EFFECTIVE DATA PARALLEL COMPUTING ON MULTICORE PROCESSORS

by

Jong-Ho Byun

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical and Computer Engineering

Charlotte

2010

Approved by:

______________________________
Dr. Arun Ravindran

______________________________
Dr. Arindam Mukherjee

______________________________
Dr. Bharat Joshi

______________________________
Dr. Gabor Hetyei

©2010
Jong-Ho Byun
ALL RIGHTS RESERVED

# ABSTRACT

JONG-HO BYUN. Effective data parallel computing on multicore processors.
(Under direction of DR. ARUN RAVINDRAN)

The rise of chip multiprocessing or the integration of multiple general purpose processing cores on a single chip (multicores), has impacted all computing platforms including high performance, servers, desktops, mobile, and embedded processors. Programmers can no longer expect continued increases in software performance without developing parallel, memory hierarchy friendly software that can effectively exploit the chip level multiprocessing paradigm of multicores. The goal of this dissertation is to demonstrate a design process for data parallel problems that starts with a sequential algorithm and ends with a high performance implementation on a multicore platform. Our design process combines theoretical algorithm analysis with practical optimization techniques. Our target multicores are quad-core processors from Intel and the eight-SPE IBM Cell B.E. Target applications include Matrix Multiplications (MM), Finite Difference Time Domain (FDTD), LU Decomposition (LUD), and Power Flow Solver based on Gauss-Seidel (PFS-GS) algorithms. These applications are popular computation methods in science and engineering problems and are characterized by unit-stride (MM, LUD, and PFS-GS) or 2-point stencil (FDTD) memory access pattern. The main contributions of this dissertation include a cache- and space-efficient algorithm model, integrated data pre-fetching and caching strategies, and in-core optimization techniques. Our multicore efficient implementations of the above described applications outperform naïve parallel implementations by at least 2x and scales well with problem size and with the number of processing cores.

# ACKNOWLEDGMENTS

First of all, I would like to deeply thank my advisor Dr. Arun Ravindran for his great support, guidance, patience and encouragement. I would also like to thank Dr. Arindam Mukherjee and Dr. Bharat Joshi for their special help and encouragement. I am also indebted to Dr. Gabor Hetyei for devoting his time to the review of my work.

I would also like to thank David Chassin at Pacific Northwest National Lab (PNNL) for his special support. Specially thanks to all of my friends for their encouragement.

Finally, I deeply thank my dear family; my parents, sisters' families and my aunt's family for their love, support and encouragement. I cannot imagine myself going through all this work without them.

# TABLE OF CONTENTS


LIST OF FIGURES ix

LIST OF TABLES xiii

CHAPTER 1: INTRODUCTION 1

1.1 Rise of Multicore Computing 1

1.2 Research Goals 2

1.3 Dissertation Contributions 6

1.4 Dissertation Outline 7

CHAPTER 2: TRENDS IN MULTICORE COMPUTING 9

2.1 Introduction 9

2.2 Multicore Architectures 10

2.2.1 Historical Trends 10

2.2.2 Architectural Elements 12

2.2.3 Case Studies 15

2.2.3.1 Intel Gainestown 15

2.2.3.2 Sun UltraSPARC T2 17

2.2.3.3 IBM Cell Broadband Engine 19

2.2.3.4 Nvidia Fermi 24

2.3 Multicore Programming Tools 26

2.3.1 Parallel Libraries 27

2.3.1.1 Shared Address Space 28

2.3.1.2 Distributed Address Space 29

2.3.1.3 Stream Processing 30

2.3.2 Parallel Languages 31

2.3.2.1 Shared Address Space 32

2.3.2.2 Partitioned Global Address Space 32

2.3.3 Parallelizing Compilers 34

2.4 Multicore System Software 34

2.4.1 Shared Memory OS 35

2.4.2 Multikernel OS 37

2.4.3 Virtualization 39

CHAPTER 3: DESIGNING CACHE- AND SPACE-EFFICIENT DATA PARALLEL ALGORITHMS FOR MULTICORES 40

3.1 Introduction 40

3.2 Background 43

3.2.1 Computational Models 43

3.2.2 Cache-oblivious Model 45

3.2.3 Multicore Schedulers 47

3.3 Parallel Cache-oblivious Design Methodology 48

3.3.1 Computational Model 48

3.3.2 Recursive Geometric Decomposition 50

3.3.3 Red-Blue Pebble Game 51

3.3.4 Nominal Parallel Pebbling Strategy 53

3.3.5 Weighted-vertex Parallel Pebbling Strategy 54

3.3.6 Data-aware Scheduling 56

3.4 Case Studies 58

3.4.1 Matrix Multiplication 59

3.4.2 Finite Difference Time Domain 64

3.5 Conclusion 69

CHAPTER 4: INTEGRATED DATA PREFETCHING AND CACHING IN MULTICORES 71

4.1 Introduction 71

4.2 Background 72

4.3 Computation and Data Transfer Parallelism in the IBM Cell/B.E. 73

4.4 Machine Model and General Bounds 75

4.5 Matrix Multiplication 77

4.5.1 Theoretical Bounds 77

4.5.2 Discussion 86

4.6 Finite Difference Time Domain (FDTD) 88

4.6.1 Theoretical Bounds 88

4.6.2 Discussion 94

4.7 Conclusion 96

CHAPTER 5: EXPERIMENTAL STUDIES IN COMPUTING ON COMMERCIAL MULTICORE PROCESSORS 98

5.1 Introduction 98

5.2 Experimental Systems 99

5.3 In-core Optimization Techniques 102

5.3.1 Matrix-Vector Multiplication 103

5.3.2 Data Transformation: Data Layout Scheme 104

5.3.3 Loop Transformation: Loop Blocking (Loop Tiling) 108

5.3.4 Loop Transformation: Loop Unrolling 112

5.3.5 Loop Transformation: Loop Interchange (Computational Reordering) 113

5.3.6 Vectorization 115

5.4 Case Studies: Experimental Results and Performance Analysis 120

5.4.1 Dense Matrix Multiplication (DMM) 120

5.4.1.1 Multicore-efficient Implementation 122

5.4.1.2 Optimization at Register Level 126

5.4.1.3 Performance Analysis 130

5.4.2 Finite Difference Time Domain (FDTD) 136

5.4.2.1 Multicore-efficient Implementations 138

5.4.2.2 Optimization at Register Level 141

5.4.2.3 Performance analysis 144

5.4.3 LU Decomposition 148

5.4.3.1 Multicore-efficient Implementations 150

5.4.3.2 Optimization at Register Level 155

5.4.3.3 Performance Analysis 155

5.4.4 Power Flow Solver based on Gauss-Seidel method (PFS-GS) 158

5.4.4.1 Multicore-efficient Implementations 161

5.4.4.2 Optimization at Register Level 163

5.4.4.3 Performance Analysis 167

5.5 Conclusion 170

CHAPTER 6: CONCLUSION AND FUTURE WORK 173

6.1 Conclusion 173

6.2 Future work 175

REFERENCES 177

# LIST OF FIGURES

FIGURE 1.1: Design flow for multicore-efficient software design. 4

FIGURE 2.1: Organization of the Nehalem processors. 15

FIGURE 2.2: Organization of the UltraSPARC T2 processor. 17

FIGURE 2.3: Organization of the Cell Broadband Engine. 19

FIGURE 2.4: Organization of the Nvidia Fermi. 25

FIGURE 2.5: Multikernel model. 37

FIGURE 2.6: Organization of Barrelfish. 38

FIGURE 2.7: A simplified representation of the virtualized software stack, demonstrating the deployment of a hypervisor and several VMs, each of which is managing a subset of the cores and a subset of the processes. 39

FIGURE 3.1: The cache hierarchy of the Intel quad-core Clovertown processor. 50

FIGURE 3.2: A 2-level geometric decomposition of $A$, $B$, and $C$ matrices. 60

FIGURE 3.3: Illustrative level-($i+1$) and level-$i$ DAGs for matrix multiplication. 60

FIGURE 3.4: The weighted-vertex pebble game: (a) Initial vertex weight assignment for level-($i+1$) DAGs of Figure 3.2 under $\Omega_s$; (b) An intermediate step in the pebbling of the level-($i+1$) DAGs under $\Omega_s$. 60

FIGURE 3.5: A 2-level geometric decomposition of the **E**- and **H**-field cubes. 65

FIGURE 3.6: DAGs for FDTD: Note that there are $6$ DAGs corresponding to $E_x$, $E_y$, $E_z$ and $H_x$, $H_y$, $H_z$. 65

FIGURE 4.1: Simultaneous computing and DMA transfer: (a) Execution sequence for single read buffering; (b) Execution sequence for double read buffering. 74

FIGURE 4.2: Matrix multiplication with $3 \times 3$ blocks. 77

FIGURE 4.3: Simultaneous computing and data transfer for single buffer each for matrix $A$ and $B$: (a) Execution sequence obtained by considering no reuse of the data present in the local memory; (b) Execution sequence obtained by considering reuse of the data present in the local memory. 79

FIGURE 4.4: Simultaneous computing and data transfer for single buffer for matrix *A* and double buffer for matrix *B* obtained by considering reuse of the data present in the local memory: (a) Data transfer bound case; (b) Compute bound case. 83

FIGURE 4.5: Simultaneous computing and data transfer for double buffer for both matrix *A* and *B* obtained by considering reuse of the data present in the local memory: (a) Data transfer bound case; (b) Compute bound case. 85

FIGURE 4.6: Theoretical lower bounds for matrix multiplication on IBM Cell/B.E. 88

FIGURE 4.7: Simultaneous computing and data transfer for single buffer for **E**-field computation: (a) Execution sequence obtained by considering no reuse of the data present in the local memory. This scheme requires the storage of *4* blocks of data in the local memory; (b) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$ and $E_y$, $E_z$. This scheme requires the storage of *4* blocks of data in the local memory; (c) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$, and $E_z$. This scheme requires the storage of *5* blocks of data in the local memory; (d) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$, and $E_z$ but with all the data fetched initially. This scheme requires the storage of *9* blocks of data in the local memory. 91

FIGURE 4.8: Simultaneous computing and data transfer for double buffers for E-field computation as data transfer bound cases: (a) Execution sequence obtained by considering no reuse of the data present in the local memory. This scheme requires the storage of *4* blocks of data in the local memory; (b) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$, and $E_z$ but with all the data fetched initially. This scheme requires the storage of 9 blocks of data in the local memory. 92

FIGURE 4.9: Theoretical lower bounds for FDTD on IBM Cell/B.E. 96

FIGURE 5.1: Dell Precision 690 with dual Intel quad-core Xeon E5345. 99

FIGURE 5.2: SONY PlayStation3 with one PPE and eight SPEs. 100

FIGURE 5.3: The PPE-centric programming models. 101

FIGURE 5.4: Matrix-vector multiplication with $n=4$ and $m=4$. 104

FIGURE 5.5: Data Layout Schemes of $4\times4$ Matrix: (a) Row-major order; (b) Column-major order; (c) Space-filling-curve order; (d) Z-Morton order. 104

FIGURE 5.6: Performance of data layout schemes for matrix-vector multiplication with fixed $m = 1024$: (a) The performance in seconds; (b) The performance in MFLOPS. 106

FIGURE 5.7: Implementation of loop blocking algorithm in row-major layout scheme with $n=4$, $m=4$ and $2\times2$ blocks. 109

FIGURE 5.8: Memory access pattern of matrix-vector multiplication with $n=4$ and $m=4$. 110

FIGURE 5.9: The data dependency the standard matrix multiplication with $n\times n$ square matrices. 121

FIGURE 5.10: The example of the scheduling schemes on the weighted DAGs at register level blocking with $b_0=2$; Note, the number on right side of each computational vertex represents the sequential scheduling order. 127

FIGURE 5.11: The example of vector computations for two multiplications following by two addition operations simultaneously: (a) Based on 1DF or 1BF scheduling scheme; (b) Based on hybrid scheduling scheme. 128

FIGURE 5.12: Vectorization implementation at register level blocking with $b_0$=4 using hybrid scheduling scheme and Intel x86_64 SSE2 intrinsics for Intel Clovertown platform. 129

FIGURE 5.13: Overall performance on Intel Clovertown platform: (a) Performance in GFLOPS per core; (b) Execution time in seconds on single core. 133

FIGURE 5.14: Overall performance on IBM Cell/B.E. platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on single SPE. 135

FIGURE 5.15: Example of data dependency in space domain for a cell of $E_x$ computation. 138

FIGURE 5.16: An example distribution of threads among four cores: (a) Data partitioning scheme for the naïve parallel algorithm; (b) Mapping threads to cores for both (a) and (c) data partitioning schemes; (c) Data partitioning scheme for the multicore efficient algorithm. 139

FIGURE 5.17: The hybrid scheduling scheme for the four $E_x$ computations for the register level blocking; the number on the right side of each computational vertices indicates the SIMDize scheduling order. 141

FIGURE 5.18: The example of the conflict alignment of 128-bit vector registers for $H_y(i,j,k)$ and $H_y(i,j,k-1)$. 142

FIGURE 5.19: Overall performance on Intel platform: (a) Performance in GFLOPS per core; (b) Execution time in seconds on a single core. 146

FIGURE 5.20: Overall performance on IBM platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on a single SPE. 148

FIGURE 5.21: The data dependency of the LU decomposition based on Gaussian elimination method. 150

FIGURE 5.22: The block partition with the four different types in sub-matrix $A^k$ at d-level. 153

FIGURE 5.23: The example of data dependency of LU decomposition for a matrix A with 4×4 blocks. 154

FIGURE 5.24: Overall performance on Intel platform: (a) Performance in GFLOPS per core; (b) Execution times in seconds on a single core. 156

FIGURE 5.25: Overall performance on IBM Cell BE platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on a single SPE. 157

FIGURE 5.26: The sample power network computation with 5 buses and 5 branches. 159

FIGURE 5.27: *Vectorized Unified-Bus-Computation* Module. 164

FIGURE 5.28: Overall performance on Intel Clovertown platform: (a) Performance in GFLOPS per core; (b) Execution time in seconds on a single core. 167

FIGURE 5.29: Overall performance on IBM Cell/B.E. platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on a single SPE. 169

# LIST OF TABLES


TABLE 5.1: Pseudo code of the matrix-vector multiplication for different data layout schemes. 105

TABLE 5.2: The memory access pattern obtained by following a row-major computational order in the nested loop with the 4×4 matrix *A* laid out in a row-major layout scheme. 107

TABLE 5.3: The memory access pattern obtained by following a row-major computational order in the nested loop with the 4×4 matrix *A* laid out in a column-major order layout scheme. 107

TABLE 5.4: The memory access pattern obtained by following a row-major computational order in the nested loop with the 4×4 matrix *A* laid out in a space-filling curve order layout scheme. 108

TABLE 5.5: An example of matrix vector multiplication with $m \times n$ matrix *A* using loop blocking ($c = A \times b$). 109

TABLE 5.6: The performance of loop blocking algorithm with varying block size. 111

TABLE 5.7: An example of matrix-vector multiplication with $m \times n$ matrix *A* using loop unrolling ($c = A \times b$). 112

TABLE 5.8: The performance of loop unrolling algorithm with varying unrolling factor. 113

TABLE 5.9: Examples of matrix-vector multiplication with $m \times n$ matrix *A* using loop interchange ($c = A \times b$). 113

TABLE 5.10: The memory access pattern for the loop interchange algorithm with 4×4 matrix *A* in column-major order layout scheme shown in Table 5.9 (b). 114

TABLE 5.11: The performance of loop interchange algorithm with varying problem size *n*. 115

TABLE 5.12: The example of matrix-vector multiplication with $m \times n$ matrix *A* using vectorization ($c = A \times b$). 117

TABLE 5.13: The performance of vectorization algorithms with a varying problem size n and a fixed $m=1024$. 119

TABLE 5.14: The conventional serial algorithm for multiplying of two $n \times n$ square matrices. 121

TABLE 5.15: The summary of our implementation techniques of matrix multiplication used for our platforms. 126

TABLE 5.16: The performance (GFLOPS) for varying schedules and sizes of L1-block ($b_1$) with fixed size of L2-block ($b_2$=512) and register-block ($b_0$=4) on a single core of Intel Clovertown platform. We use 1DF scheduling scheme for L1-level and L2-level blocking, and vary the scheduling scheme at the register level. 130

TABLE 5.17: Cache miss rate (%) and system bus bandwidth utilization (%) on Intel Clovertown platform. 132

TABLE 5.18: The performance (GFLOPS) for different multi-buffering schemes and size of LS-block ($b_1$) with fixed size of the register-block ($b_0$=4) on a single SPE of IBM Cell/B.E. platform. We use the 1DF scheduling scheme for both level blocking. 134

TABLE 5.19: The naïve serial 3D-FDTD algorithm. 138

TABLE 5.20: The summary of our implementation techniques of 3D FDTD for our platforms. 140

TABLE 5.21: The pseudo code for the SPE 3D-FDTD **E**-field computation using double buffers. 143

TABLE 5.22: The performance (GFLOPS) for different register level schedules and sizes of L1-block ($b_1$) with fixed size of L2-block ($b_2$=64) and register-block ($b_0$=4) on a single core of the Intel Clovertown platform. We use 1DF scheduling scheme for L1-level and L2-level blocking, and vary the scheduling scheme at the register level. 144

TABLE 5.23: Cache miss rate (%) and system bus bandwidth utilization (%) on Intel Clovertown platform. 145

TABLE 5.24: The LU decomposition based on Gaussian elimination method with $n \times n$ square matrix $A$. 149

TABLE 5.25: The summary of our implementation techniques of LU Decomposition for our platforms. 151

TABLE 5.26: The different tasks of the four blocks at *d*-level. 153

TABLE 5.27: Pseudo-code of naïve serial algorithm for the bus and branch computations. 160

TABLE 5.28: The summary of our implementation techniques of PFS-GS for our platforms. 163

TABLE 5.29: Pseudo code of the multicore-efficient implementation for PFS- GS on the IBM Cell/B.E. platform. 166

TABLE 5.30: GFLOPS with varying DMA transfer size in bytes on single SPE. 168

TABLE 5.31: Distributed speedup and % of computation on single SPE; Note our multicore-efficient implementation combines both double-buffering scheme and vectorized unified-bus-computation module. 169

# CHAPTER 1: INTRODUCTION

## 1.1 Rise of Multicore Computing

Since the introduction of the microprocessor in the mid-70s the computer industry has pursued a uniprocessor hardware architecture paradigm accompanied by a sequential programming model. The steady growth in performance over the years was achieved primarily through a steady increase of clock frequency enabled by scaling of the underlying transistors. Architectural innovations such as hardware controlled on-chip memory hierarchies (caches) were introduced so that the increasing gap between the processor speeds and the memory access latencies could be hidden from the programmer. At the chip level, application parallelism was primarily exploited at the instruction level in a manner transparent to the programmer through multiple execution pipelines and out-of-order processing controlled by complex logic.

