branch prediction. The clock speed of Cell processor is 3.2 GHz and it has a single-precision peak performance of 204.8 Gflops/s and double-precision peak performance of 14.6 Gflops/s.

A.1. IBM Cell Broadband Engine Architecture

The Cell B./E. consists of a traditional microprocessor (PPE), 8 smaller, simpler processors (SPEs) and an element interconnect bus (EIB) which connects the processors and provides access to main memory and I/O devices.

A.1.1. Power Processing Element

PPE is the main processor in the Cell B./E.. It is a 64-bit multithreaded PowerPC core with two levels of on-chip cache, 32 Kbyte L1 instruction and 32 Kbyte L1 data cache, 512 Kbyte L2 cache. The PPE is a dual issue processor so it supports two way simultaneous multithreading. SMP feature of the PPE is similar to Intel's Hyper-Threading technology. PPE seems to provide two independent execution units to the software layer. PPE uses IBM's VMX feature for SIMD instructions [54]. PPE is responsible for running the operating system and applications, distributing workload among SPEs and coordinating them. The design of the PPE is very simple, it does not have advanced scheduling capabilities because it is not designed for running applications but for coordinating SPEs, so it works in low frequencies as compared to traditional processors.

A.1.2. Synergistic Processor Elements

The real power of the CELL processor lies in eight synergistic processing units. Each SIMD co-processing unit deals with computation. Each SPE consists of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). Each MFC has a DMA controller, a Memory Management Unit (MMU), a bus interface unit and an atomic synchronization unit. Figure A.1 summarizes the contents of a single SPE.

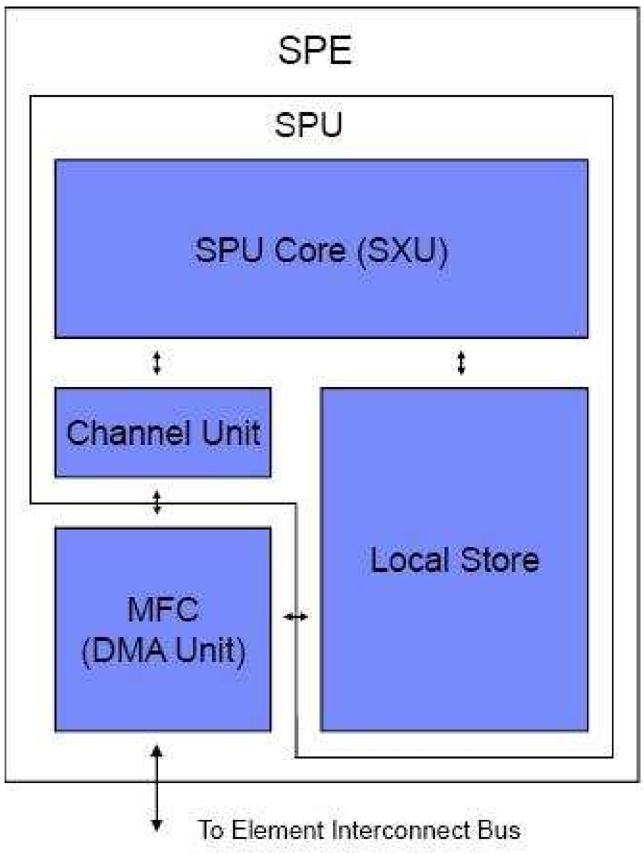


Figure A.1. SPE block diagram [55].

SPE's architectural design is simple like the PPE, they don't support out of order execution, branch prediction. SPEs contain 128 128-bit registers, 4 floating point units and 4 integer units. The SPEs instruction set is designed to take advantage of 128-bit registers, and most of the instructions are SIMD instructions. SIMD instructions can operate on 2-way 64-bit double precision floating point numbers, 4-way 32-bit integers or single precision floating point numbers, 8-way 16-bit integers or 16-way

8-bit integers. The programmer decides how to use these 128-bit registers. Memory operations access 128-bits at a time even if the request data is 8-bits, so for an efficient implementation, the programmer should request 128-bits in each memory operation.