However, by the middle of this decade, the traditional uniprocessor architecture performance had hit a roadblock due to a combination of factors, such as excessive power dissipation due to high operating frequencies, growing memory access latencies, diminishing returns on deeper instruction pipelines, and a saturation of available instruction level parallelism in applications. An attractive and viable alternative to improve performance are multicore processors where multiple processor cores, interconnects, and both shared and private caches are integrated on a single chip. The individual cores are often simpler than uniprocessor counterparts, exploit instruction level

parallelism adequately, and typically achieve better performance-power figures. Moreover, multicore architectures allow the programmer to exploit multiple levels of parallelism at the data and task level than was possible with a traditional uniprocessor. From a modest beginning of dual and quad cores, multicore processors are expected to include hundreds of cores in a single chip in the near future. Currently almost all of the high performance processors offered by leading industry vendors such as Intel, IBM, AMD and Sun subscribe to the multicore paradigm.

## 1.2 Research Goals

As discussed in the previous section, programmers can no longer expect continued increases in software performance without developing parallel, memory hierarchy friendly software that can effectively exploit the chip level multiprocessing paradigm of multicores. Further, due to power issues favoring architectures with lower clock frequencies and simpler in-order processing cores, the single threaded performance of commercial multicores may actually suffer in the coming years. Unfortunately, there is no easy solution to this problem. In many cases, serial code cannot be parallelized without investing considerable time and effort. Also, existing parallel libraries are often not designed to exploit the on-chip shared memory hierarchy characteristic of multicore processors. Piecemeal solutions developed for specific architectures run the risk of being non-portable not only across different architectures, but also across future versions of the same architecture. Considerable effort continues to be made in developing tools that seek to generate parallel code starting from a serial code base with minimal effort. Although this approach has its merits in terms of short term productivity, we argue in this dissertation that over the long term, a systematic design process that starts from the

sequential algorithm of the problem and develops a scalable, parallel, memory hierarchy friendly algorithm with tunable parameters has the best chance of avoiding technology obsolescence. Note that the choice of an appropriate machine model is an important element of this approach. *The goal of this dissertation is to demonstrate a design process for data parallel problems that starts with a sequential algorithm and ends with a high performance implementation on a multicore platform.* The dissertation focuses on data parallel algorithms since they are the basis of several scientific computing kernels where high performance is critical.

While elements of the proposed design process have been reported previously, the focus has tended to be either on theoretical algorithm analysis or on code engineering, limiting its utility to programmers. The focus of our work is to provide the programmer with a design process that integrates algorithm development with actual implementation on commercial multicores. We identify and integrate recently reported research results into this design process and innovate where necessary. The flowchart shown in Figure 1.1 summarizes the multicore-efficient software design process proposed in this dissertation.

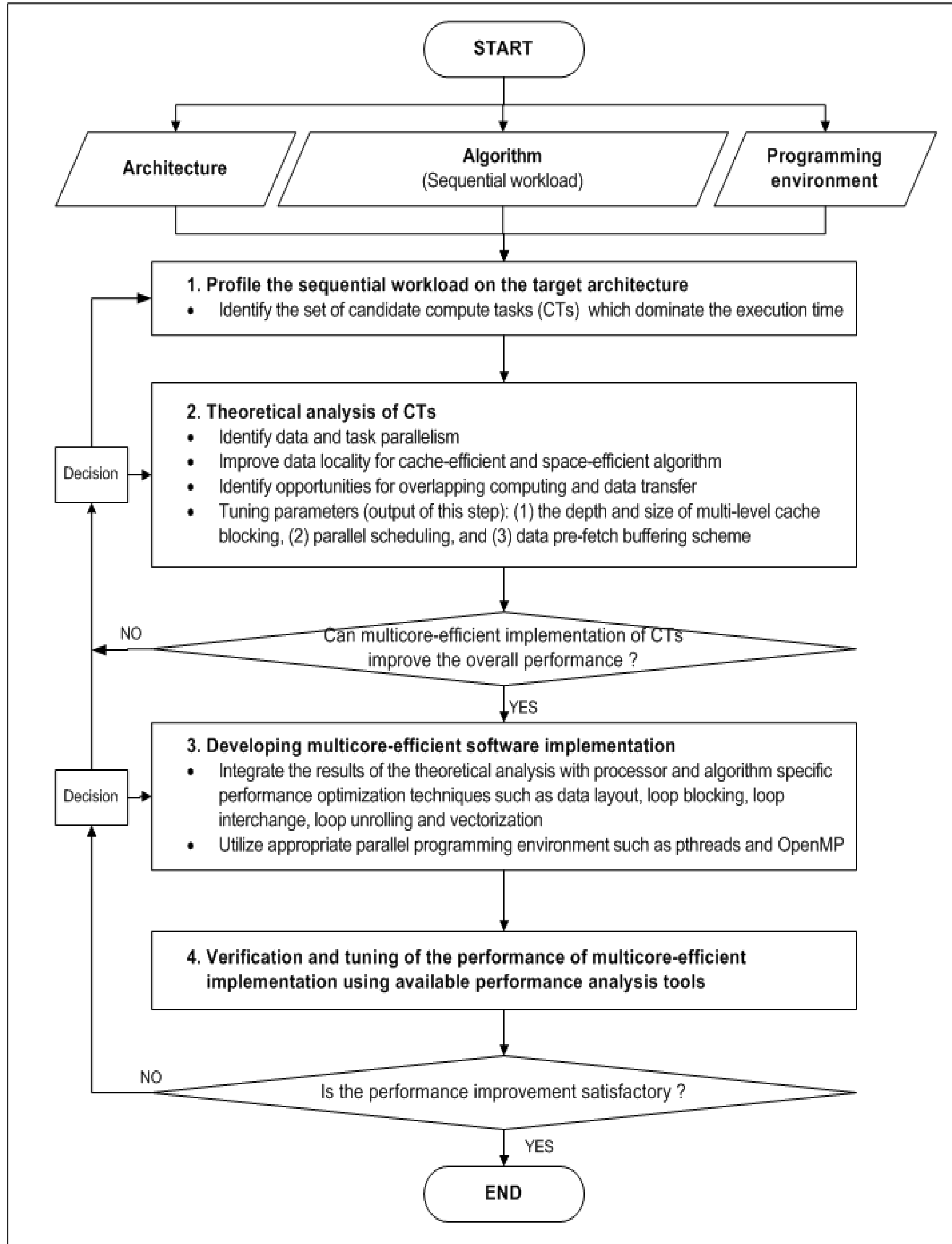



FIGURE 1.1: Design flow for multicore-efficient software design.

The inputs to the design process shown in Figure 1.1 are the target multicore platform, the parallel programming tools available for the multicore platform, and the sequential workload for the problem. The design process consists of four major steps -

We first profile the sequential workload on the target architecture to determine the set of candidate compute tasks whose multicore efficient implementation would improve the overall performance of the workload. This step uses available statistical profilers such as GNU's gprof, Oprofile, Intel Vtune Performance Analyzer, and Sun CoolTool. Note that in this dissertation we do not explicitly demonstrate this step but assume that the candidate compute kernels are known.

The second step theoretically analyzes the candidate tasks with the goals of identifying data and task parallelism, improving data locality for cache-efficient and space-efficient implementation, and identifying opportunities for overlapping computing and data transfer. The theoretical analysis is based on an appropriate machine model for the target multicore platform. The primary outputs of this step are parallel schedules for computation at different levels of the memory hierarchy, theoretical bounds on execution time under these schedules, and candidate tuning parameters.

The third step focuses on developing multicore-efficient software implementations of the candidate tasks by integrating the results of the theoretical analysis with processor specific performance optimization techniques utilizing appropriate parallel programming tools. While this step is platform and programming environment specific, many of the optimization techniques are portable across different multicore platforms.

The fourth step involves verification and tuning of the performance of multicore-efficient implementation using available performance analysis tools, such as Intel Performance Analyzer, Intel ThreadChecker, Intel ThreadProfiler, Sun Microsystems CoolTool, Sun Microsystems Thread Analyzer and IBM Cell B.E. SDK (see Chapter 2 and 5 for more details). These tools help in monitoring parallel performance including parallel overhead, synchronization and load-balance. The programmers can attempt to optimize the run-time performance by varying the tuning parameters.

The steps outline above may have to be repeated iteratively until the desired performance is achieved.

## 1.3 Dissertation Contributions

The goal of our research is to help programmers analyze and improve the performance of data parallel applications on multicore architectures. The major contributions of this dissertation are:

(1) We present a novel weighted-vertex pebble strategy for determining efficient block size to improve data locality on multicores. The weighted-vertex pebble strategy is an extended pebble game for devising space-efficient and cache-efficient algorithm through maximal data sharing between concurrent tasks under a given scheduling strategy.

(2) We describe an innovative data prefetching and caching strategy to determine the optimal multi-buffering scheme for compute bound and data transfer bound algorithms. The integrated data prefetching and caching strategy improves the performance by overlapping between computations and data transfers while simultaneously effectively exploiting data locality.

(3) We illustrate a muliticore efficient design process that blends theoretical results with practical performance optimization techniques on commercial multicores. Specifically, we integrate our theoretical results with a series of in-core optimizations to develop a robust set of design techniques that scale well both with the problem size and the number of cores on a variety of multicore architectures.

(4) We develop multicore efficient high performance computing kernels for several important scientific computing algorithms such as matrix multiplication, finite difference time domain, LU decomposition and power flow solver based on Gauss-Seidel method. These highly optimized, multithreaded libraries could be used in science and engineering applications that require maximum performance.

## 1.4 Dissertation Outline

This dissertation describes effective data parallel computing on multicore platforms motivated by our experiences working with commercial multicore platforms. The dissertation begins with an overview of trends multicore computing in Chapter 2. We focus on the important developments in multicore architecture, programming tools and system software.

Chapter 3 presents a design methodology that aids in the development of parallel cache-efficient and space-efficient algorithms for shared cache multicore processors. The methodology uses a weighted vertex pebbling game for maximal data sharing between concurrent tasks under a given scheduling strategy at each level of the memory hierarchy.

Chapter 4 presents algorithm specific integrated software caching and pre-fetching strategies. We introduce a general purpose machine model and present

conditions for when the total execution time is compute bound or data transfer bound. Through case studies we illustrate the choice of optimal buffering strategy when both pre-fetching and caching is considered.

Chapter 5 describes the multicore-efficient implementations of data parallel algorithms on commercial multicore platforms. In this chapter we highlight the synthesis of the theoretical results of Chapters 3 and 4 with practical in-core optimization techniques to derive scalable multicore efficient implementations of some of the widely used scientific computing kernels. Extensive measurement results are presented on the Intel Clovertown and IBM Cell/B.E. platforms.

Chapter 6 concludes the dissertation and provides directions for future work on programming of emerging multicore architectures.

# CHAPTER 2: TRENDS IN MULTICORE COMPUTING

## 2.1 Introduction

The rise of chip multiprocessing or the integration of multiple general purpose processing cores on a single chip (multicores), has impacted all computing platforms including high performance, servers, desktops, mobile, and embedded processors. As discussed in Chapter 1, the introduction of parallel computing at the chip level was motivated by the need to deliver Moore's law type advances in computing performance within an acceptable power budget. With this paradigm shift in computing still its early years, open questions remain on architecturally the best way to achieve this objective. Moreover, a large part of the performance of multicores hinges on the performance of parallel software that runs on them. Unfortunately, despite the progress made in developing parallel algorithms and software in the past two decades, the considerable challenges remain in its widespread adoption to the entire software stack.

Traditionally, parallel computing was largely confined to scientific computing where either custom made supercomputers or clusters of general purpose computers were employed. The parallel code necessary for these platforms were developed by application domain specialists. The rise of internet led to the development of data centers with clusters consisting of thousands of computing nodes and terabytes of storage. In the past few years, the rising costs in maintaining these data centers as well as the availability of broadband connections, has led to the emergence of "cloud computing" where both

computing resources and software are available on the "cloud" as a service [81]. However, the parallel code running on these platforms is largely web based applications characterized by embarrassing amounts of parallelism.

We note that the successful adoption of multicore processors for general purpose, scientific, and embedded computing will depend on jointly developing both the processor architecture and the software stack necessary for code developers to efficiently exploit the many types of parallelism that may exist in a computing problem. In this chapter, we review the state-of-the-art in multicore architectures (Section 2.2), parallel programming languages and tools (Section 2.3), and system software (Section 2.4). We pay special attention to the underlying trends that portend developments in each of these areas in the next few years.

## 2.2 Multicore Architectures

In this section we examine the architectures of popular commercial multicore processors. While a plethora of such architectures exists in the embedded domain, we limit ourselves to high performance multicores where power dissipation is an important but not a dominating design issue.

### 2.2.1 Historical Trends

The architectures of today's multicore processors are based on the uniprocessor and the shared addressed space and message passing parallel architecture designs from the past two decades. Uniprocessors have evolved from a simple RISC based pipeline of the eighties to the superscalar, RISC-CISC architectures with deep execution pipelines and out-of-order execution. Also, the increasing gap between processor and the external memory latencies requires the use of deep on-chip cache hierarchies for good memory

performance. The architectural goal of these processors was to exploit as much single thread performance as possible through aggressive exploitation of Instruction Level Parallelism (ILP). Considerable logic and power budget was devoted to dynamically finding and scheduling instructions to maximally utilize the pipelines. However, the diminishing returns on the power-performance of this approach limited the continued pursuit of performance solely through ILP.

Parallel machines evolved from the Cray vector machines implementing the Single Instruction Multiple Data (SIMD) paradigm to commodity processors connected by Commercial-Off-The-Shelf (COTS) network implementing the Single Program Multiple Data (SPMD) paradigm. Parallel machine organization can be classified into two main types – a) Shared Memory Processors (SMP) where all the processors share a common memory address space and b) Message Passing Processors (MPP) where the memory address space is disjoint and explicit messages are sent between the processors. Commercial SMPs typically employ a bus based interconnect and provide hardware cache coherence. Bus contention and the difficulties in scaling the cache coherence protocols limit the number of processors to around 32. MPPs employ point-to-point COTS network such as Ethernet (Beowulf cluster) or specialized network (IBM Blue Gene/L). The disjoint address space, lack of hardware memory coherence, and the use of scalable interconnect allows for MPPs with hundreds of processors. A great majority of the TOP500 supercomputers are MPPs.

As will be seen in the next couple of sub-sections chip level multiprocessing has borrowed a number of ideas from the above described sequential and parallel computer architectures.

### 2.2.2 Architectural Elements

Architecturally multicore processors can be classified on the basis of a) the processing elements, b) the memory system, and c) interconnect.

*Processing Elements:*

High performance multicore processors today adopt a mix of two design extremes – for a given transistor budget integrate a small number of complex superscalar, super-pipelined cores with out-of-order processing (example Intel Xeon Clovertown quadcore), or a large number of simple in-order cores (example Sun UltraSPARC T1). The complex cores are geared towards applications requiring good single thread performance while the large number of simple cores target applications with abundant thread level parallelism. Interestingly in multicores with simple cores, the operating frequency is far below the maximum allowed by the process technology so as to manage the power budget. In either case, Symmetric Multi-Threaded (SMT) cores are utilized to manage the memory latency. While most high performance multicores have homogenous cores, processors with heterogeneous cores specialized for different application domains have also made their appearance. The Instruction Set Architecture (ISA) of the cores is an extension of the ISA of the corresponding unicores (example x86, SPARC, Power) with additional instructions such as atomic operations to support synchronization. The use of legacy ISAs allows the execution of the existing software without recompilation.

*Memory System:*

Most of the high performance multicore processors in the market today follow the shared address space architecture described in Section 2.2.1. However, shared memory multicore processors differ from traditional SMPs in the following three significant ways-

(1) The processing cores, the interconnect, and a part of the shared memory hierarchy are on the same chip/module resulting in potentially lower communication and synchronization costs.

(2) The shared memory (typically the L2 or L3 cache) is not only shared by all or a subset of the processing cores but is of a limited size.

(3) The integration of the processing cores, the interconnect, and the cache hierarchy on a single chip necessitates micro-architectural tradeoffs between the performance, die area, and power budgeted to the different components.

Hardware support for cache coherence is typically provided following either the broadcast (ordered interconnect) or directory based protocols (ordered and unordered interconnects). In principle the concept of a shared addressed space makes programming simple. However, in practice, since shared memory does not provide implicit synchronization of parallel tasks, memory consistency models and synchronizations routines are needed to provide the necessary synchronization. The complex interaction of synchronization, coherence, and consistency has the potential to complicate the programming and also limit the core scaling in these processors. An example of a commercial processor using the shared memory paradigm is the Intel Xeon Clovertown quad-core processor (see Section 2.2.3).

Message passing multicores have also made their appearance commercially. Here the processing cores are cache-less but instead has software managed local memory. Messages are passed between the cores on high speed on chip interconnect through Direct Memory Access (DMA) type operation. The sending and receiving of the messages implicitly synchronizes the processors. While a better hardware

power/performance is possible with this approach, the machines are harder to program. An example of a commercial processor using the shared memory paradigm is the IBM Cell Broadband Engine Processor (See Section 2.2.3).

Taking advantage of the ample on-chip bandwidth in multicores, a new protocol known as Transactional memory Coherence and Consistency (TCC) has been introduced. TCC is an extension of the shared address space paradigm, where, instead of load/store operations, atomic transactions are the basic unit of parallel work, communication, coherence, and consistency [36]. As described by Hammond et. al. "TCC hardware combines all writes from each transaction region in a program into a single packet and broadcasts this packet to the permanent shared memory state atomically as a large block. This simplifies the coherence hardware because it reduces the need for small, low-latency messages and completely eliminates the need for conventional broadcast cache coherence protocols, as multiple speculatively written versions of a cache line may safely coexist within the system. Meanwhile, automatic, hardware-controlled rollback of speculative transactions resolves any correctness violations that may occur when several processors attempt to read and write the same data simultaneously. The cost of this simplified scheme is higher interprocessor bandwidth". A commercial processor incorporating TCC is the Sun Microsystems' Rock multicore (see Section 2.2.3).

*Interconnect:*

The on-chip interconnect found in today's commercial high performance multicores include point-to-point, ring, bus, and crossbar. Bus has the simplest design and has global ordering that supports broadcast cache coherence protocols. However, buses do not scale well. Crossbar is unordered and offers low latency but does not scale well

either. Point-to-point interconnect (such as Intel QPI and AMD Hyper transport) has good performance but scales poorly.

### 2.2.3 Case Studies

In this section, we briefly describe the architectural features of the state-of-the-art high performance multicores available in the market today. The multicore processors presented illustrate the different design elements described in the previous sections.

#### 2.2.3.1 Intel Gainestown

Intel Gainestown (Xeon W5500 series) was released in November 2008 by Intel based on the Nehalem microarchitecture and is currently manufactured in a 45 nm process. Nehalem is based on a multicore design philosophy of integrating a modest number of homogeneous complex cores with good single thread performance. Figure 2.1 shows the organization of the Nehalem processors.

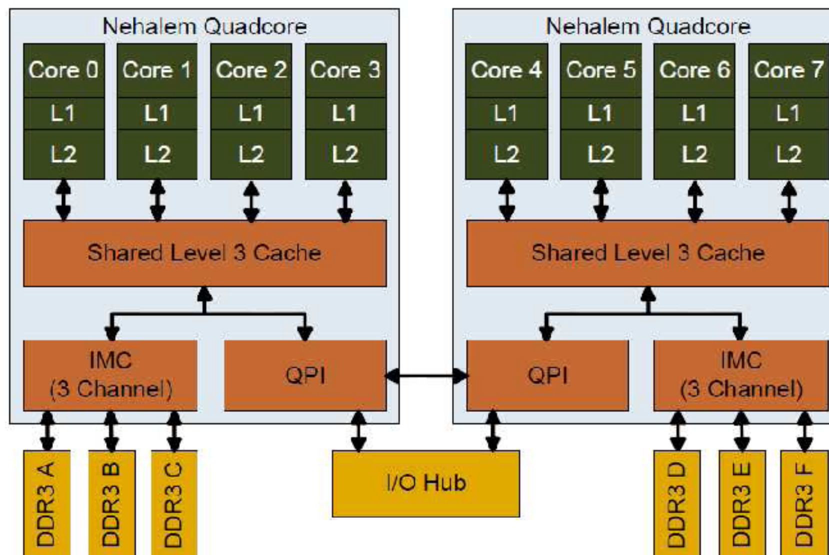



FIGURE 2.1: Organization of the Nehalem processors.

*Processor Cores:*

Gainestown is a true quad core processor with an operation frequency of up to 3.2 GHz consuming about 130W. The x86 based cores are out-of-order and is Simultaneously Multi Threaded (SMT) supporting two threads per core. Each core can issue 4-double precision floating point operations per clock. The cores incorporate Intel's Turbo Boost Technology which allows active processor cores to run faster when there is available headroom with power, temperature, and temperature specification limits. Gainestown also incorporates Application Targeted Accelerators (ATA) which are low latency, low power, and fixed function accelerators on the processor die targeted at specific applications. The seven ATAs target string and text processing operations. Integrated power gates allow the individual idling cores to be reduced to near-zero power independent of other cores, reducing the idle power consumption to 10 W.

*Memory System:*

Gainestown has a three level on-chip cache hierarchy with private 64 KB L1 cache (32 KB data + 32 KB instruction), 256 KB L2 cache, and an 8 MB L3 cache shared by all cores [82]. A 512 entry second level TLB is included to improve performance. The Nehalem implements a cache coherent Non Uniform Memory Architecture (ccNUMA) with a broadcast based MESIF cache coherence protocol [53]. The MESIF protocol extends the MESI protocol with a "Forwarding" state that allows unmodified data shared by two processors to be forwarded to a third processor. Programmers must consider the NUMA nature of the architecture in accessing data from a remote socket compared to a local DRAM.

*Interconnect:*

The Nehalem micro-architecture uses a point-to-point interconnect that uses the Intel QuickPath Technology. The interconnect uses up to 6.4 Giga transfers/second links, delivering up to 25 GB/s of total read bandwidth per core. Each processor integrates a triple channel integrated memory controller with a peak bandwidth of 32 GB/s with DDR3-1333 DIMMs.

2.2.3.2 Sun UltraSPARC T2

The Sun UltraSPARC T2 was released in 2007 by Sun Microsystems based on the UltraSPARC architecture and the SPARC ISA. The UltraSPARC T2 is currently manufactured in a 65 nm process. UltraSPARC micro-architecture is based on a multicore design philosophy of integrating a large number of homogeneous simple highly multithreaded cores targeting application task level parallelism. Figure 2.2 shows the organization of the UltraSPARC T2 processor.

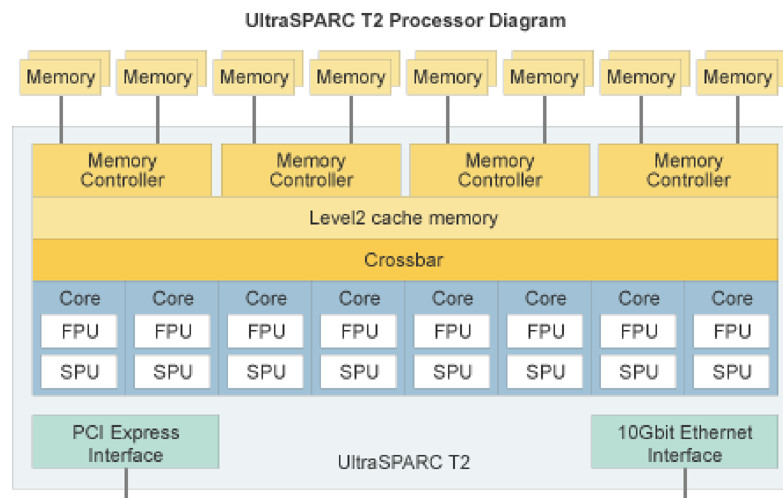



FIGURE 2.2: Organization of the UltraSPARC T2 processor.

*Processor Cores:*

The UltraSPARC T2 is an 8 core processor each with full hardware support for executing 8 independent threads. The in-order cores run at an operating frequency of up to 1.4 GHz with a total power consumption of about 84W. Each core consists of two integer execution units, a floating point and graphics unit, and a cryptographic stream processing unit [63]. UltraSPARC T2 implements a fine-grained multi-threading scheme where the threads are switched on a cycle-by-cycle basis between the available threads within the two statically partitioned thread groups of 4 threads each. When a thread encounters a cache-miss it is made unavailable and the instructions from it are not issued. In each cycle two instructions can be issued from each thread group. UltraSPARC T2 seeks to minimize power consumption through limited execution speculation, control and data-path clock gating, and through external power throttling.

*Memory System:*

The UltraSPARC T2 has a two level on-chip cache hierarchy with a private L1 cache and a shared L2 cache. The 4 MB L2 cache is 16-way set associative with a line size of 64 bytes and organized as 8 banks. The L1 data cache is 8 KB and the instruction cache is 16 KB. The L1 caches are write through, with allocate on loads and no-allocate on stores. The L2 cache maintains a directory of L1 tags. The directory maintains a shares list at the level of L1 line granularity. Local caches are not update by stores till the L2 is updated. However, in the meantime, the same thread can see its stores.

*Interconnect:*

The UltraSPARC T2 uses non-blocking pipelined crossbars interconnect that connects the 8 cores to the 8 banks and the I/O port. The crossbar has a total write

bandwidth of 90 GB/s and a read bandwidth of 180 GB/s. The L2 cache connects to a 4 on-chip memory controllers interfacing to FBDIMM channels. The peak memory bandwidth is 50 GB/s for read and 26 GB/s for writes. The crossbar establishes memory order between transactions from the same and different L2 banks.

2.2.3.3 IBM Cell Broadband Engine

The Cell Broadband Engine introduced by IBM in 2006 is a heterogeneous multicore processor initially targeted for game consoles and consumer media applications. The processor is currently manufactured in a 45 nm technology. Figure 2.3 shows the organization of the Cell Broadband Engine.

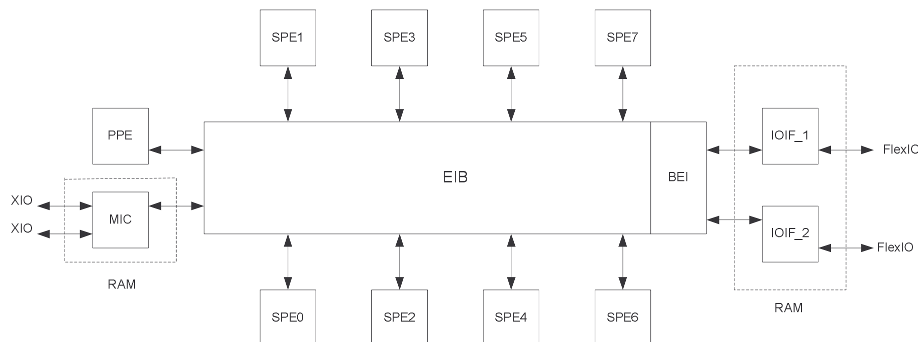



FIGURE 2.3: Organization of the Cell Broadband Engine.

The Cell processor consists of a Power Processor Element (PPE) and 8 identical Synergistic Processor Elements (SPE). The PPE contains a 64 bit PowerPC architecture core and is primarily intended for control processing, running operating systems, managing system resources and running SPE threads. The SPE is a vector processor supporting a specialized SIMD instruction set architecture for compute intensive operations. An important difference between the SPE and the PPE is in the way memory

is accessed. The PPE uses load and store instructions to transfer instructions and data from the main memory to the register files using a two level cache hierarchy. The SPE uses Direct Memory Access (DMA) to transfer data from the main memory to a private Local Store (LS) memory through the high speed Element Interconnect Bus (EIB). Note that the SPE and PPE have two distinct ISAs necessitating the use of two different compilers. A more detailed description of the different units of the Cell processor emphasizing the different levels of parallelism supported by each is given below.

*Power Processing Element:*

The PPE is a Power ISA based dual issue, in-order execution design, 2-way Symmetric Multi-Threaded (SMT) processor with the design optimized for frequency and power efficiency [42]. The two simultaneous threads of execution give software the effective appearance of two independent processing units with shared data flow. The PPE cache hierarchy consists of a 32 KB L1 data cache, a 32 KB L1 instruction cache, and 512 KB unified L2 cache. The second-level cache and the address translation caches use replacement management tables that allow the software to direct entries with specific address ranges to a particular subset of the cache [42]. The PPE consists of the Instruction Unit (IU), the fixed point unit (XU) and the vector scalar unit (VSU). The IU fetches four instructions per cycle per thread into an instruction buffer and after decode and dependency checking dual issues these to the execution unit. All dual issue combinations are possible with the exception of instructions to the same execution unit and some exceptions as described in [42]. The XU has 32 64-bit general purpose register file per thread, a fixed point execution unit and a load store unit. The L1 D-cache associated with the XU is non-blocking allowing cache hits under misses. The VSU issue queue

decouples vector and floating point pipelines from the other pipelines allowing vector and floating point instructions to be issued out of order with respect to other instructions. The VSU floating point units has 32 64-bit register file per thread and a 10-stage double precision floating point unit. The VSU vector unit has 32 128-bit vector register file per thread and all instructions are 128-bit SIMD with varying lengths [42].

As can be seen from the above description of the architecture, the PPE allow exploitation of parallelism at multiple levels. The dual-issue nature of the architecture allows exploitation of ILP [35]. Further, ILP partially hides memory latency by concurrently servicing multiple outstanding cache misses [35]. Such Memory Level Parallelism (MLP) can also be used between threads to increase overall memory bandwidth utilization by enabling interleaving of multiple memory transactions. However, lack of instruction re-ordering capability and sharing of execution units limits the effective exploitation of ILP on the PPE. The architects favored these limitations of dual-issue for power efficiency [35]. The SIMD instruction set enables exploitation of Data Level Parallelism (DLP). The dual threaded nature of the PPE supports Thread Level Parallelism (TLP).

*Synergistic Processing Element:*

The SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). The SPU is a RISC core with a 256 KB software-controlled LS for instruction and data, and a 128-bit 128 entry unified register file. The execution units of the SPU are 128-bit wide and all instructions are 128-bit SIMD with varying widths [42]. The SPE ISA provides a rich set of vector such as arithmetic, logical, and load/store operations that can be performed on 128-bit vectors of either fixed point or floating point

values. The ISA also provides instructions to access scalars from vector registers enabling scalar operations on the SPE. Up to two instructions are issued per cycle, with one slot support fixed/floating point instructions and the other slot supporting load/store, byte permutation operations, and branch instructions. Single precision instructions are performed in 4-way SIMD fashion and are fully pipelined, while double precision instructions are performed in 4-way SIMD fashion, and are only partially pipelined. Also, double precision operations stalls the dual issue of other instructions making the Cell processor less suited for applications with massive use of double-precision instructions. The SPU assumes sequential execution of instructions leading to serious performance degradation on branch mispredictions. The ISA provides branch hint instructions enabling software to pre-fetch instructions at the target branch address.

Similar to the PPE, the SPE allows exploitation of parallelism at multiple levels. The SIMD instructions support DLP. ILP is obtained through the dual issue execution unit of the SPE. TLP is supported through the multiple SPE cores available on the Cell processor. At 3.2 GHz each SPE provides a theoretical peak performance of 25.6 GFlops/s of single precision performance and 2.6 GFlops/s of double precision performance.

*Memory Flow Controller:*

The MFC implements the communication interface between the SPE and PPE elements, and serves as a high-performance data transfer engine between the LS and Cell system memory. Data and instructions are transferred between the LS and the system memory through asynchronous coherent DMA commands. Since the address translation is governed by the PowerPC address and page tables, addresses can be passed between

the PPE and the SPE enabling the operating system to share memory and manage all resources in the system in a consistent manner. Also, LS to LS DMA transfers between SPEs are possible. The MFC controls DMA transfers and communicates with the system by means of unidirectional message interfaces known as channels. The channels support enqueueing of DMA commands and other facilities such as mailbox and signal-notification messages. The PPE and other devices in the system, including other SPEs, can also access the MFC state of an SPE through the MFC's memory-mapped I/O (MMIO) registers and queues, which are visible to software in the main-storage address space [3]. Each MFC can independently process DMA commands from its associated SPU and from other devices. Also, the MFC can autonomously process a list of DMA commands with up to 2048 such DMA transfers. The MFC supports naturally aligned DMA transfer sizes of 1, 2, 4 or 8 bytes and multiples of 16 bytes, with a maximum transfer size of 16 KB per DMA transfer. Peak transfer performance is achieved if both the effective address and the LS address are 128-byte aligned and the size of the transfer is an even multiple of 128 bytes.

A unique feature of the SPE is support of Compute Transfer Parallelism (CTP) where computation is parallelized with data and instruction transfer that feeds the computation. CTP is made possible by the asynchronous data transfers made possible by the MFC.

*Element Interconnect Bus:*

The Element Interconnect Bus (EIB) connects 12 elements – the PPE, 8 SPEs, the Memory Interface Controller (MIC) and the Bus Interface Controller (BIC) to each other [46]. The EIB runs at half the processor frequency and can transfer a maximum of 192

bytes per processor cycle. It has 12 ports for the elements each of which can read and write 16 bytes of data per bus cycle. Physically, the EIB consists of 4 rings with 2 rings transferring data clockwise and 2 rings transferring data counter clockwise. Each ring can transfer 16 bytes of data and supports 3 concurrent non-overlapping transfers. The EIB can thus support 102.4GB/s of coherent commands with transient rates as high as 307.2 GB/s. The Cell BE's external memory bandwidth is 25.6GB/sec inbound and outbound to the Rambus Dual XDR memory controller, roughly 3-8 times the bandwidth of a typical DDR memory bus [46].

The high bandwidth of the EIB supports streaming of data by allowing the SPEs to be arranged in a pipeline fashion, where each SPE kernel acts on the data, produce intermediate results, and pass on the data to the next SPE. Compared to SIMD, the stream model supports data parallelism at a larger granularity level and supports more complex data transformations. Although EIB supports simultaneous transactions, care must be taken to ensure that the transactions to do not block each other [46].

2.2.3.4 Nvidia Fermi

GPU computing refers to the use of Graphics Processing Units (GPUs) for high performance data parallel applications beyond graphics. The Fermi architecture to be released in early 2010, represents the latest in the evolution of Nvidia Compute Unified Device Architecture (CUDA), a software and hardware architecture that enables GPUs to be programmed with a variety of high level programming languages. The GPU design philosophy is based on the integration of a large number of specialized processing cores to support massive hardware thread level parallelism. Nvidia Fermi is currently

manufactured in the 40 nm process. Figure 2.4 shows the organization of the Nvidia Fermi.

FIGURE 2.4: Organization of the Nvidia Fermi.

*Processing Cores:*

The Nvidia Fermi architecture consists of 512 computing cores known as CUDA cores designed to execute one instruction per clock cycle for a thread before switching to another thread. Each CUDA core has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Unlike the general purpose processor cores, the CUDA cores lack individual register files, caches, or load store units to access memory. Instead, a set of 32 CUDA cores (known as a streaming multiprocessor: SM) share resources such as registers, caches/local memory and load store units. The 32 CUDA cores operate in parallel on 32 instructions from 32 threads (also known as "warp"). Each streaming processor features two warp schedulers and two instruction dispatch units thus allowing two warps to be issued and executed concurrently. Each streaming

multiprocessor can manage 48 such warps for a total of 1,536 threads. Additionally it features 4 texture engines and 4 polymorphic engines for graphics. The Fermi architecture consists of 16 such streaming multiprocessors with a capability of handling 24,575 parallel threads with 512 executed at a time. A central scheduler (Giga Thread Scheduler) schedules the warps on to the streaming multiprocessors. The Fermi architecture also incorporates 4 Special Function Units (SFUs) for complex math operations. The Fermi GPU is expected to run at 1.5 GHz and dissipate about 240 W.