SPEs are in-order processors with two instruction pipelines, even pipeline and the odd pipeline, the even pipeline is responsible for arithmetic operations, while the odd pipeline deals with memory and branch instructions. In a single clock cycle, SPEs can dispatch two instructions if these instructions have no data dependency in between them, otherwise it executes one instruction in one clock cycle. The organization of these two pipelines can be seen in Figure A.2.

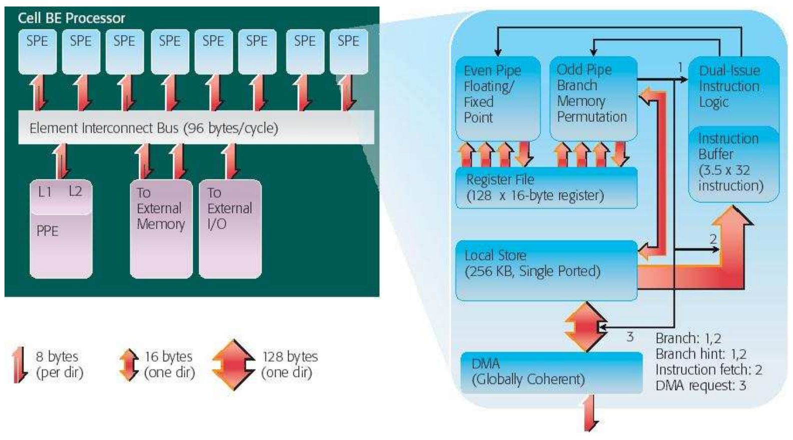

Cell BE Processor
SPE
Element Interconnect Bus (96 bytes/cycle)
L1
L2
PPE
To External Memory
To External I/O
8 bytes (per dir)
16 bytes (one dir)
128 bytes (one dir)
Even Pipe Floating/ Fixed Point
Odd Pipe Branch Memory Permutation
Dual-Issue Instruction Logic
Instruction Buffer (3.5 x 32 instruction)
Register File (128 x 16-byte register)
Local Store (256 KB, Single Ported)
DMA (Globally Coherent)
1
2
3
Branch: 1,2
Branch hint: 1,2
Instruction fetch: 2
DMA request: 3

Figure A.2. SPE Architecture [55].

SPEs do not have a cache, but they have a 256 Kbyte Local Store which can be called private memory. Both the program and the data should be in LS to be executed, SPEs have no direct access to main memory. Unlike traditional architectures data is not moved between the main memory and the LS by the processor itself, but the programmer decides what to put in the local store, so the efficient usage and the management of the LS is left to the programmer or the compiler. This reduces the complexity of the architecture, but increases the complexity of programming.

SPEs can perform only one memory operation at a time, there is no separate instruction and data cache, so in one cycle, either an instruction or data can be fetched from memory. For this purpose, there is an instruction fetch buffer which fetches a bunch of pending 32-bit instructions as a block, but this buffer can only be filled when it is empty or when the programmer tells it to do so. The SPU is a statically scheduled architecture therefore it does not support dynamic branch prediction; the processor assumes that branches are not taken and fills the instruction fetch buffer according to this assumption. The penalty of a branch miss prediction is 18 cycles, which is costly. To overcome this penalty, software branch prediction (branch hint) is used. SPE fetches the instructions at the branch target address so that there will be no penalty for the branches. Traditional processors handles branches, but Cell forces the programmer or the compiler to handle branches.

SPU has no direct access to main memory, so it has DMA controller which performs high bandwidth data transfers between the local store, main memory and other local stores. If the data needed resides in other SPE's LS, then fetching the data from the other LS has two advantages, it is faster to go to another LS than to the memory, and by this way we can avoid data inconsistencies. In order to read or write data, SPE sends DMA commands to MFC and MFC brings data from memory to local store, or writes results back to main memory. DMA commands are queued in the MFC, either polling or blocking interfaces are used to check whether the transfer is completed or not. DMA is non-blocking, SPU is not interrupted while DMA transactions are performed, so this supports the scheduling of two SIMD instructions, compute and memory instruction, per cycle.

MFC runs at the EIB's frequency and supports aligned transfers of 1, 2, 4, 8 bytes, or a multiple of 16-bytes up to 16 KB and it consists of several DMA engines that are used to transfer data between memory and LS. The data requested from either the memory or other LS is written in a DMA-list by specifying the source/destination addresses and the length of the requested data. DMA list commands can request a list of up to 2048 DMA transfers using a single MFC. Peak performance is achieved when the size of a data transfer is multiples of 128-bits. Each DMA command can be assigned a tag identifier and this identifier can be used to enforce in-order execution of two or more commands with a common tag identifier. This identifier can also be used to monitor the commands status. SPUs can also use signals or mailboxes for performing simple, low-latency communication between the PPE and the SPEs.

A.1.3. Element Interconnect Bus

The Element Interconnection Bus connects all components of the CELL processor including the PPE's, the SPE'S, the main memory and I/O as shown in Figure A.3. It supports a peak bandwidth of 204.8 Gbytes/s [56]. EIB is build of 16-byte wide four unidirectional rings, two in each direction. Each ring can allow three concurrent data transfers if their paths do not overlap. For a data transfer request, the EIB decides on which ring to take to travel in the direction of the shortest transfer and makes sure that no request travels more than halfway around the ring. Figure A.4 shows an example of eight concurrent data transfers by using four rings. This shows that even if there are four rings available, each ring can support more than one data transfer at the same time if the paths of the transfers do not overlap.

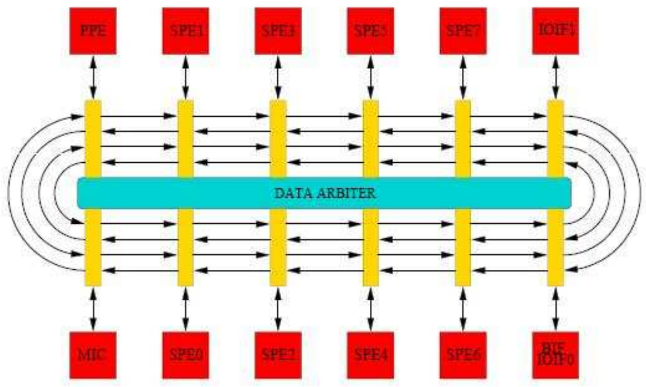


Figure A.3. Internal structure of Cell B./E. [57].

A.2. Application Development on IBM Cell Broadband Engine

Cell processor has many parallel and distributed computational and communicational resources. The resources give Cell processor a large performance advantage over traditional single core processors, but large amount of heterogeneous resources

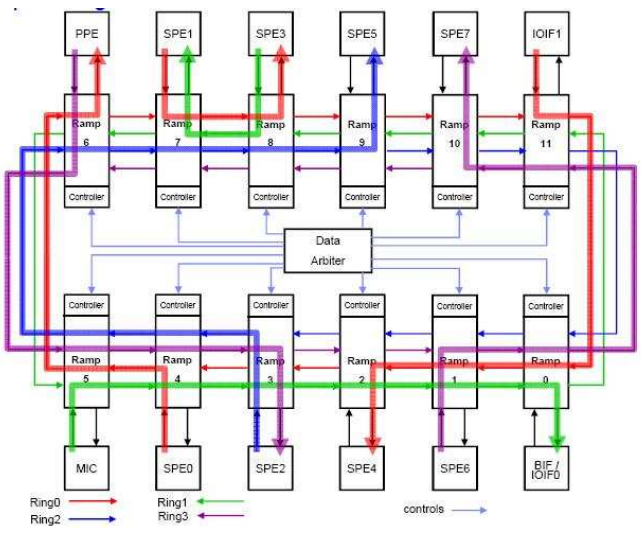


Figure A.4. Example of eight concurrent data transfers [58].

increases the complexity of programming [42, 59].