*Memory System:*

Each streaming multiprocessor has a shared L1 instruction cache and a 64 KB of configurable local memory that can be partitioned as an L1 data cache and a general-purpose shared memory. The 16 streaming multiprocessors share a unified cache-coherent 768 KB L2 cache. The GPU is attached to up to 6 GB of local DRAM through 6 GDDR5 memory controllers with 172.8GB/s of memory bandwidth. Access to the system memory of the host CPU is through a PCI express bus. A special feature of the Fermi architecture compared to earlier GPU architectures is the extensive support for hardware error-correction codes to protect the external DRAM, L1 and L2 caches, and the register files from soft errors. From a programmer’s perspective, unlike its predecessors, the Fermi has a unified memory space of shared and global memory enabling C++ code to execute on the GPU.

## 2.3 Multicore Programming Tools

Historically, parallel software was limited to high performance computing, where domain specialists wrote parallel code which was often optimized for a given parallel architecture. With the advent of multicore processors, developing parallel software has

become a mainstream requirement. The success of the multicore revolution hinges critically on the availability of high productivity programming tools that enable a broad class of programmers to effectively develop software that exploits the parallelism inherent in the problem. In this section, we review the state-of-the-art programming tools to express parallel algorithms. The programming tools reviewed follow three principal approaches – parallel libraries, parallel languages, and parallelizing compilers. Since a large number of parallel programming libraries and languages exist, we limit our review to commercially available tools which support programming of multicore platforms in C/C++.

### 2.3.1 Parallel Libraries

The library based approach provides Application Programming Interfaces (API) that allows programmers to both explicitly generate parallel tasks and manage the communication and synchronization between the tasks. Parallel libraries are available for both shared memory and distributed memory machines. Libraries enable programmers to operate within the framework of popular sequential languages such as C, C++, Java and Fortran and incorporate parallelism through library calls. Although this approach gives programmers the greatest degree of control, beyond a small number of threads/processes such an explicit management of parallelism is bug prone and does not scale well to a large number of processing cores. However, currently the library approach is popular due to the availability of standardized parallel libraries on a wide variety of machines, and programmer familiarity. A brief review of popular parallel libraries for multicore computing is given below.

2.3.1.1 Shared Address Space

*PThreads:*

A thread is a single stream of control that can be independently scheduled by the operating system. In UNIX environments, threads exist within a process sharing resources with other threads while independently maintaining its own stack and data. In particular, inter-thread communication occurs within the shared address space. POSIX Threads (Pthreads) refers to the IEEE POSIX standard API for creating and managing threads. The implementation of the API is available on all commonly used UNIX flavors, Windows (Pthreads-win32), and Mac OS X. Pthreads is defined as a set of C language programming types and procedure calls implemented with a *pthread.h* header file. Pthreads are popular because of their ease of programming and portability. Some of the basic Pthread operations include creation and termination of threads, implementation of critical sections through mutual exclusion locks (mutex-locks), and thread synchronization through condition variables. While Pthread gives the programmer extensive control over threading operations, the inherently low level-API requires multiple operations to perform thread management tasks, thus making it more challenging to use. Pthreads is a good choice for event based, or I/O based parallelism.

*OpenMP:*

Open MultiProcessing (OpenMP) is a compiler directive based standardized API for programming shared address space machines. OpenMP enjoys support for C, C++ and Fortran and is available on many UNIX flavors, Windows, and Mac OS X. Unlike Pthreads, OpenMP is a higher level API where the user instructs the compiler through *pragmas* the concurrency, synchronization and data handling operations without

explicitly setting up and scheduling threads, mutex-locks and so on. It should be noted that OpenMP requires compiler support. Widely used compilers such as GNU GCC, IBM XL compiler for C/C++/Fortran, Intel compliers for C/C++/Fortran, and Microsoft Visual Studio 2008 C++ support OpenMP. A major advantage of OpenMP over Pthreads is that it does not tie the application to a pre-set number of threads. Also, the compiler directive based approach simplifies parallel programming since many of the tasks associated with thread creation and management are handled automatically. OpenMP is a good choice for data intensive computing with loop level parallelism.

*Intel TBB:*

Intel Thread Building Blocks (TBB) is a C++ template library from Intel Corp. for shared address space parallel programming on multicores. TBB includes algorithms, highly concurrent containers, locks and atomic operations, a work stealing task scheduler and a scalable memory allocator. Further Intel TBB provides generic parallel patterns such as parallel for-loops, parallel while-loops, data-flow pipeline models, parallel sorts, and prefixes. Similar to OpenMP, Intel TBB frees programmer from the explicit management of threads. However, unlike OpenMP Intel TBB does not require explicit compiler support. It is a good choice for compute intensive, highly object oriented C++ code. Intel TBB is open-source and is available on many UNIX flavors, Windows, and Mac OS X.

2.3.1.2 Distributed Address Space

*MPI:*

Message Passing Interface (MPI) is a language-independent communications library for parallel programming with processes, on distributed address space machines.

MPI provides for both point-to-point (send/receive) and collective communication (broadcast) operations between processes. Typically, the processes run on separate processor cores with no sharing of memory. However, the processes could also be located in a shared address space with inter-process communication through explicit memory copy. Similar to Pthreads, in MPI, parallelism is explicit since the programmer is responsible for generating and managing parallel processes. However, unlike Pthreads, the message passing paradigm of MPI implies explicit user control of the inter-process communication as well. While this makes it difficult to program with MPI, it encourages the development of parallel code with good data locality. MPI implementations such as Open MPI are available on all commonly used UNIX flavors, Windows, and Mac OS X. MPI is the parallel library of choice for massively parallel machines and workstation clusters.

#### 2.3.1.3 Stream Processing

Stream processing is a form of data parallelism, where data is streamed through a multiple computational units subjecting the data to a series of operations. Stream processing works well for certain applications such as signal processing and image processing where the data undergoes a series of transformations. Stream processors such as Graphics Processing Units (GPUs) employ both shared and distributed memory paradigms. The paradigm of GPU computing seeks to extend the use of GPUs for non-graphics high performance computing applications. We describe the OpenCL framework targeted at GPU computing.

*OpenCL:*

Open Computing Language (OpenCL) is a royalty free C and API based parallel programming framework targeted at a heterogeneous computing system consisting of a host processor connected to one or more OpenCL devices (processors and GPUs) [54]. OpenCL comprises of a C-based language for programming the compute kernel and platform and runtime APIs for control and communication operations. Compute kernels is the basic unit of executable code and is similar to C functions. The execution domain of a kernel is defined by an N-dimensional computation domain. Each element in the execution domain is a work-item and OpenCL provides the ability to group together work-items into work-groups for synchronization and communication purposes. OpenCL defines a multi-level memory model with memory ranging from private memory visible only to the individual compute units in the device to global memory that is visible to all compute units on the device. Depending on the actual memory subsystem, different memory spaces are allowed to be collapsed together. OpenCL is supported by a number of GPU vendors including Nvidia (GeForce and Qaudro series) and AMD (ATI Radeon series).

### 2.3.2 Parallel Languages

The language based approach provides new language constructs that enables programmers to express parallel operations independent of the underlying machine. Although numerous parallel languages have been developed by the research community, this approach has not been popular due to the diversity of parallel architectures and the corresponding support need to develop the compiler infrastructure. However, the emergence of multicores and the need parallel programming productivity has given

impetus to developing parallel languages. In this section we briefly review the shared memory parallel language Cilk++ and the partitioned global address space parallel language UPC.

#### 2.3.2.1 Shared Address Space

*Cilk++:*

Cilk++ is a shared address space parallel programming language based on C++ with an associated Cilk++ compiler. Cilk++ extends C++ with a few key words for parallel programming while maintaining the serial semantics of the original program. Similar to OpenMP and Intel TBB, Cilk++ frees programmers from explicit management of threads. However, unlike OpenMP which targets loop-level data parallelism, Cilk++ relies on parallelizing function calls through a divide-and-conquer approach. Also, compared to OpenMP, Cilk++ has better support for nested parallelism and provides guaranteed space bounds (on $P$ processor Cilk++ does not occupy more than $P$ times the serial space). The Cilk++ run time system uses a dynamic work-stealing scheduler that supports dynamic load balancing. Cilk++ also comes with productivity enhancing tools such as a parallel performance analyzer to estimate the parallel code performance (such as processor scalability) and a race detector to find race conditions. Cilk++ was recently acquired by Intel Corp. from Cilk Arts Inc. and is currently available on Linux and Windows for x86 architectures.

#### 2.3.2.2 Partitioned Global Address Space

The Partitioned Global Address Space (PGAS) programming model has a logically shared global address space that is logically partitioned such that each partition is local to one processor. Thus unlike a shared address space programming model, the

threads have affinity to one or more of the partitions. The PGAS programming model is the basis of Unified Parallel C.

*Unified Parallel C:*

Unified Parallel C (UPC) is a parallel extension of the C programming language supporting both shared and distributed memory machines through the PGAS programming model. UPC uses a thread based parallel execution model with data declared as either shared between threads or private to each thread. For shared data the same address refers to the same memory location while for private data the same address corresponds to different memory locations. The language provides constructs for specifying a thread ownership (affinity) of shared data. All scalar data including pointers and user defined aggregate types have affinity with thread 0 while the thread affinity of array data is specified at the cyclic, blocked-cyclic, or blocked level. UPC provides constructs for explicit synchronization between the threads. Currently available UPC compilers include GCC UPC, IBM XL UPC, HP UPC, Cary UPC, and Berkeley UPC. The Berkeley UPC compiler infrastructure is a layered design including a top level Open64 compiler based UPC to C translator, followed by a run time system (performance instrumentation, communication tracing and debugging), and a GASNet communication system (language and network independent low-level networking layer providing high performance communication primitives) [15]. UPC is available on all commonly used UNIX flavors, Windows (Pthreads-win32) and Mac OS X. Hardware platforms supported including x86, SPARC, MIPS, PA-RISC and PowerPC architectures, clusters (Ethernet, Infiniband, Myrinet) and massively parallel processors (Cray XT3, IBM Blue Gene).

### 2.3.3 Parallelizing Compilers

Parallel compilers seek to automatically recognize parallel structures and generate multi-threaded a sequential code with minimum programmer input. While successful automatic parallelization of sequential code can greatly enhance programmer productivity, in practice efficient parallelization of sequential code continues to be a challenging task. Parallelizing compilers have been most successful on array loops where precise memory dependence analysis is possible. However, the use of pointers, recursion, and dynamic data structures in C/C++ code makes hard for compilers to analyze dependences. Recently the technique of Thread Level Speculation, where possibly parallel sections of the code are speculatively executed and the execution rolled back if dependence violations are detected, have been used in research compilers [49]. Profile driven parallelization, where sequential code instrumentation of memory access is used to detect parallelization opportunities, has also been proposed in the literature as a means to find parallelizable tasks [49, 67]. Automatic parallelization option is available on popular compilers such as GNU GCC, Intel ICC, and IBM XLC.

## 2.4 Multicore System Software

Operating System (OS) serves as an interface between the hardware architecture and user level applications. Unlike traditional high performance computing applications, general purpose parallel computing using multicore processors require much more OS support spanning a diverse range of applications. Moreover, the diversity in multicore architectures implies that portability of architecture specific OS optimizations is limited across architectures and even among successive generations of the same architecture, making design of OS for multicores a challenging task.

The types of parallel OS are closely tied to the underlying parallel architecture. Common parallel OS designs include – 1) Separate OS per processor, common in message passing clusters 2) Master-Slave OS where the master processor runs the OS, while the slave processor runs the user processes. This paradigm is common in Asymmetric Multiprocessing Systems (ASMPs) and 3) Shared memory OS for shared memory multiprocessors, where the OS can run on any of the processors. Since most of the commercial general purpose multicore processors today use the shared memory paradigm, we briefly review support for shared memory in popular operating systems. However, scalability limitations of the shared memory OS design approach have led to research on message passing approaches as used in the multi-kernel OS design and the factored OS (FOS) [74, 75]. We briefly review the recently introduced multi-kernel Barrefish OS. A related development in dealing with scalability limitations of complex monolithic OS is the use of virtualization, where multiple (and often diverse) operating systems run on a hypervisor layer, with the hypervisor layer managing the machine's physical resources. We briefly review the benefits of OS virtualization of multicores.

### 2.4.1 Shared Memory OS

Shared memory OS associated with shared memory processors has one copy of the OS kernel in memory which can be executed by any of the processors. System calls are trapped and served by the processor on which it is issued. However, the need to prevent concurrent access to shared resources such as OS tables results in performance bottlenecks. Although splitting the shared resources into fine-grained critical sections ameliorates some of these bottlenecks, the need to keep track of these critical sections and guard against deadlocks and race conditions limits the robustness and portability of

this approach. Popular OS such as Linux and Windows have shared memory support. We briefly review SMP support in Linux.

*Linux:*

From its early days Unix (and its flavors such as Linux) has provided abstractions that that enable memory sharing of well defined regions of the process space as well as synchronous and asynchronous inter-process communication through semaphores and message queues. Support for shared memory in Linux has improved considerably with the introduction of Linux kernel 2.6. The pre-2.6 scheduler used a poorly scaling $O(n)$ scheduling algorithm. Also, the pre-2.6 scheduler used a single run-queue for all processors which meant that a task could be scheduled on any processor. While this was good for load balancing, it was resulted in poor cache efficiency. The pre-2.6 scheduler also used a single run-queue lock resulting in decreased efficiency when processors were idled waiting for release of the lock. Also, the earlier scheduler did not allow preemption, resulting in possible execution of lower priority tasks when the higher priority tasks were awaiting execution. The Linux 2.6 version, uses an $O(1)$ scheduler based on the number of task priorities rather than the number of active tasks resulting in good scaling of the performance of the scheduler with the number of threads. Moreover, each processor now maintains a separate run-queue with a separate lock on each run-queue. The separate run-queue per processor allows for better cache affinity of the task. To maintain load balance across the processors, every 200 ms, the scheduler does a cross-CPU balancing of tasks. The Linux 2.6 scheduler also supports task preemption and dynamic task prioritization.

### 2.4.2 Multikernel OS

Microsoft in collaboration with ETH Zurich has unveiled a new multicore oriented message-based OS known as the Barrelfish [7]. As shown in Figure 2.5, the OS is designed as a distributed system of cores that communicate using messages and share no memory. The motivation behind the *multikernel* approach include the emergence of on-chip message passing interconnects, portability limitations of shared memory OS kernel optimizations, and the scalability limitations of cache coherent shared memory. Note that although the *microkernel* approach also uses messages between processes, unlike the *multikernels*, it follows a shared memory paradigm with multithreaded micro kernels.

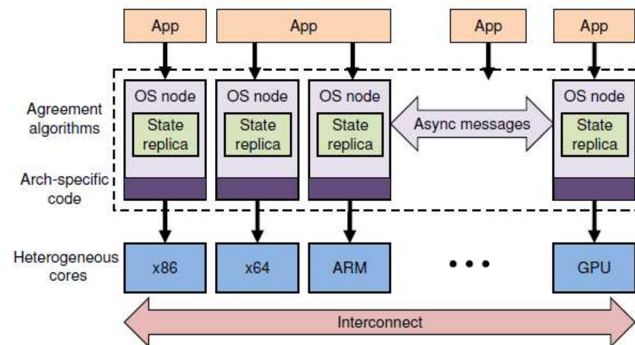



FIGURE 2.5: Multikernel model.

Baumann et. al. list the following three design principles behind the multi-kernel design [7] –

(1) Make all inter-core communication explicit through the use of explicit messages.

(2) Make OS structure hardware-neutral by separating the OS structure as much as possible from the hardware with only the message transport mechanism and interface to hardware being dependent on the machine.

(3) View state as replicated instead of shared by treating access and updates of shared states in a multi-kernel as local replicas while maintaining consistency through messaging.

The organization of Barrelfish is shown in Figure 2.6. The privileged mode CPU drivers are local to a core and handles functions such as protection, authorization, time slicing of processes and interface to hardware. The CPU driver is event-driven, single threaded and non-preemptable. These features make it easier to develop and maintain the CPU driver compared to a conventional kernel. The user mode monitor process collectively co-ordinates system wide states such as memory allocation tables and address space mappings through inter-core message based agreement protocols. Initial evaluation results show good core scaling on microbench marks such as TLB shootdown and compute bound parallel benchmarks such as NAS and SPLASH.

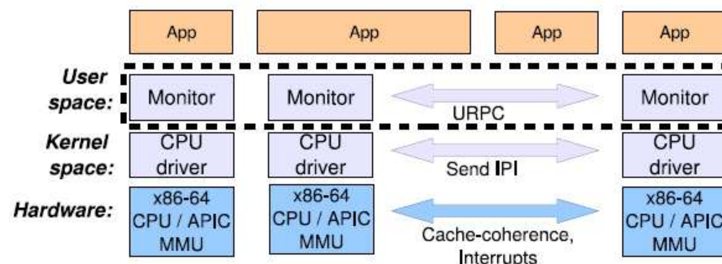



FIGURE 2.6: Organization of Barrelfish.

### 2.4.3 Virtualization

*Xen:*

Hypervisor is a hardware/software platform that serves as an interface between the hardware and the OS. Hypervisor enables virtualization where multiple OS can run on the same hardware. Processor vendors have recently introduced hardware support for virtualization such as the Intel VT-x technology in the Nehalem architecture, allowing for root operation for hypervisors, and non-root operation for guest OS. Virtualization provides several benefits including better utilization of hardware, better security through isolation of virtual operating systems, and the ability to run legacy software and OS. Figure 2.7 demonstrates one approach proposed by Youseff and Wolski for using the virtualization paradigm as a means for customizing the OS for the core architecture and the associated workloads in heterogeneous multicores [80]. The virtualization paradigm also leads to better cache efficiency on cc-NUMA multicore architectures by pinning virtual OS instances to a core thus improving cache locality. Moreover, virtualization can also help in saving power by consolidating low utilization loads to one processing core, and turning the other processing cores off.