From the instruction architecture point of view, PPE and SPE cores are executed over different address spaces. PPE instructions execute over the system memory mapped to the effective address space, whereas the SPE instructions execute directly over their own local store. Both the data and code of the SPE program must reside in the local store to be executed. Each core has their own memory access controller to transfer data to and from the system memory or I/O devices. There is a latency associated with each data transfer. The latency comes from both the time needed to setup a DMA operation and the time needed to actually transfer the data. Experimental studies showed that the main latency of data transfer is not the actual time needed to transfer data, but the time needed to set up a DMA operation, so the amount of data requested per DMA operation should be at least 128 bytes, or even higher (512 bytes). This latency is an important performance consideration, and we need to make sure that any memory transfer does not become a bottleneck to a program's execution.

While programming on Cell B./E., one must keep in mind both the architectural properties and heterogeneous resources of the architecture. Heterogeneous resources of the processor provides two levels of parallelism, and if one can combine two levels of parallelism in the implementation of an application, the performance improvement can be two orders of magnitude over single-core scalar processors.

A.2.1. Levels of Parallelism

Resources available on the Cell processor, computational and communicational resources, provide two levels of parallelism. One is at the instruction level, and the other is at the task level.

- Instruction Level Parallelism: Depending on the size of the data, performance improvement can be 2 times to 16 times as compared to single instruction single data type of operation. To achieve instruction level parallelism, SPE SIMD engine and PPE VMX engine are used. SPU has 128 128-bit registers and instructions are SIMD instructions. An instruction operates on all elements in SIMD register at once. So the data should be vectorized to take full advantage of SIMD instructions. SIMDizing code can be done either by the compiler (Auto SIMDization) or by the programmer by using instrincts. Instrincts can be treated as high level assembly instructions, and low level C functions.
- Task Level Parallelism: PPE can distribute 8 independent tasks on the 8 independent cores and execute them at the same time. Task distribution among SPEs changes according to the dependencies between tasks and the amount of space needed for both the code and data of the task. For this purpose, four different programming models are introduced and according to the properties of the applications, the most suitable programming model should be chosen for an efficient implementation.

A.2.2. Cell Programming Models

To deal with the endless possibilities in mapping an algorithm to instruction and task level parallelism and the distribute resources among SPEs, Cell Programming Models are introduced. The possible programming models for a single Cell environment are PPE programming models and SPE programming models which are described in detail below.

- PPE Programming Models: The traditional programming model on the PPE is the simplest programming model where it does no computation but coordinates the SPEs so it serves as a controller and establishes a runtime environment for SPE programs for memory mapping, exception handling and SPE run control. It allocates and manages Cell system resources, such as SPE scheduling and it provides OS services to SPE programs and threads. PPE acts as an operating system service mechanism to the SPE to perform printf, file I/O, memory management without programmer interruption.
- SPE Programming Models: SPE programming models mainly differ according to the amount of data needed. The working area of an SPE is a relatively small memory so we may not be able to put the whole data and instructions into LS.
  - (i) Small Single-SPE Model: This is the first and most simple SPE programming model, it is a single task environment. It contains a simple SPE program that fits in the 256 KB Local Store. Data is brought in using DMA transactions, addresses can be brought in using mailbox transactions. SPE side system calls that are handled by the PPE, are controlled using explicit input and output from the SPE program.
  - (ii) Large Single-SPE Programming Model: This programming model is used when the SPE program does not fit in the 256 KB Local Store and DMA transactions are used to pull in the data into the local store and push the results back to the main memory.For large size input data, the streaming model is used where the data is streamed in and out of the system memory. The data is first brought into the local store, then processed and it is pushed back to the secondary memory. Both instructions and data are stored in Local Store so instructions can be streamed in and out of the system memory as well. If the application's code segment is larger than 256 KB LS, we can split that code segment into multiple plug-ins and use a manual overlay framework (manual plug-in framework) to bring in data associated with

code segment into the local store, execute that data and then overwrite that data with an other plug-in as necessary. Part of this SW managed cache is also an automatic SW managed code overlay. Another model for Large single-SPE programming model is the Job Queue where we have a very small SPE kernel that is running within the LS and simply using the system memory to pull in jobs that are being queued up by either the PPE or some other external force within the system. Code and data packaged together as inputs to an SPE kernel program and the SPE kernel program executes this code and data as needed.

(iii) Parallel Programming Models: The Cell processor has traditional parallel programming models applicable. It is based on interacting single-SPE programs. Parallel SPE program have several methods for synchronization including cache line-based MFC atomic update commands (similar to the PowerPC lwarx, ldarx, stwcx, and stdcx instructions), SPE input and output mailboxes with PPE, SPE signal notification register, SPE events and interrupts and SPE busy poll of shared memory location. Parallel programming models are shared memory, job queue, message passing and pipeline (streaming).

In shared memory programming model, the data is accessed by its address. With proper locking mechanism, large SPE programs may access shared memory objects located in the effective-address space. There is also compiler OpenMP support to allow shared memory type of operation.

Another parallel programming method is job queue model. Large set of jobs are fed through a group of SPE programs and SPEs do not care which job they are assigned in job queue model. Streaming is a special case of job queue with regular and sequential data. Each SPE program locks on the shared job queue to obtain next job. It locks onto this job queue using synchronization and atomic locking mechanisms that are available to it. For uneven jobs, workloads are self-balanced among available SPEs because the SPEs do not interact with each other so it does not matter how long one job takes compared to the other as long as the shared memory area is locked whenever it is accessed. The entire workload should be self-balanced.

Message passing methodology is based on data access by connection so it is sequential in nature. This type of programming is applicable to SPE programs where addressable data space only spans over local store so the amount of interaction with the system memory and/or other SPEs is not very high.

Pipeline programming is much more flexible since it uses the bandwidth between SPEs and not the system memory bandwidth which can become a significant bottleneck. The biggest problem with this type of operation is that the load-balance is difficult, so the execution time of each SPE should be close to each other.

(iv) Multi-tasking SPEs Programming Model: In Multi-tasking SPEs model, the LS contains several tasks that can be executed in parallel. There is no memory protection running among tasks, however there is a co-operative, non-preemptive, event-driven scheduling mechanism, where multiple tasks are performed. If there are large number of small operations that a single SPE is responsible for, the PPE can schedule up a series of events and then in the PPE's dispatch order, the SPE performs the operations. For this case, another alternative is to use software-managed threads, where each SPE has multiple threads dedicated to each of the small jobs. If the tasks, the codes and the data for these jobs do not fit in the LS, then SPEs can context switch out of the LS and schedule back later any job within the system, as in self-managed multi tasking model.

REFERENCES

1. Dual-Core Intel Itanium Processor, http://www.intel.com/pressroom/kits/itanium2, 2009.

2. AMD Athlon, http://www.amd.com/us-en/Processors/ProductInformation, 2009.

3. Kongetira, P., K. Aingaran and K. Olukotun, "Niagara: A 32-Way Multithreaded SPARC Processor", *IEEE MICRO Magazine*, Vol. 25, No. 2, pp. 21-29, 2005.

4. Kahle, J., M. Day, H. Hofstee, C. Johns, T. Maeurer and D. Shippy, "Introduction to the Cell Multiprocessor", *IBM Journal of Research and Development*, Vol. 49, No. 4-5, pp. 589-604, 2005.

5. Intel® Xeon® Processor 5000, http://www.intel.com/products/processor/xeon5000, 2009.

6. Teraflops Research Chip, http://techresearch.intel.com/articles/Tera-Scale/1449.htm, 2010.

7. Taylor, M. B., J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe and A. Agarwal, "The RAW Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs", *IEEE Micro*, Vol. 22, No. 2, pp. 25-35, 2002.

8. Sankaralingam, K., R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler and C. R. Moore, "Exploiting ILP, TLP, and DLP with Polymorphous TRIPS Architecture", *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 422-433, 2003.

9. Swanson, S., K. Michelson, A. Schwerin and M. Oskin, “Wavescalar”, *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 291-302, 2003.