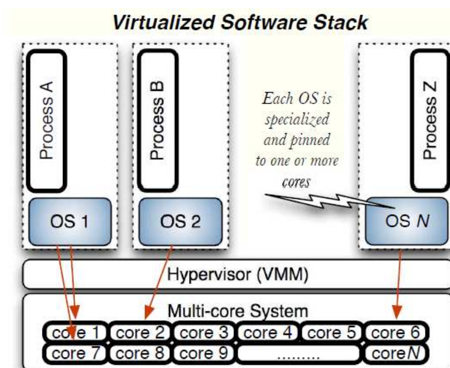



FIGURE 2.7: A simplified representation of the virtualized software stack, demonstrating the deployment of a hypervisor and several VMs, each of which is managing a subset of the cores and a subset of the processes.

# CHAPTER 3: DESIGNING CACHE- AND SPACE-EFFICIENT DATA PARALLEL ALGORITHMS FOR MULTICORES

## 3.1 Introduction

In this chapter, we present a design methodology that aids in the development of parallel cache-efficient and space-efficient algorithms for shared cache multicore processors. Shared cache multicore processors differ from traditional shared memory processors in two significant ways (See Chapter 2) –

(1) The processing cores, the interconnect, and a part of the shared memory hierarchy are on the same chip/module resulting in potentially lower communication and synchronization costs. For example the Sun UltraSparc T2 processor has 8 cores, a crossbar interconnect, L1, and L2 cache integrated on a single chip. The Intel Clovertown processor has four cores, the bus interconnect, L1 and L2 caches integrated on a single chip/module.

(2) The shared memory (typically the L2 or L3 cache) is not only shared by all or a subset of the processing cores but is of a limited size. For example in the Sun UltraSparc T2, the 4MB L2 cache (8 banks) is shared by all 8 processing cores. The quad-core Intel Clovertown processor has private L1 caches, pairs of cores sharing the 4MB L2 cache while all 4 cores share the main memory.

The integration of the processing cores, the interconnect, and the cache hierarchy on a single chip necessitates micro-architectural tradeoffs between the performance, die

area, and power budgeted to the different components. Power constraints have led processors vendors to incorporate lower frequency simple in-order processing cores. For example, the Sun UltraSparc T2 processor cores have a simple 8-stage pipeline and operate at a frequency of 1.4 GHz. The reduction in single threaded performance in favor of better multithreaded performance requires algorithms that are parallel and scalable for continued high performance on multicores. The increasing latencies up the multiple levels of the memory hierarchy, and sharing of the limited sized cache between multiple processing cores motivates the need to formulate scalable parallel algorithms that are optimal both with respect to the memory used (space-efficient), and to the number of read/write operations between the different levels of the cache hierarchy (cache-efficient). While utilizing the concurrency in the problem enables development of work efficient parallel algorithms, space- and cache-efficiency can be achieved by exploiting the data locality in the problem.

In achieving the above described objectives, we note the importance of a computational model of the targeted multicore architecture that offers a good trade-off between simplicity and accuracy. Cache-aware and cache-oblivious algorithms that are both cache-efficient and space-efficient can then formulated on this model. Scheduling algorithms are then employed that map the concurrent tasks to the computing cores such that the load is balanced and the data locality demands of shared and private caches are satisfied. In the past few years considerable work has been reported on computational models, cache-oblivious algorithms, and schedulers for multicore processors. *In this chapter, we contribute to the existing body of knowledge by proposing a parallel cache-*

*oblivious algorithm design methodology for devising space-efficient and cache-efficient algorithms through maximal data sharing between concurrent tasks.*

The computing problems considered in this chapter are data parallel where the concurrency in the problem is best described in terms of decomposition of the underlying data structures. Further, we limit ourselves to problems where the underlying data structures are multidimensional arrays usually representing the inherent geometry of the problem. In such cases, the decomposition of the arrays along one or more dimensions into sub-arrays represents a geometric decomposition of the problem. Computations with the above mentioned characteristics are widespread in scientific computing especially in numerical solutions of partial differential equations, and in image processing. Owing to the increasingly large data sizes inherent in these applications, a geometric decomposition of the problem facilitates the effective utilization of limited size cache hierarchies as well as a parallel solution of the problem.

The remainder of the chapter is organized as follows – Section 3.2 reviews the existing literature on computational models, design of cache-oblivious algorithms, and schedulers for multicore processors. In Section 3.3, we describe the proposed parallel cache-oblivious algorithm design methodology for multicores. In Section 3.4, we present case studies on the application of the proposed methodology to two representative data parallel problems from scientific computing – matrix algebra (dense matrix multiplication), and stencil computing (Finite Difference Time Domain). Section 3.5 concludes this chapter with a step-by-step elucidation of the design methodology for developing space-efficient and cache-efficient algorithms on multicores.

## 3.2 Background

In this section we review the existing literature on parallel computational models, the cache-oblivious model, and scheduling strategies that have been proposed for multicore computing.

### 3.2.1 Computational Models

The development of sequential algorithms has benefitted greatly from the Random Access Model (RAM) of computing which has been very successful in abstracting a great variety of uniprocessors. However, the parallel extension of the RAM, the Parallel RAM (PRAM) has been far less successful in accurately capturing the behavior of parallel machines primarily due to its assumption of a global infinite address space shared by $P$ processors accessible in constant time [50]. A variety of parallel computational models such as the Bulk Synchronous Processor model (BSP) [66, 68], Parallel Disk Model (PDM) [72, 73], and the Log-P model [24] have been proposed to remedy the situation. In general, these models seek to include all or a subset of parameters such as computational parallelism, communication latency, communication overhead, communication bandwidth, execution synchronization, memory hierarchy, and network topology [50]. With the introduction of parallel machines in the form of chip multiprocessors (multicores), researchers have sought to adapt the existing parallel computational models to this new platform.

In [69] Valiant extends the BSP model to multicore processors. The model uses $4d$ parameters $\{p_i$, $g_i$, $L_i$, and $M_i\}$ where $d$ is the depth of the memory hierarchy, and at level-$i$, $p_i$ is the number of $i$-$1$ components in $i$, $g_i$ is the communication bandwidth, $L_i$ is the latency, and $M_i$ is the size of the memory not inside an immediately lower level.

General lower bounds are established for communication and synchronization complexities and optimal multi-BSP algorithms are derived for matrix multiplication, FFT, and sorting. Savage and Zubair [62] have proposed a Unified Model for multicores, where for a $d$-level memory hierarchy, the parameters at level-$i$ include, $p_i$ the number of cores sharing a cache, $\alpha_i$ the number of caches, and $\sigma_i$ the size of the cache. The application of the model is illustrated for matrix multiplication, FFT, and binomial options pricing. Blelloch et. al. [11] have proposed a computational model for multicores based on the Parallel Disk Model (PDM) with a two level cache hierarchy with a private L1 cache and a shared L2 cache. The model parameters include $p$ the number of processing cores, $C_1$ the size of the L1 cache, $C_2$ the size of the L2 cache, and $B$ the size of the L1 and L2 cache blocks transfers. An online scheduler is proposed for divide-and-conquer algorithms including matrix multiplication, matrix inversion, sorting, and the Gaussian elimination paradigm. A similar multicore model is used by Chowdhury and Ramachandran [23] with additional parameters $B1$ and $B2$ for the size of the L1 and L2 cache blocks. The model is applied to derive parallel dynamic programming algorithms for the local dependence dynamic programming problems, the Gaussian elimination paradigm, and the parenthesis problem.

While the multi-BSP model is the most general of the multicore models, the large number of parameters makes the model difficult to use. The model used by Blelloch et. al. [11] and Chowdhury et. al. [23] are limited to a two level memory hierarchy with private and shared caches. However, modern processors have deep memory hierarchies (upto three levels of on-chip cache, main memory and disk storage), which is more accurately modeled by Savage and Zubair's Unified Model for Multicores [62]. In our

work on cache-oblivious multicore algorithms, we therefore utilize the Unified Model for Multicores.

### 3.2.2 Cache-oblivious Model

The cache oblivious model was proposed by Frigo et. al. to design portable algorithms for uniprocessors with deep memory hierarchies [29, 32]. The model simplifies the Parallel Disk Model (PDM) by ignoring the parameters $B$ (the cache line size or block size) and $M$ (size of the memory). Further the model assumes an optimal cache-line replacement strategy where the cache line evicted will be accessed furthest in the future. Note that the real caches use replacement policies such as Least Recently Used (LRU) or replacing the oldest block (FIFO). However, as shown in [29], the cache- and space-complexity of the optimal replacement policy differs from those of LRU and FIFO by a constant factor. Also, the caches are assumed to be fully associative and the cache is assumed to be taller than it is wide. Among the advantages of the model are [27] –

(1) If the algorithms perform well for two levels of memory, it easily extends to any two levels in an arbitrarily deep memory hierarchy due to the inclusion property.

(2) If the memory transfers are optimal to within a constant factor between any two levels of the memory hierarchy, then any weighted combination of the memory transfers between different levels of the memory hierarchy, with weights corresponding to the relative memory speeds, is optimal to within a constant factor.

(3) Since the model makes minimal assumptions about the machine, the resulting algorithms are portable on a wide variety of machines. However, in practice, the cache parameters $B$ and $M$ are required to determine the base case of recursions.

Note that the algorithms designed using the cache-oblivious model does not explicitly manage the cache since that would involve explicit use of cache parameters. As described in Chapter 2, in many of today's multicore shared cache architectures the block replacement is decided by the cache hardware according to a fixed cache-line replacement strategy and is not under programmer control. However, emerging multicore architectures such as the IBM Cell/B.E. (See Chapter 2) are cache-less and allow the programmer to explicitly control of the local memory.

Cache-oblivious algorithms are formulated typically using the recursive divide-and-conquer strategy where the underlying problem is repeatedly divided until the smallest instance fits into the cache (base case of recursion). Recurrence relations for the number of memory transfers are then developed and solved to estimate performance bounds [27]. Cache-oblivious algorithms rely on cache-oblivious data structures where data is laid out in the cache in a recursive fashion. Examples of cache-oblivious data structures include space filling Peano curves for matrix operations [6], van Emde Boas layout for static search trees [70], cache-oblivious B-trees [8], priority queues [4], and linked lists [9]. Similar to the memory transfers, the space requirements for cache-oblivious data structures are estimated by solving appropriate recurrence relations [27]. Cache-oblivious algorithms and data structures have been developed for a number of problems including searching, sorting, and matrix operations [29], graph operations, computational geometry [4], stencil computing [31], and dynamic programming algorithms [22].

#### 3.2.3 Multicore Schedulers

Consider the computations as modeled by Directed Acyclic Graphs (DAGs). The number of vertices in the DAG determines the total work while the depth corresponds to the longest path in the DAG. A scheduler maps each vertex to a (time step, processor) pair such that each processor has at most one task per time step and no dependence is violated [10]. An off-line scheduler has knowledge of the DAG before start of the computation, while the structure of the DAG is revealed to an on-line scheduler as the computation proceeds. In our work, we only consider problems amenable to off-line scheduling.

Different sequential and parallel scheduling algorithms have been proposed in the literature. In a breadth-first sequential schedule (1BF), a node is scheduled only after all the higher level nodes have been scheduled. In a depth-first sequential schedule (1DF) the scheduling is as follows – at each step, if there are no scheduled nodes with a ready child, a root node is scheduled; else the ready child of the most recently scheduled node with a ready child is scheduled. A greedy parallel $p$-schedule schedules nodes such that if at least $p$ nodes are ready then $p$ nodes are scheduled; else if fewer than $p$ nodes are ready, all the ready nodes are scheduled. A greedy scheduler thus attempts to do as much work as possible on each time step. Among the state-of-the-art greedy schedulers are Parallel Depth First (PDF) and Work Stealing (WS). In a depth-first parallel schedule (PDF) the ready-to-execute nodes are prioritized based on a 1DF schedule. In a WS scheduler each processor maintains a local queue of ready-to-execute nodes. If the local-queue of a processor is empty, then the nodes from the bottom of a non-empty queue are scheduled on that processor. Recent work suggests that for computation with fine-grained data

parallelism, a PDF scheduler performs better then a WS scheduler on shared cache multicores due to constructive cache sharing [21]. Hybrid schedulers that combine the PDF with WS [55] and the 1DF with PDF [11] have also been reported. While the former has only been evaluated experimentally for certain benchmarks, the latter has been shown to have provably good performance on multicores for many divide-and-conquer algorithms.

## 3.3 Parallel Cache-oblivious Design Methodology

We now describe a parallel cache-oblivious algorithm design methodology for developing cache-efficient and space-efficient data parallel algorithms using a weighted-vertex red-blue pebbling game.

### 3.3.1 Computational Model

We use Savage and Zubair's Unified Model for Multicores [62] as a basis for developing cache and space-efficient algorithms. Our model consists of $2d$ parameters where $d$ is the depth of the shared cache hierarchy. For a multicore processor, the RAM storage can be considered to be the top most level of the memory hierarchy (level-$d$). The level-$d$ memory is shared by all the processing cores and is considered to be sufficiently large enough to hold the input data set. For $1 \leq i \leq d$, the model parameters include –

- $P_i$ : The effective number of level-($i$-$1$) caches (or processing components) contained in level-$i$. $P_1$ is the number of processing cores/threads associated with the L1 cache.
- $M_i$ : Total memory available (in bytes) on a component at level-$i$.

To simplify analysis with this model we adopt the following assumptions proposed by Blelloch et.al. for a tree-of-caches hierarchy [12] –

(1) The memory hierarchy is considered to be inclusive – each cache line at level-$i$ is also cached in its parent cache at level-($i+1$). Further we assume that $M_{i+1} > \alpha_i M_i$ where $\alpha_i \geq P_i$.

(2) The caches in the hierarchy are considered to be fully associative.

(3) The model assumes a variant of the DAG consistency cache consistency model that uses an optimal cache-line replacement strategy where the cache line evicted will be accessed furthest in the future.

(4) Caches are considered non-interfering in that cache misses by one processor can be analyzed independently of other processors. To maintain this property, the BACKER cache-coherence protocol proposed by Blumofe et.al. [14] is used. The protocol ensures that while instructions in a DAG see writes by their ancestors, concurrent writes by instructions with no path between them are not seen. Such writes are only seen in the shared memory and are reflected in other cache copies when the descendant instructions tires to access them.

The proposed set of parameters models most commercially available homogenous multicore processors with all inter-processor communication occurring through the memory hierarchy. The values of the parameters $P_i$, and $M_i$, can be obtained from the processor data sheets. For the Intel Clovertown processor ($d = 3$), our computational model uses a total of $6$ parameters. Here, level-1 of the cache hierarchy is the L1 cache and level-3 is the main memory. As shown in Figure 3.1, the effective number of processors for each level-$i$ are $P_1 = 1$, $P_2 = 2$, and $P_3 = 2$.

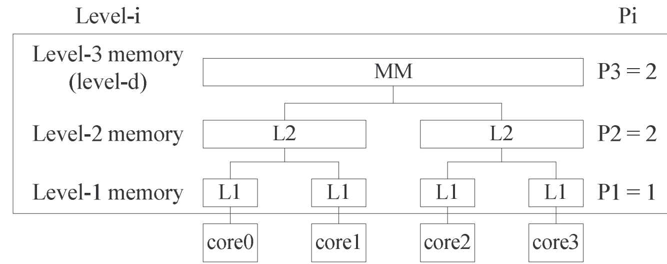



FIGURE 3.1: The cache hierarchy of the Intel quad-core Clovertown processor.

We now detail a design methodology that enables the development of scalable cache-efficient and space-efficient parallel algorithms under the above described computational model. The resulting algorithms seek to be optimal with respect to the memory used (space-efficient), and with respect to the number of read/write operations between the different levels of the cache hierarchy (cache-efficient).

### 3.3.2 Recursive Geometric Decomposition

As mentioned in the introduction, the computing problems considered in this chapter employ multidimensional arrays with the arrays typically representing the inherent geometry of the problem. The arrays can be broken along one or more dimensions into sub-arrays (also known as blocks) and the computation described in terms of updates of these blocks. Further, these blocks can be divided recursively without explicit consideration of the cache parameters into smaller blocks representing a finer granularity of decomposition. Blocking exploits the temporal locality inherent in these computations thus reducing the memory traffic. If the updates require the use of boundary values, the decomposition may require overlapping of blocks which include the boundary values needed to update that block. For a size limited cache-hierarchy, the goal is to find the maximum size of the recursively decomposed block $B_i$ at memory hierarchy level-$i$

such that the read-write operations between the different levels of memory are minimized by maximally exploiting the temporal locality of the decomposed $B_i$ problem. We now describe the use of Hong-Kung's red-blue pebble game [40] to achieve this objective.

### 3.3.3 Red-Blue Pebble Game

The red-blue pebble game proposed by Hong and Kung [40] is a graph pebbling game that enables the determination of the lower bound for memory read/writes in a computing machine with a two level memory hierarchy. Red pebbles represent the lower level faster memory while the blue pebbles represent the higher level slower memory. The number of red pebbles is finite, modeling the limited size of the faster memory while the number of blue pebbles is infinite, modeling the large size of the slower memory. We represent the computation as a Directed Acyclic Graph (DAG) *G(V,E)* with **V** vertices and **E** edges. Here the vertices represent the input, output, and intermediate data while the edges represent the data dependencies. The input nodes of *G(V,E)* are initially pebbled with blue pebbles. Red pebbles can replace blue pebbles and vice-versa modeling the read/write operations between the faster and slower levels of memory. However, red pebbles can only be placed on non-input vertices of *G(V,E)* if all the parent vertices hold red pebbles. This constraint models the fact that a vertex can only be computed if all the parent vertices are present in the faster memory. Also, a pebble holding the input to a vertex can be reused to hold the results of the computation of a vertex. *The goal of the red-blue pebble game is to pebble all the output vertices of G(V,E) with red pebbles.* The red pebbles on the output vertices are then replaced by blue pebbles thus completing the pebble game. The total memory read/write between the faster and slower memory is then calculated by the number of times the red and blue pebbles replace each other. Note that

for a given $G(V,E)$ and a finite number of red pebbles, different pebbling strategies are possible.

For a graph $G(V,E)$ and a pebbling strategy $\Omega$, the total number of two level memory read/write operations is at least equal to the number of input and output vertices since the input has to be read from the slower to the faster memory and the output has to be written from the faster to the slower memory. Also, the number of red pebbles required to play the red-blue pebble game is at least as much as the maximum degree of input to any vertex in $G(V,E)$. This follows from the requirement that a vertex can only be pebbled with a red pebbled if its parent vertices hold red pebbles. These observations are stated in the following two lemmas.

**Lemma 3.1** *Let $N^{R/W}$ be the number of two level memory read/write operations and $|In(G)|$ and $|Out(G)|$ be the number of input and output vertices respectively of $G(V,E)$. Then,*

$$N^{R/W} \geq |In(G)| + |Out(G)| \tag{3.1}$$

**Lemma 3.2** *Let $S$ be the number of red pebbles used by a pebbling strategy $\Omega$ and $\alpha$ be the maximum input degree of $G(V,E)$. Then,*

$$S \geq \alpha \tag{3.2}$$

Lemma 3.1 gives the lower bound on the number of memory operations required for computing a graph $G(V,E)$ such that the temporal locality of the problem is fully exploited. For a pebbling game with a finite number of red-pebbles $S$, we use the following result originally by Hong and Kung and modified by Savage [60, 61] to estimate the lower bound on $N^{R/W}$.

**Lemma 3.3** *Let ρ(S,G) be the S-span of G(V,E) where the S-span of DAG G(V,E) is defined as the maximum number of vertices of G that can be pebbled with S red pebbles in the red-blue pebble game maximized over all initial placements of S red pebbles [61]. Let $N^{R/W}$ be the number of two level memory read/write operations and let |In(G)| and |Out(G)| be the number of input and output vertices respectively of G(V,E). Then,*

$$N^{R/W} \geq \frac{S(|V| - |In(G)|)}{\rho(2S,G)} \tag{3.3}$$

Intuitively, the Hong-Kung lower bound given by Lemma 3.3 represents the tradeoff between the data read/write time and memory usage. For a finite number of red-pebbles *S*, our goal is to develop a pebbling strategy *Ω* that is optimal with respect to the Hong-Kung lower bound. By ensuring the minimum number of read/write operations between two levels of memory, the computation determined by such a pebbling strategy best exploits the temporal locality of data.

### 3.3.4 Nominal Parallel Pebbling Strategy

For the data parallel problems considered in this chapter, the computation at memory hierarchy level-*i* can be described using a DAG $G_i(V,E)$ with $G_d(V,E)$ describing the computation on the whole problem. From a data point of view, $G_{i+1}(V,E)$ represents computation on the data block $B^o_{i+1}$ at level-($i+1$) with the vertices of size $|B^o_i|$. Due to the recursive geometric decomposition of the problem as described in Section 3.3.2, $B^o_i$ is a subset of $B^o_{i+1}$ with $B^o_d$ representing the whole data set. Thus $G_i(V,E)$ is a sub-DAG of $G_{i+1}(V,E)$ with $k^o_i$ such sub-DAGs modeling the computation at level-*i* where $k^o_i$ is given by,

$$k_i^o = \frac{|B_{i+1}^o|}{|B_i^o|} \tag{3.4}$$

We then utilize the red-blue pebble game and develop a pebbling strategy $\Omega^o$ utilizing $S^o$ number of pebbles at level-$i$ to pebble the DAG $G_i(V,E)$ while seeking to be optimal to within a constant factor of the Hong-Kung lower bound. Here level-$i$ is considered to be the faster memory while level-$(i+1)$ is considered to be the slower memory. Note that the due to the recursive geometric decomposition of the problem, the DAGs and sub-DAGs at all levels of the memory hierarchy have the same topology but differ in the size of the data represented by their vertices. Hence the number of pebbles $S^o$ used by the pebbling strategy $\Omega^o$ is the same for all the DAGs at all levels. The block size $|B^o{}_i|$ is then given by the inequality,

$$|B_i^o| \leq \frac{M_{i+1}}{S^o P_{i+1}} \tag{3.5}$$

Here $|B^o{}_0|$ is the size in bytes of the underlying data type. Note that assuming a work-efficient scheduling strategy, $P_i$ such DAGs $G_i(V,E)$ are pebbled in parallel. The details of the scheduling strategy are presented in Section 3.3.6.

The nominal pebbling strategy thus helps determine the block size at each level-$i$. In the cache-oblivious model the block size represents the base case of the recursions used to determine the space complexity and cache complexity of the algorithm (See Section 3.2.2).

### 3.3.5 Weighted-vertex Parallel Pebbling Strategy

The nominal pebbling strategy $\Omega^o$ pebbles a given DAG $G_{i+1}(V,E)$ by parallel pebbling $P_i$ sub-DAGs $G_i(V,E)$ independently such that the pebbling of each $G_i(V,E)$ is optimal respect to the Hong-Kung lower bound. However, the resulting pebbling of the parent DAG $G_{i+1}$ is not optimal since, depending on the degree of sharing, the shared vertices between the individual sub-DAGs $G_i$s may be pebbled more than once. We now

outline a weighted-vertex DAG pebbling strategy $\Omega^s$ that considers the degree of vertex sharing in pebbling the sub-DAGs $G_i$s to minimize multiple pebbling of these vertices. As shown in Section 3.4, the pebbling strategy $\Omega^s$ thus results in an equal or lower number of read-writes between memory hierarchy levels $i$ and $i+1$ compared to $\Omega^o$. Consider the $k^s{}_i$ sub-DAGS $G_i(V,E)$ of DAG $G_{i+1}(V,E)$ where $k^s{}_i$ is given by,

$$k_i^s = \frac{|B_{i+1}^s|}{|B_i^s|} \tag{3.6}$$

Here $|B^s{}_i|$ is the size of the block under $\Omega^s$. As in the nominal case, we assume a work-efficient scheduling strategy (See Section 3.3.6) such that $P_i$ such sub-DAGs are computed in parallel. The pebbling strategy $\Omega^s$ in pebbling this $P_i$ sub-set of $G_i$s is as follows –

(1) Assign a weight $w$ to each vertex corresponding to its out-degree. In determining $w$, presence of the vertex in other $G_i$s of the subset (sibling DAGs) must be considered.

(2) Decide on a computational order in calculating the sub-set of $G_i$s.

(3) To start the game, pebble any $\alpha_i$ input vertices of $G_i$ with red pebbles following the pebbling strategy $\Omega^o$ following computational order of the problem.

(4) When a vertex is pebbled, all the vertices representing the same data in the sibling DAGs (data sharing) is also covered by that pebble. We refer to this pebbling operation as pebble cloning.

(5) To pebble the remaining vertices use the following rules consistent as follows:

    (a) If $w > 1$, the red pebble on a vertex can neither be deleted nor moved to another vertex.

(b) If $w = 1$, a red pebble on a vertex can only be moved to the immediate child vertex.

(c) If $w = 0$, a red pebble on a vertex is moved to any other vertex.

(d) A new pebble is introduced into the game when no currently used pebble can neither be deleted nor moved.

(e) When a vertex is pebbled, the weights $w$ of the parent vertices are decreased by 1.

(6) When all the output vertices of the sub-set of $G_i$s are pebbled once, the game ends.

(7) Repeat this game for the different $k^s_i/P_i$ subsets. Note that sharing of pebbles between the different $k^s_i/P_i$ subsets may also be possible.

Let $S^s$ be the average number of pebbles required for pebbling sub-DAGs $G_i$s at level-$i$. Assuming the typical case of $k^s_i >> P_i$, the block size $|B^s_i|$ is then given by the inequality,

$$|B_i^s| \leq \frac{M_{i+1}}{S^s P_{i+1}} \tag{3.7}$$

Note that $|B^o_0| = |B^s_0|$ and $|B^o_d| = |B^s_d|$. The weighted DAG pebbling strategy is applied to all levels of the cache hierarchy to obtain the parallel algorithm for the problem.

### 3.3.6 Data-aware Scheduling

Scheduling for multicores is challenging due to the conflicting data sharing demands of private and shared caches. Private cache performance is good when the processors work on disjoint cache sets. On the contrary shared cache performance is good when the processors work on the same cache blocks at the same time. For the parallel

pebbling strategies described in Sections 3.3.5 and 3.3.6, we use the CONTROLLED-PDF scheduling algorithm [11] proposed by Blelloch et. al. for divide-and-conquer problems. The algorithm is a hybrid combination of the 1DF and PDF schedulers outlined in Section 3.2.3. The scheduler assumes a multicore computational model with a two level cache hierarchy having a private L1 cache and a shared L2 cache. The model parameters include $p$ the number of processing cores, $C_1$ the size of the L1 cache, $C_2$ the size of the L2 cache, and $B$ the size of the L1 and L2 cache blocks transfers.

Similar to the hierarchical DAGs described in Section 3.3.4, a given computation DAG $G(V,E)$ with $n$ nodes is contracted to $n_2$ L2-supernodes each of which are in turn recursively contracted into $n_1$ L1-supernodes. The L2-supernodes (L1-supernodes) represent the granularity of computation at the L2 (L1) cache level. Blelloch *et. al.* [11] describes the CONTROLLED-PDF scheduling as follows – the L2-supernodes are scheduled one at a time following the 1DF schedule. Within each L2-supernode the L1-supernodes are scheduled based on the PDF schedule using all $p$ processors. Each L1-supernode scheduled is entirely executed on that processor. After all L1-supernodes of an L2-supernode have been executed, the scheduler moves on to the next L2-supernode. The number of cache misses is then proved to be within a constant factor of the sequential cache complexity through the following Lemma.

**Lemma 3.4** *Consider the multicore-cache model in which $C_2 \geq \alpha \cdot C_1$, where $\alpha \geq p$ is a constant. If a multicore hierarchical recursive algorithm incurs $Q_{L1}(n)$ L1 cache-misses and $Q_{L2}(n)$ L2 cache-misses under the CONTROLLED-PDF scheduler, then*

$$\text{(a) } Q_{L1}(n) = O(Q(C_1, n)), \text{ and (b) } Q_{L2}(n) = O(Q(C_2, n)). \quad (3.8)$$

*where $Q$ is the sequential cache complexity.*

*Proof*: See [11].

The following Lemma gives the parallel time complexity of the CONTROLLED-PDF scheduler.

**Lemma 3.5** *For an L2-supernode, let $T(n_2)$ denote the sequential time complexity, $T_p(n_2)$ denote the p processor parallel time complexity under the CONTROLLED-PDF scheduling and $T_\infty(n_2, n_1)$) denote the inherent parallel time complexity. Then we have,*

$$T_p(n_2) \leq \frac{T(n_2)}{p} + T_\infty(n_2, n_1) \tag{3.9}$$

*Proof*: The upper bound follows from the standard Brent-Graham scheduling. For details see [11] and [13].

For our computation model described in Section 3.3.1, Lemmas 3.4 and 3.5 hold for any level-*i* of the *d*-level hierarchy, because of the following assumption in our computation model (see Section 3.3.1) –

(1) An inclusive memory hierarchy implies that misses at level-*i* do not affect the misses at level > *i*.

(2) An inclusive memory hierarchy also implies that cache lines evicted at level-*i* are also evicted for level < *i*.

(3) $M_{i+1} > \alpha_i M_i$ where $\alpha_i \geq P_i$

3.4 Case Studies

We demonstrate the parallel algorithm design methodology using two widely used data parallel problems – matrix multiplication and the solution of Maxwell's equations using the Finite Difference Time Domain (FDTD) method. In each case, the data structures used are arrays – 2D for matrix multiplication, and 3D for FDTD. We state the equations describing the computations, highlight the opportunities for recursive array

decomposition, and demonstrate the parallel pebbling of the associated DAGs under the weighted vertex ($\Omega^s$) pebbling strategy. Further, for these two cases, we derive problem specific bounds for communication (cache) and space complexities.

### 3.4.1 Matrix Multiplication

Matrix multiplication refers to the standard dense matrix multiplication algorithm for multiplying two $n \times n$ square matrices $A$ and $B$ to get a result matrix $C = AB$. The computational complexity of the algorithm is $O(n^3)$ while the data access time and space requirements are $O(n^2)$. As shown in Figure 3.2, a possible geometric decomposition of the problem involves recursive binary decompositions of the $A$, $B$ and $C$ matrices into sub-matrices (blocks) along both the dimensions.

Level-($i+1$) DAGs and the level-$i$ sub-DAGs for a possible two level decomposition of the $A$, $B$, and $C$ matrices are shown in Figure 3.3. Note from Figure 3.3 that all the $C_i$ level-($i+1$) DAGs can be computed independently but each share data with the other sub-DAGs.

In formulating the weighted pebbling strategy for level-($i+1$) DAGs of Figure 3.3, we first assign weights to the individual vertices as outlined in Section 3.3.5. Figure 3.4(a) shows the initial weighting of the vertices of the level-($i+1$) DAGs of Figure 3.3 under $\Omega^s$ along with the initial assignment of pebbles. Figure 3.4(b) shows an intermediate step in the pebbling of the DAGs associated with the computation of $C_2$ where all the inputs $B_0$ to $B_3$ have been pebbled. For pebbling the level-($i+1$) DAGs of Figure 3.3, a total of *6* pebbles are required with *16* read/write operations. Note that here we count both the reads and writes of the outputs (matrix $C$).

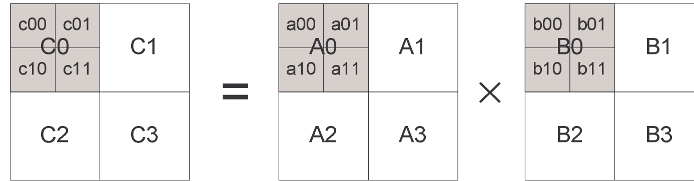



FIGURE 3.2: A 2-level geometric decomposition of $A$, $B$, and $C$ matrices.

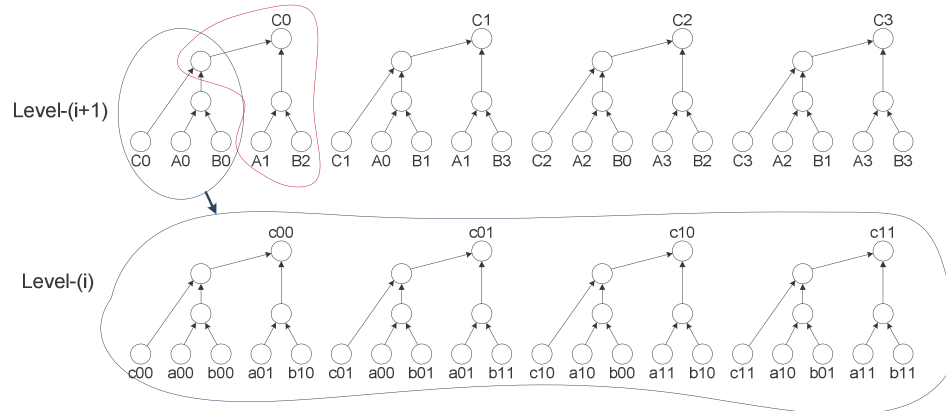



FIGURE 3.3: Illustrative level-($i$+$1$) and level-$i$ DAGs for matrix multiplication.

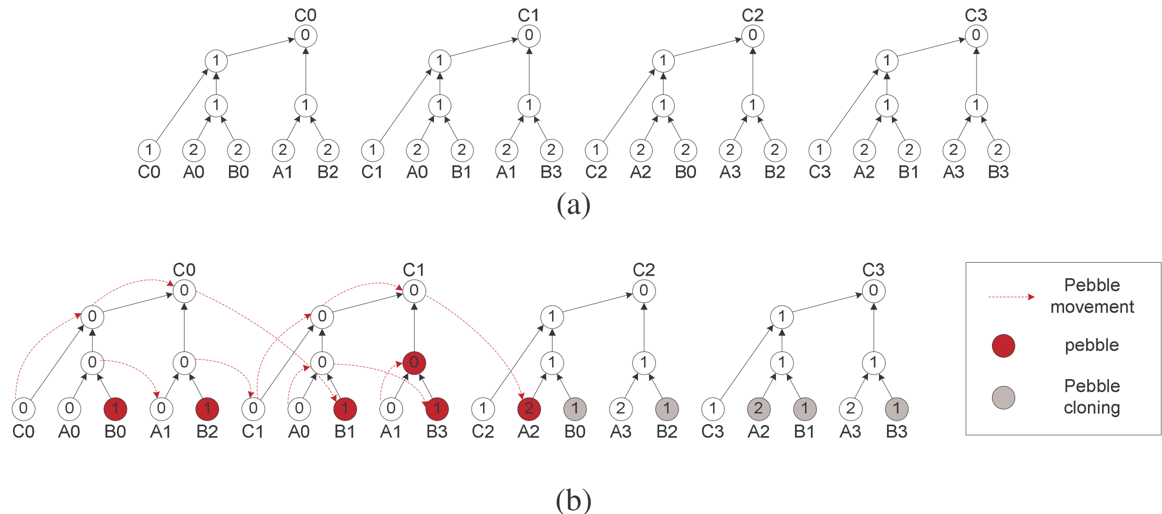



FIGURE 3.4: The weighted-vertex pebble game: (a) Initial vertex weight assignment for level-($i$+$1$) DAGs of Figure 3.2 under $\Omega^s$; (b) An intermediate step in the pebbling of the level-($i$+$1$) DAGs under $\Omega^s$.

We now derive bounds for space and cache complexities under $\Omega^s$ with $d=2$ (2-level memory hierarchy) and $P_2 = P_1 = 1$ (a single core) for matrix multiplication.