10. Mai, K., T. Paaske, N. Jayasena, R. Ho, W. J. Dally and M. Horowitz, “Smart Memories: A Modular Reconfigurable Architecture”, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 161-171, 2000.

11. Ren, B., “System Design for Chip Multiprocessor”, 2004, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.8569&rep=rep1-&type=pdf.

12. Olukotun, K. and L. Hammond, “The Future of Microprocessors”, *ACM Queue*, pp. 26-29, 2005.

13. Kumar, R., D. M. Tullsen, N. Jouppi and P. Ranganathan, “Heterogeneous Chip Multiprocessors”, *Computer Journal*, Vol. 38, No. 11, pp. 32-38, 2005.

14. Kumar, R., K. I. Farkas, N. P. Jouppi, P. Ranganathan and D. M. Tullsen, “Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction”, *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 81-92, 2003.

15. Kumar, R., D. M. Tullsen, P. Ranganathan, N. P. Jouppi and K. I. Farkas, “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance”, *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 64-75, 2004.

16. Kumar, R., N. P. Jouppi and D. M. Tullsen, “Conjoined-Core Chip Multiprocessing”, *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 195-206, 2004.

17. Sherwood, T., E. Perelman, G. Hamerly, S. Sair and B. Calder, "Discovering and Exploiting Program Phases", *IEEE Micro*, Vol. 23, No. 6, pp. 84-93, 2003.

18. Sherwood, T., S. Sair and B. Calder, "Phase Tracking and Prediction", *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 336-349, 2003.

19. Becchi, M. and P. Crowley, "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures", *Proceedings of the 3rd Conference on Computing Frontiers*, pp. 29-40, 2006.

20. Sondag, T., V. Krishnamurthy and H. Rajan, "Predictive Thread to Core Assignment on a Heterogeneous Multi-Core Processor", *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, 2007.

21. Sondag, T. and H. Rajan, "Phase-Guided Thread-to-Core Assignment for Improved Utilization of Performance-Asymmetric Multi-Core Processors", *Proceedings of the ICSE Workshop on Multicore Software Engineering*, pp. 73-80, 2009.

22. Tam, D., R. Azimi and M. Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors", *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp. 47-58, 2007.

23. Hong, S., S. H. K. Narayanan, M. Kandemir and O. Ozcan, "Process Variation Aware Thread Mapping for Chip Multiprocessors", *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 821-826, 2009.

24. Bowman, K. A., A. R. Alameldeen, S. T. Srinivasan and C. B. Wilkerson, "Impact of Die-to-Die and Within-Die Parameter Variations on the Throughput Distribution of Multicore Processors", *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, pp. 50-55, 2007.

25. Cao, Y. and L. T. Clark, “Mapping Statistical Process Variations Toward Circuit Performance Variability: An Analytical Modeling Approach”, *Proceedings of Design Automation Conference*, pp. 658-663, 2005.

26. Augonnet, C., S. Thibault, R. Namyst and P. Wacrenier, “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”, *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pp. 863-874, 2009.

27. Wang, Z. and M. OBoyle, “Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach”, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 75-84, 2009.

28. Balasundaram, V., G. Fox, K. Kennedy and U. Kremer, “A Static Performance Estimator to Guide Data Partitioning Decisions”, *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 213-223, 1991.

29. Kim, S., D. Chandra and Y. Solihin, “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture”, *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 111-122, 2004.

30. Kandemir, M., O. Ozcan and S. P. Muralidhara, “Dynamic Thread and Data Mapping for NOC Based CMPs”, *Proceedings of the 46th Annual Design Automation Conference*, pp. 852-857, 2009.

31. Jiang, Y., K. Tian and X. Shen, “Combining Locality Analysis with Online Proactive Job Co-Scheduling in Chip Multiprocessors”, *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers*, pp. 201-215, 2010.

32. Ding, C. and Y. Zhong, "Predicting Whole-Program Locality with Reuse Distance Analysis", *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 245-257, 2003.