**Theorem 3.1** *Consider the multiplication of matrices A and B each of size n×n to generate an n×n matrix C. Let* $|B_i|$ *be the size of the geometrically decomposed square sub-matrix of A, B, and C at level-i such that* $|B_2| = n^2$, $|B_1| = b^2$, *and* $|B_0|=1$ *(single element) for a two level cache hierarchy. Let* $G_2$ *be a DAG with vertices of size* $|B_1|$ *with* $G_2$ *representing the computation of a single block of the C matrix at level-2, while let* $G_1$ *be a sub-DAG of* $G_2$ *with vertices of size* $|B_0|$ *representing the computations of a single element of the C matrix at level-1. The level-2 memory is considered infinite while level-1 memory is of size M. The number of two level memory read/write operations* $N_b^{R/W}$ *required in computing the* $b^2$ $G_1$*s representing the single block level multiplication of A and B satisfies the following lower bound:*

$$N_b^{R/W} \geq 4b^2 \tag{3.10}$$

*Proof*: From Lemma 3.1, each sub-DAG $G_1$ requires at least $|In(G_1)| + |Out(G_1)| = (2b+1) +1$ read and write operations, and there are $b^2$ $G_1$s. Thus, a total $b^2 \times (2b+2)$ read/write operations are required to compute $b^2$ $G_1s$. Each element of the input sub-matrix $A$ and $B$ can be reused at most $(b-1)$ times between the $b^2$ $G_1s$. Since there are $2b^2$ elements of sub-matrices $A$ and $B$, a total of at most $2b^2 \times (b-1)$ pebbles can be reused without any additional read/write operation between the $b^2$ sub-DAGs $G_1s$. Therefore, at least $b^2 \times (2b+2) - 2b^2 \times (b-1) = 4b^2$ read/write operations are required for $b^2$ $G_1s$.

**Theorem 3.2** *Consider the multiplication of matrices A and B each of size n×n to generate an n×n matrix C. Let* $|B_i|$ *be the size of the geometrically decomposed square sub-matrix of A, B, and C at level-i such that* $|B_2| = n^2$, $|B_1| = b^2$, *and* $|B_0|=1$ *(single*

*element) for a two level cache hierarchy. Let $G_2$ be a DAG with vertices of size $|B_1|$ with $G_2$ representing the computation of a single block of the C matrix at level-2, while let $G_1$ be a sub-DAG of $G_2$ with vertices of size $|B_0|$ representing the computations of a single element of the C matrix at level-1. The level-2 memory is considered infinite while level-1 memory is of size M. Then, the lower bound on the number of pebbles $S^s$ required to pebble $b^2$ $G_1$s under the weighted-vertex pebbling such that the cache read/write lower bound of Theorem 3.1 is satisfied is*

$$S^s \geq b^2 + b + 2 \text{ , where } b>2 \tag{3.11}$$

*Proof*: Consider a row major computation of *C*. Pebbling the $b^2$ $G_1$s requires at most $b^2$ pebbles for *A* (input), $b^3$ pebbles for *B* (input), *1* pebble for *C* (output), and *1* pebble for the intermediate node. However, the weighted-vertex pebbling strategy (See Section 3.3.5), assigns an initial weight $w=b$ for the input vertices representing elements of matrix *A* and *B* since these are shared (cloned) by *b* such $G_1$s. Further, $w=1$ is assigned to the intermediate vertices of $G_1$ and $w=0$ for the output node. The number of pebbles required is thus reduced by a factor of *b* - that is *b* pebbles for *A*, and $b^2$ pebbles for *B*. Therefore, the total number of pebbles needed is no more than $b^2 + b + 2$. Also, since the elements of the blocks of *A* and *B* are reused *b-1* times in the weighted pebbling strategy, following the arguments presented in the proof of Theorem 3.1, the resulting number of read/write operations is lower bounded by $4b^2$.

**Corollary 3.1** *As a consequence of Theorem 3.2 and Equation 3.7, the block size b satisfies the following upper bound:*

$$b \leq \left( \frac{-1 + \sqrt{4M - 3}}{2} \right) \tag{3.12}$$

**Theorem 3.3** *Consider the multiplication of matrices A and B each of size n×n to generate an n×n matrix C. Let* $|B_i|$ *be the size of the geometrically decomposed square sub-matrix of A, B, and C at level-i such that* $|B_2| = n^2$, $|B_1| = b^2$, *and* $|B_0| = 1$ *(single element) for a two level cache hierarchy. Let* $G_2$ *be a DAG with vertices of size* $|B_1|$ *with* $G_2$ *representing the computation of a single block of the C matrix at level-2, while let* $G_1$ *be a sub-DAG of* $G_2$ *with vertices of size* $|B_0|$ *representing the computations of a single element of the C matrix at level-1. The level-2 memory is considered infinite while level-1 memory is of size M. The total number of two level memory read/write operations* $N_{total}^{R/W}$ *required in the multiplication of A and B satisfies the following lower bound:*

$$N_{total}^{R/W} \geq 3\left(\frac{n^3}{b}\right) + b^2 \tag{3.13}$$

*Proof*: There are a total $(n/b)^3$ sets of $G_1$s at level-*1* with $b^2$ $G_1$s per set. From Theorem 3.1 the total number of read/write operations without considering data sharing between these sets is at least $4b^2 \times (n/b)^3 = 4(n^3/b)$. Since from Theorem 3.2 level-*1* can hold at least one complete block of size $b^2$, the total data sharing that is possible between the $G_1$ sets is at most $b^2 \times ((n/b)^3 - 1)$. Hence the lower bound on the number of read/write operations between level-1 and level-2 cache is at least $4(n^3/b) - b^2 \times ((n/b)^3 - 1) = 3(n^3/b) + b^2$.

We refer to [30] for extending our two-level cache tree to multilevel cache tree. We invoke the same assumptions as [30]; (a) that caches satisfy the inclusion property [39], which says that the data stored in cache at level-*i* are also stored in cache at level-(*i+1*), and (b) that if two elements belong to the same cache line at level-*i*, then they belong to the same line at level-(*i+1*). Additionally, we assume that $M_{i+1} > \alpha_i M_i$ where $\alpha_i \geq P_i$ (See Section 3.3.1). These assumptions ensure that each cache at level-(*i+1*) includes

at least the contents of $P_i$ caches at level-$i$. Therefore, we can also apply the weighted-vertex pebble game between level-$i$ and level-$(i+1)$ with the vertex size $|B_i|$.

### 3.4.2 Finite Difference Time Domain

Finite-Difference Time-Domain (FDTD) method is based on Yee's algorithm and computes the electric-field (**E**-field) and magnetic-field (**H**-filed) in both time and space domain. The characteristic features of our 3D-FDTD algorithm are (a) it is a computation and data-intensive problem performing $O(n^3)$ computations with $O(n^3)$ data access time and space requirement, (b) there is data dependency between **E-** and **H**-field computation in time domain, (c) there is no risk of a race condition for each field computation in space domain, and (d) a cell (*e.g.* $E_x(i,j,k)$) computation of each field in each direction refers to nearest-neighbors as *2*-point stencil communication pattern in the space domain. The following difference Equations 3.14 and 3.15 describe the FDTD computations for $E_x$ and $H_x$ components. Similar equations hold for the other $E_y$, $H_y$ and $E_z$, $H_z$.

$$E_x^{\left(t+\frac{1}{2}\right)}(i,j,k) = E_x^{\left(t-\frac{1}{2}\right)}(i,j,k) + \frac{\Delta t}{\epsilon_x(i,j,k)}\left(\frac{H_z^t(i,j,k)-H_z^t(i,j-1,k)}{\Delta y} - \frac{H_y^t(i,j,k)-H_y^t(i,j,k-1)}{\Delta z}\right) \quad (3.14)$$

$$H_x^{(t+1)}(i,j,k) = H_x^t(i,j,k) - \frac{\Delta t}{\mu}\left(\frac{E_y^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_y^{\left(t+\frac{1}{2}\right)}(i,j,k+1)}{\Delta z} - \frac{E_z^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_z^{\left(t+\frac{1}{2}\right)}(i,j+1,k)}{\Delta y}\right) \quad (3.15)$$

As shown in Figure 3.5, a possible geometric decomposition of the problem involves recursive binary decompositions of the 3D $E_x$, $E_y$, $E_z$ and $H_x$, $H_y$, $H_z$ matrices into sub-matrices (blocks) along both *x*-, *y*-, *z*-directions.

The DAGs describing the FDTD computation are shown in Figure 3.6. Note the data dependence between the **E**- and **H**-field DAGs.

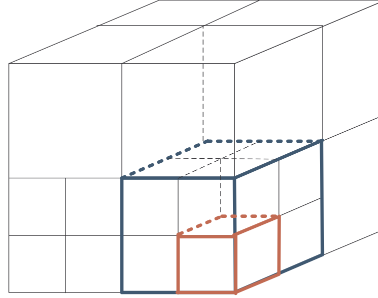


FIGURE 3.5: A 2-level geometric decomposition of the **E**- and **H**-field cubes.

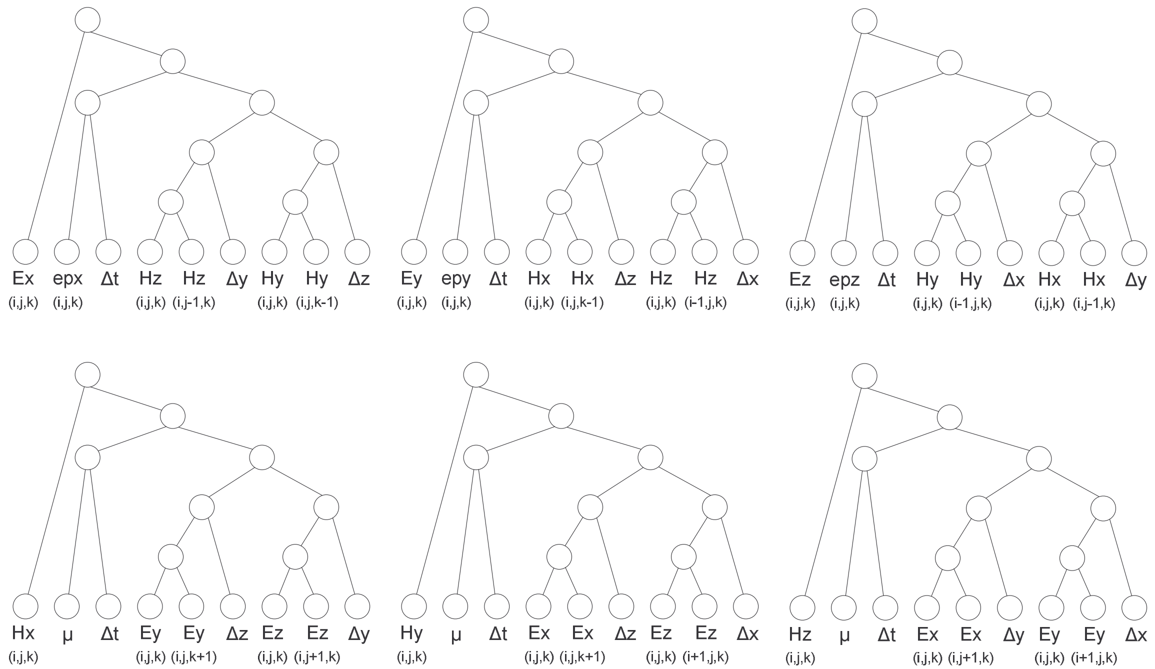



FIGURE 3.6: DAGs for FDTD: Note that there are *6* DAGs corresponding to $E_x$, $E_y$, $E_z$ and $H_x$, $H_y$, $H_z$.

We now derive bounds for space and cache complexities under $\Omega^s$ with $d=2$ (2-level memory hierarchy) and $P_2 = P_1 = 1$ (a single core) for 3D-FDTD. Note that we compute **E**-field first and then **H**-field since there is data dependency between the two in the time domain. Here we only consider **E**-field computations.

**Theorem 3.4** *Consider the* **E**-*field computation in a cube of size* $n^3$. *Let* $|B_i|$ *be the size of the geometrically decomposed sub-cube at level-i such that* $|B_2| = n^3$, $|B_1| = b^3$, *and* $|B_0|=1$ *(single cell) for a two level cache hierarchy. Let* $G_2$ *be a DAG with vertices of size* $|B_1|$ *with* $G_2$ *representing the computation of the* **E**-*field of a single block at level-2, while let* $G_1$ *be a sub-DAG of* $G_2$ *with vertices of size* $|B_0|$ *representing the computations of a single cell at level-1. The level-2 memory is considered infinite while level-1 memory is of size M. The number of two level memory read/write operations* $N_b^{R/W}$ *required in computing the* $3b^3$ $G_1$*s representing the single block level* **E**-*field computation satisfies the following lower bound:*

$$N_b^{R/W} \geq 12b^3 + 6b^2 + 4 \tag{3.16}$$

*Proof*: From Lemma 3.1, each sub-DAG $G_1$ requires at least $|In(G_1)| + |Out(G_1)| = 10$ read/write operations (See Figure 3.6). Since there are $3b^3$ sub-DAGs for each $E_x$, $E_y$ and $E_z$ computation, a total of at least $3\times b^3\times(10)$ read/write operations are required. The input constant parameter ($\Delta t$) can be reused at most $3b^3-1$ times while the input constant parameters $\Delta x$, $\Delta y$ *and* $\Delta z$ each can be reused at most $2b^3-1$ times. The input $H_x$ is used in the computation of both $E_y$ and $E_z$. Also, the input $H_x$ is shared between two adjacent cells and hence can be reused at least $b^3 + 2b^2(b-1) = 3b^3-2b^2$ times. The same argument applies for $H_y$ and $H_z$. Therefore, the total number of two level memory read/write operations is at least $30\times b^3 - (3b^3 - 1) - 3\times (2b^3 - 1) - 3\times (3b^3-2b^2) = 12b^3 + 6b^2 + 4$.

**Theorem 3.5** *Consider the* **E**-*field computations in a cube of size* $n^3$. *Let* $|B_i|$ *be the size of the geometrically decomposed sub-cube at level-i such that* $|B_2| = n^3$, $|B_1| = b^3$, *and* $|B_0|=1$ *(single cell) for a two level cache hierarchy. Let* $G_2$ *be a DAG with vertices of size* $|B_1|$ *with* $G_2$ *representing the computation of the* **E**-*field of a single block at level-2,*

*while let $G_1$ be a sub-DAG of $G_2$ with vertices of size $|B_0|$ representing the computations of a single cell at level-1. The level-2 memory is considered infinite while level-1 memory is of size M. Then, the lower bound on the number of pebbles $S^s$ required to pebble $3b^3$ $G_1$s under the weighted-vertex pebbling such that the cache read/write lower bound of Theorem 3.4 is satisfied is*

$$S^s \geq 2b^2 + 2b + 10 \quad (3.17)$$

*Proof:* Based on the degree of sharing, the weighted-vertex pebbling strategy (See section 3.3.5), assigns initial weights w to the $G_1$ vertices as follows – $w_{\Delta t} = 3b^3$, $w_{\Delta x} = w_{\Delta y} = w_{\Delta z} = 2b^3$, and $w_{Hx} = w_{Hy} = w_{Hz} = 4$. Without loss of generality, we assume the computation proceeds along the $z$-direction followed by $y$-direction, and $x$-direction. In that case, at least $b$ pebbles are required for $H_x$ due to data dependency along the $z$- (at least $2$ pebbles to hold values $H_x$ along the $z$-direction) and $y$-directions (at least b pebbles to hold $b$ $H_x$ values along the $z$-direction). For $H_y$ due to data dependency along the $z$- (at least $2$ pebbles to hold $H_y$ values along the $z$-direction) and $x$-direction (at least $b^2$ pebbles to hold $b^2$ $H_x$ values on the $yz$-plane) at least $b^2+2$ pebbles are required. For $H_z$ due to data dependency along the $x$- (at least $b^2$ pebbles to hold $b^2$ $H_z$ values on the $yz$-plane) and the $y$-direction (at least $b$ pebbles to hold $b$ $H_z$ values along the $x$-direction). Following the weight assignments of $\Delta t$, $\Delta x$, $\Delta y$ *and* $\Delta z$ described above a total of at least $4$ pebbles are required to hold these parameters. Similarly, at least $3$ pebbles are required for holding the *epx*, *epy*, and *epz* values, at least $2$ pebbles for storing intermediate vertices and one at least pebbles for the DAG output. Thus, summing up all the pebbles the calculation of $3b^3$ $G_1$ DAGs requires at least $2b^2+2b+10$ pebbles. Since the above pebble estimation

considers data sharing as described in Theorem 3.4, the resulting number of read/write operations is lower bounded by $12b^3 + 6b^2 + 4$.

**Corollary 3.2** *As a consequence of Theorem 3.5 and Equation 3.7, the block size b satisfies the following upper bound:*

$$b \le \left( \frac{-1 + \sqrt{M_1 - 8}}{2} \right) \tag{3.18}$$

**Theorem 3.6** *Consider the **E**-field computations in a cube of size $n^3$. Let $|B_i|$ be the size of the geometrically decomposed sub-cube at level-i such that $|B_2| = n^3$, $|B_1| = b^3$, and $|B_0| = 1$ (single cell) for a two level cache hierarchy. Let $G_2$ be a DAG with vertices of size $|B_1|$ with $G_2$ representing the computation of the **E**-field of a single block at level-2, while let $G_1$ be a sub-DAG of $G_2$ with vertices of size $|B_0|$ representing the computations of a single cell at level-1. The level-2 memory is considered infinite while level-1 memory is of size M. The total number of two level memory read/write operations $N_{total}^{R/W}$ required in the **E**-field computation satisfies the following lower bound:*

$$N_{total}^{R/W} \ge 12(n^3) + 3\left(\frac{n^3}{b}\right) + 4\left(\frac{n}{b}\right)^3 + 4b^2 \tag{3.19}$$

*Proof:* We have a total $(n/b)^3$ sets of sub-DAGs $G_1s$ at level-$1$ (total number of block computations). Since from Theorem 3.4, each $G_1$ requires at least $12b^3 + 6b^2 + 4$ read/write operations, a total of at least $12(n^3) + 6(n^3/b) + 4(n/b)^3$ read/write operations are required when data sharing between the blocks is not considered. Since at most $3b^2$ cells can be shared between any neighboring $b^3$-sets of $G_1$ (corresponding to a block), the total number of read/write operations is at least $12(n^3) + 6(n^3/b) + 4(n/b)^3 - 3b^2 \times ((n/b)^3 - 1) = 12(n^3) + 3(n^3/b) + 4(n/b)^3 + 4b^2$.

As shown in Figure 3.6, DAGs for **H**-field computations are similar as DAGs for **E**-field computations. Instead of three parameters *epx*, *epy* and *epz* for **E**-field computations, we consider only one parameter $\mu$ for **H**-field computations.