33. Fedorova, A., M. Seltzer, M. Smith and C. Small, "CASC: A Cache-Aware Scheduling Algorithm for Multithreaded Chip Multiprocessors", Technical Report TR-2005-0142, Sun Labs, 2005.

34. Chen, S., P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry and C. Wilkerson, "Scheduling Threads for Constructive Cache Sharing on CMPs", *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 105-115, 2007.

35. Chandra, D., F. Guo, S. Kim and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multiprocessor Architecture", *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 340-351, 2005.

36. Pichel, J. C., D. E. Singh and J. Carretero, "Reordering Algorithms for Increasing Locality on Multicore Processors", *Proceedings of 10th IEEE International Conference on High Performance Computing and Communications*, pp. 123-130, 2008.

37. Hofstee, H. P., "Power Efficient Processor Architecture and the Cell Processor", *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pp. 258-262, 2005.

38. Pham, D., S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki and K. Yazawa, "The Design and Implementation of a First-Generation Cell Processor", *International Solid State Circuits Conference*, pp. 184-185, 2005.

39. Williams, S., J. Shalf, L. Oliker, S. Kamil, P. Husbands and K. Yelick, "The Potential of the Cell Processor for Scientific Computing", *Proceedings of the 3rd Conference on Computing Frontiers*, pp. 9-20, 2006.

40. Bader, D. A. and V. Agarwal, "FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine", *Proceedings of the 14th IEEE International Conference on High Performance Computing*, pp. 172-184, 2007.

41. Chow, A. C., G. C. Fossum and D. A. Brokenshire, "A Programming Example: Large FFT on the Cell Broadband Engine", *Technical Conference Proceedings of the Global Signal Processing Expo*, 2005.

42. Cico, L., R. Cooper and J. Greene, "Performance and Programmability of the IBM/Sony/Toshiba Cell Broadband Engine Processor", *White Paper*, 2006.

43. Bader, D. A., V. Agarwal and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking", *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2007.

44. Villa, O., D. P. Scarpazza, F. Petrini and J. F. Peinador, "Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-Core Processors", *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2007.

45. Sachdeva, V., M. Kistler, E. Speight and T. H. K. Tzeng, "Exploring the Viability of the Cell Broadband Engine for Bioinformatics Applications", *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2007.

46. Asanovic, K., R. Bodik, B. C. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", Technical Report UCB/EECS-2006-183, EECS Department, Berkeley, CA, 2006.

47. Barnes, J. and P. Hut, "A Hierarchical O(n log n) Force Calculation Algorithm", *Nature*, Vol. 324, pp. 446-449, 1986.

48. Demiroz, B., H. Topcuoglu, M. Kandemir and O. Tosun, "Parallelizing Barnes-Hut Method on the Cell BE Architecture", *Third Workshop on Programmability Issues for Multicore Computers*, 2010.

49. Grama, A., V. Kumar and A. Sameh, "Scalable Parallel Formulations of the Barnes-Hut Method for N-Body Simulations", *Proceedings of Supercomputing Conference*, pp. 439-448, 1994.

50. Greengard, L., *The Rapid Evolution of Potential Fields in Particle Systems*, The MIT press, 1988.

51. Ramachandran, U., G. Shah, A. Sivasubramaniam, A. Singla and I. Yanasak, "Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors", *Proceedings of Supercomputing Conference*, pp. 62-82, 1995.

52. Warren, M. and J. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm", *Proceedings of Supercomputing Conference*, pp. 12-21, 1993.

53. Singh, J., C. Holt, T. Totsuka, A. Gupta and J. Hennessy, "Load Balancing and Data Locality in Hierarchical N-Body Methods", *Journal of Parallel and Distributed Computing*, 1992.

54. Eichenberger, A. E., J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao and R. Koo, "Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture", *IBM Systems Journal*, Vol. 45, 2006.

55. Bader, D. and F. Petrini, "IBM Cell B./E. Architecture", 2008, http://www.first.dk/?n=Events.AFAPA.

56. Kistler, M., M. Perrone and F. Petrini, “Cell Multiprocessor Communication Network: Built for Speed”, *IEEE Micro*, Vol. 26, No. 3, pp. 10-23, 2006.

57. Scarpazza, D., O. Villa and F. Petrini, “Peak Performance DFA-Based String Matching on the Cell Processor”, *Proceedings of Parallel and Distributed Processing Symposium*, 2007.

58. IBM Course on Cell Architecture, http://publib.boulder.ibm.com, 2006.

59. IBM Course on Cell Software Programming Model, http://publib.boulder.ibm.com, 2006.

60. Bader, D. A., V. Agarwal, K. Madduri and S. Kang, “High Performance Combinatorial Algorithm Design on the Cell Broadband Engine Processor”, *Parallel Computing*, Vol. 33, No. 10-11, pp. 720-740, 2007.

61. Brokenshire, D. A., “Maximizing the Power of the Cell Broadband Engine Processor: 25 Tips to Optimal Application Performance”, *IBM Developer Works*, 2006.

62. Virtutech Simics 4.0, http://www.virtutech.com, 2008.

63. Magnusson, P. S., M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner, “Simics: A Full System Simulation Platform”, *IEEE Computer*, Vol. 35, No. 2, pp. 50-58, 2002.

64. Parsec Benchmark Suite, http://parsec.cs.princeton.edu, 2008.

65. Black, F. and M. Scholes, “The Pricing of Options and Corporate Liabilities”, *Journal of Political Economy*, Vol. 81, No. 3, pp. 637-654, 1973.

66. Sun Studio 12.1, http://developers.sun.com/sunstudio, 2007.

67. Wellein, G., G. Hager, A. Basermann and H. Fehske, "Fast Sparse Matrix-Vector Multiplication for TeraFlop/s Computers", *High Performance Computing for Computational Science*, pp. 287-301, 2002.

68. Geus, R. and S. Rollin, "Towards a Fast Parallel Sparse Matrix-Vector Multiplication", *Proceedings of the International Conference on Parallel Computing*, pp. 308-315, 1999.

69. Haque, S. A., S. Hossain and M. Maza , "Cache Friendly Sparse Matrix-Vector Multiplication", *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pp. 175-176, 2010.

70. Pinar, A. and M. Heath, "Improving Performance of Sparse Vector Multiplication", *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1999.

71. Demmel, J., J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. W. Vuduc, R. C. Whaley and K. Yelic, "Self-Adapting Linear Algebra Algorithms and Software", *Proceedings of IEEE*, Vol. 93, No. 2, pp. 293-312, 2005.

72. Im, E., K. Yelick and R. W. Vuduc, "SPARSITY: Optimization Framework for Sparse Matrix Kernels", *International Journal of High Performance Computing Applications*, Vol. 18, No. 1, pp. 135-158, 2004.

73. Williams, S., R. W. Vuduc, L. Oliker, J. Shalf, K. Yelick and J. Demmel, "Optimizing Sparse Matrix-Vector Multiply on Emerging Multicore Platforms", *Journal of Parallel Computing*, Vol. 35, No. 3, pp. 178-194, 2009.

74. Choi, J. W., A. Singh and R. W. Vuduc, "Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs", *Proceedings of the ACM SIGPLAN Symposium Principles and Practice of Parallel Programming* , pp. 115-126, 2010.

75. Bolz, J., I. Farmer, E. Grinspun and P. Schroder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid", *ACM Transactions on Graphics*, Vol. 22, pp. 917-924, 2003.

76. Strout, M. M., L. Carter and J. Ferrante, "Rescheduling for Locality in Sparse Matrix Computations", *Proceedings of the International Conference on Computational Science*, pp. 28-30, 2001.

77. Srikantaiah, S., M. Kandemir and M. Irwin, "Adaptive Set Pinning: Managing Shared Caches in CMPs", *Proceedings of 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

78. Davis, T., "The University of Florida Sparse Matrix Collection", 2010, http://www.cise.ufl.edu/research/sparse/matrices.