## 3.5 Conclusion

In order to derive scalable parallel algorithms for shared cache multicore machines with the properties described above, we propose the following design methodology in formulating space- and cache-efficient parallel algorithms for the geometrically decomposable problems –

(1) Develop a computational model that captures the salient features of the multicore processor under consideration. Although a variety of multicore architectures exist today, many of them have a shared cache architectural paradigm where a subset of processing cores share the cache hierarchy.

(2) Recursively decompose the data arrays representing the problem into sub-arrays such that overall solution is obtained by solving the problem on the sub-arrays. The depth of the recursive decomposition is determined by the depth of the cache hierarchy.

(3) Express the computation on the arrays as Directed Acyclic Graphs (DAG) $G(V,E)$ with **V** vertices and **E** edges. Here the vertices represent the input, output, and intermediate data while the edges represent the data dependencies.

(4) At each level of the memory hierarchy, map the DAGs representing the sub-array computations between the processing components sharing that level of the cache hierarchy such that the load is balanced.

(5) Between each pair of levels of the memory hierarchy formulate a weighted vertex red-blue pebbling strategy on the DAGs so as to determine the minimum number of pebbles that minimizes the memory read/write operations. The pebbling strategy essentially describes the parallel algorithm for the problem.

(6) Based on the computational model, estimate bounds on the space, compute, cache, and synchronization complexities.

(7) Using a suitable parallel programming model, implement the algorithm on the targeted multicore processor. Measure the performance and if necessary, tune the performance of the code using machine specific optimizations.

# CHAPTER 4: INTEGRATED DATA PREFETCHING AND CACHING IN MULTICORES

## 4.1 Introduction

To bridge the growing latency gap between the processing cores and the memory hierarchy, multicore processor designers have sought to exploit Compute Transfer Parallelism (CTP) where data transfer and computing are decoupled and can be executed in parallel. Compute transfer parallelism utilizes the architecture's ability to explicitly and independently sequence data transfer operations. Using application-level knowledge the software programmer can explicitly fetch large blocks of data ahead of time thus reducing resource idle time. A related technique in reducing processor stall time examined extensively in Chapter 3 is caching, where temporal and spatial data locality is exploited to minimize data movement (cache efficiency). Although the two techniques CTP and caching are architecturally independent, in practice there is a strong interaction. Given the limited sizes of the cache hierarchy, if data is pre-fetched too early, cache blocks needed in the near future could get evicted thus adversely affecting temporal locality. On the other hand, holding the blocks in cache for too long, negatively affects the ability to pre-fetch data. While previous work has considered both caching and pre-fetching in multicores separately, *in this chapter, we propose algorithm specific integrated software caching and pre-fetching strategies. Specifically, by using a simple model for data transfer, we attempt to theoretically determine the size and number of*

*read buffers implemented on machines with a limited size local memory and different compute and data transfer capabilities.*

The rest of the chapter is organized as follows – In Section 4.2 we examine related work done in integrated caching and pre-fetching for managing disk access latencies. We also review work done in pre-fetching on multicores. Section 4.3 briefly reviews the capability of the Cell Broadband Engine as an example of a multicore processor that supports compute transfer parallelism. In Section 4.4 we introduce a general purpose machine model and present conditions for when the total elapsed time is compute bound or data transfer bound. Section 4.5 and 4.6 illustrate our approach for integrating caching and prefetching in multicores using matrix multiplication and the FDTD algorithms as case studies. Section 4.7 concludes the chapter with our observations on the choice of an optimal buffering strategy when both pre-fetching and caching is considered.

## 4.2 Background

Integrated caching and pre-fetching techniques for disk systems have been reported in the literature since the mid 90s. Since the disk access latencies are far larger (~ 1 million times) the memory access latencies, pre-fetching is important in the hiding of expensive disk access latencies. Cao et. al. [17] introduced two integrated caching-prefetching algorithms - *Conservative* and *Aggressive* for single disks. The *Conservative* algorithm pre-fetches a missing block by evicting a cache block that is used as far as possible in the future. The *Aggressive* algorithm pre-fetches blocks as soon as possible. Specifically, a missing block in a computational sequence is pre-fetched if it can evict a cache block that is not used before the missing block. [43] and [44] extend these

algorithms to parallel disk systems. Albers and Buttner [2] generalized these algorithms by introducing a family of algorithms called Delay($d$) where the pre-fetch operation is delayed for $d$ time units.

Regarding pre-fetching in multicore processors, Chen et. al. [20] investigated the choice of buffering scheme and the size of the buffer on the IBM Cell processor. They introduced a DMA model that accounted for the set-up time latency and transfer rates. However, their work was focused on pre-fetching only and did not consider its interaction with caching. Sancho and Kerbyson [59] experimentally investigated the performance of double buffering on the quadcore AMD Opteron and the IBM Cell processor. They observed a performance improvement of *1.4*x and *2.2*x for the Opteron and the Cell processors when double buffering was employed for fictitious computing and data access patterns. Again, the effects of caching were not considered. Also, the reliance on empirical study without analytical performance modeling limits the extrapolation of their results to realistic data parallel benchmarks. Experimental studies on the performance bottlenecks in pre-fetching on the Cell architecture for an encryption/decryption workload were reported in [57].

## 4.3 Computation and Data Transfer Parallelism in the IBM Cell/B.E.

The architectural features of the IBM Cell/B.E. processor were introduced in Chapter 2. In this section we focus on the DMA capabilities of the Synergistic Processing Elements (SPEs). Each SPE consists of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). The SPU is a RISC-style processor with a 256 KB non-cached Local Store (LS) that holds program instructions and data. The SPU cannot access main memory directly, but it can issue DMA commands to the MFC to bring data into LS

or write computation results back to main memory [45]. The MFC includes a DMA controller, a Memory Management Unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE. The MFC supports naturally aligned DMA transfer size of 1, 2, 4 or 8 bytes and multiples of 16 bytes. The maximum size of a DMA transaction is 16 KB and the minimum recommended size is 128 bytes, the size of a cache line of the PowerPC processor. In addition, larger DMA transactions can be issued by DMA-list operation. The DMA-list transaction can be composed of up to 2,048 regular DMA transactions. The user can initiate multiple DMA transactions at a time that are queued for processing by the DMA engine [59]. The queue has 16 entries, and so the total number of outstanding DMA transactions can be $16 \times 2{,}048$ using DMA-list. Moreover, the SPE dual-pipelines allow the overlap of data transfer and computation, with one pipeline performing most of the arithmetic instructions while the other pipeline performing load and store instructions [47].

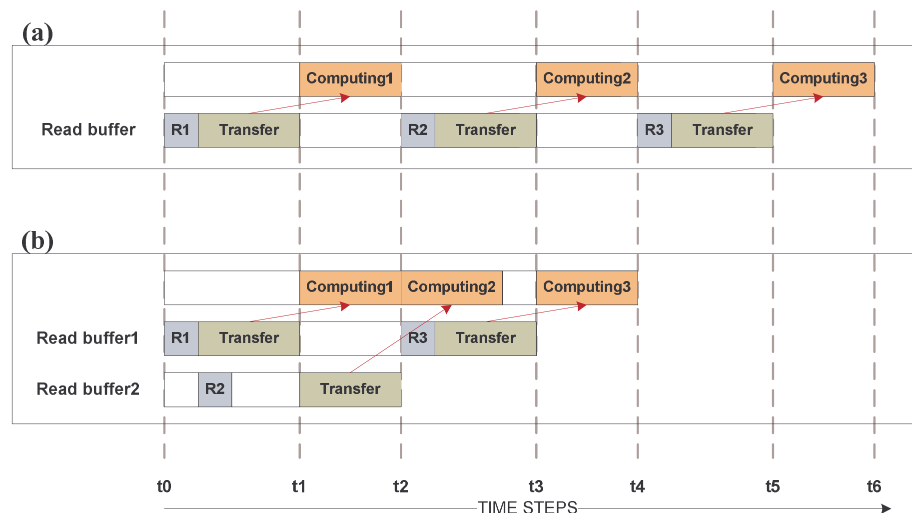



FIGURE 4.1: Simultaneous computing and DMA transfer: (a) Execution sequence for single read buffering; (b) Execution sequence for double read buffering.

Figure 4.1 illustrates a possible execution sequence highlighting the possible overlap of compute and DMA operations. In Figure 4.1(a), no overlapping is possible since DMA transfer as well as computing utilizes a single buffer. However, with an additional buffer, data transfer and computing operations can be overlapping from separate buffers with the buffers swapped in alternate cycles. For example as shown in Figure 4.1(b), while the computing operation uses the data in buffer 1, data is pre-fetched into buffer 2. The computing operation then utilizes data from buffer 2, while data is pre-fetched into buffer 1. Note that double buffering is only effective if the data to be pre-fetched in known in advance of the current computation. Also, additional buffers (*n*-buffering scheme) can be utilized for better overlap between the computations and data transfer. However, the limited size of the local store limits the number of additional buffers that can be employed.

## 4.4 Machine Model and General Bounds

We utilize a simple model of the data transfer operation of a machine fetching data from the main memory to its local memory. For a single data transfer operation, the total data transfer time ($T_{data}$) includes a setup time ($T_s$), the transfer time for one byte ($T_{mem}$) from the memory to the local store, and the number of bytes transferred ($B$).

$$T_{data} = T_s + B \times T_{mem} \tag{4.1}$$

The total elapsed time ($T_{total}$) for execution sequence with single buffering is

$$T_{total} = N_{data} \times T_{data} + N_{comp} \times BT_{comp} \tag{4.2}$$

where $N_{data}$ is a number of data transfer operations, $N_{comp}$ is a total number of computations, and $BT_{comp}$ is a computing time associated with a single $B$ data transfer. In a machine capable of concurrently scheduling compute and data transfer operations, with

double buffering scheme the total elapsed time ($T_{total}$) can be reduced by overlapping the two operations. Depending on the relative magnitudes of the machine specific parameters we can classify program execution into two regimes – data transfer bound and compute bound.

Let $BT_{comp}$ be the time required to perform computations on data of size $B$ transferred to the local memory from the main memory in time $BT_{mem}$ with setup time $T_s$. Then, the resulting program execution is data transfer bound if the data transfer operations can be scheduled back-to-back without a break. This condition holds when $BT_{comp} \leq BT_{mem} - T_s$. Such a back-to-back data transfer operations can help hide the setup time of an individual data transfer through overlap with subsequent data transfers. Note that data transfer bound execution results in stalling of the processor. Similarly, the program execution is compute bound if computations can be scheduled back-to-back without the processor stalling. This condition holds when $BT_{comp} \geq BT_{mem} + T_s$. Note that compute bound execution does not allow overlap of the setup time. If program execution is such that $BT_{mem} - T_s < BT_{comp} < BT_{mem} + T_s$, then stalls occur both in computation and data transfer. However, for large enough size $B$, the set up time $T_s$ can be ignored, and hence program execution is either only compute bound or only data transfer bound. Ignoring temporal locality of data if we let $N_{data}$ be the equal number of $N_{comp}$, then the total number of operations ($N_{oper}$) that can be overlapped between compute and data transfer operations is at most ($N_{comp}$ – 1) for data transfer bound and at most ($N_{data}$ – 1) for compute bound. Using the double buffering scheme, the overlap of computations with data transfer reduces the total elapsed time ($T_{total}$) as follows:

$$T_{total} \geq T_s + N_{data} \times (BT_{mem}) + BT_{comp} \qquad \textit{for data transfer bound} \qquad (4.3)$$

$$T_{total} \geq T_s + BT_{mem} + N_{comp} \times BT_{comp} \qquad \textit{for compute bound} \qquad (4.4)$$

Note that if the application has sufficient temporal locality between DMA transfers then caching can reduce the total elapsed time ($T_{total}$) by reducing the number of data transfers ($N_{data}$).

## 4.5 Matrix Multiplication

### 4.5.1 Theoretical Bounds

We consider the multiplication of two $n \times n$ matrices $A$ and $B$ to obtain an output $n \times n$ matrix $C$. The matrices are partitioned into blocks of size $b \times b$ such that all multiplication operations are carried out at the block level. As described in Chapter 3, blocking promotes cache efficient computation. Let the number of blocks in each matrix be $N \times N$ where $N$ is $(n/b)$. The example of $N=3$ is shown in Figure 4.2.

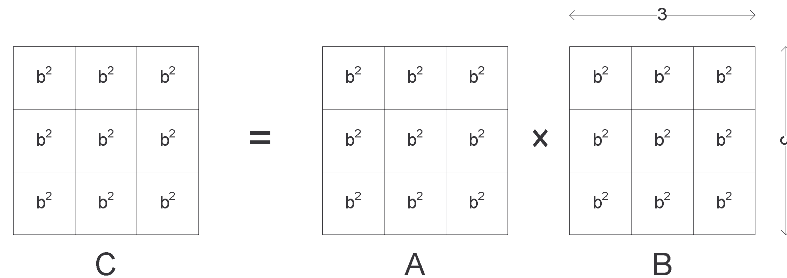



FIGURE 4.2: Matrix multiplication with $3 \times 3$ blocks.

To simplify our analysis we only consider the integrated caching and pre-fetching involving the reading of matrices $A$ and $B$. Let $M$ be the size of memory available for the input buffers and let $T_{comp}$ be the computation time required for a floating point operation (double or single precision). We consider the data transfer operation from the main memory to the local memory (cache) to involve a set-up time and a data transfer time and use Equation 4.1 to model the time taken to transfer a block of data. We derive upper and lower bounds for the total elapsed time for different buffering strategies. Here total

elapsed time is defined as the total time taken to multiply the two matrices and includes both the computation and the data transfer time.

Case I: Single buffer each for matrix *A* and *B*

Initially two data blocks of both *A* and *B* matrices have to be fetched before computation proceeds. Since only a single buffer is used for matrix *A* and *B*, we need to complete the computation involving the two blocks before the next data transfer operation is scheduled. Theorem 4.1 provides bounds for the total number of data transfers involved and Theorem 4.2 provides bounds on the total elapsed time.

**Theorem 4.1** *Let A and B be two N×N matrices each consisting of* $N^2$ *blocks. If the local memory (cache) is large enough to hold not more than a single block each of A and B, the upper and lower bounds for the total number of block-level data transfers* $N_{data}$ *is given by*

$$N^3 + N \leq N_{data} \leq 2N^3 \tag{4.5}$$

*Proof*: The upper bound is obtained by considering no reuse of the data present in the local memory. Thus two blocks of data needs to be fetched for each multiplication. Since there are $N^3$ such computations in multiplying matrices *A* and *B*, a total of not more than $2N^3$ blocks transfers block level data transfers is required. The lower bound is obtained by considering reuse of the data present in the local memory (temporal locality). We note that each block of matrix *A* is multiplied with an *N*-block row of matrix *B*. Thus, each block of matrix *A* can be re-used at most $N-1$ times. Since there are $N^2$ blocks of matrix *A*, a total of at most $N^2 \times (N-1)$ blocks can be reused without additional data fetches. Therefore, the total number of data transfer operations transfers of matrix *A* is at least $N^3 - (N^2 \times (N-1)) = N^2$. For matrix *B*, at most one block from each row of matrix *B* can be

reused. Since each row of matrix $B$ is required $N$ times for multiplying with one column of matrix $A$, at most $N\text{-}1$ blocks can be reused across all operations involving a single row of matrix $B$. There are $N$ rows of $B$, so at most $N\times(N\text{-}1)$ blocks can be reused. Hence the total number of block data transfer operations for matrix $B$ is at least $N^3 - (N\times (N\text{-}1)) = N^3 - N^2 + N$. Thus, the total number of data transfer operations for both $A$ and $B$ is at least $N^3 + N$.

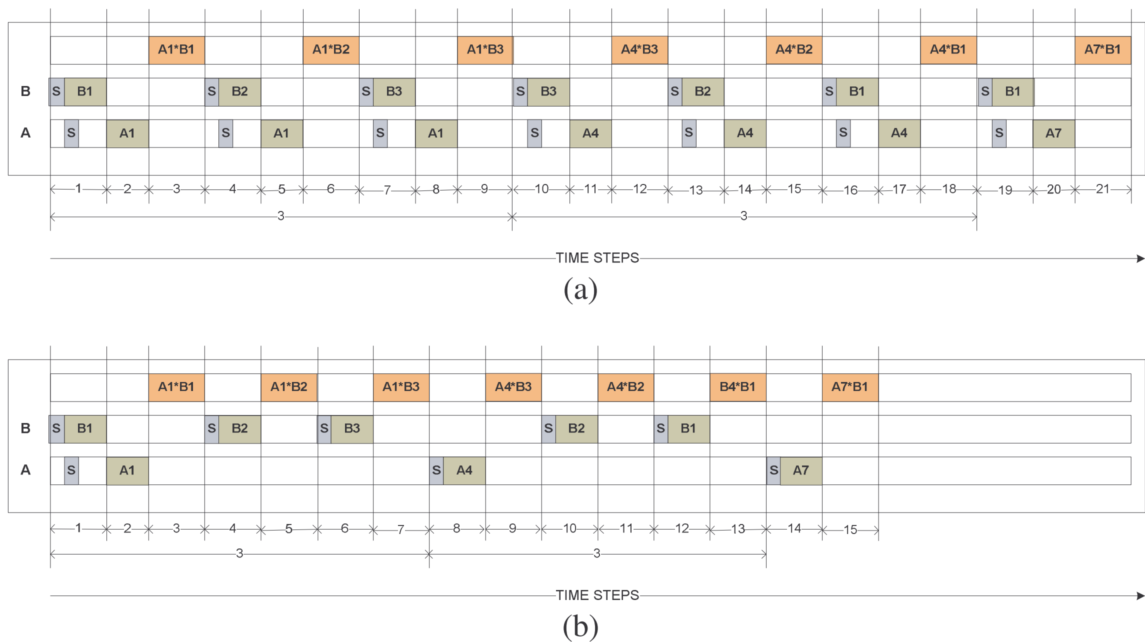



FIGURE 4.3: Simultaneous computing and data transfer for single buffer each for matrix $A$ and $B$: (a) Execution sequence obtained by considering no reuse of the data present in the local memory; (b) Execution sequence obtained by considering reuse of the data present in the local memory.

**Corollary 4.1** *If $T_s$ is the set up time for each data transfer operation, $T_{mem}$ the time required to transfer one data from the main memory to the local memory (cache), and $T_{comp}$ the time required for a single floating point operation, then the total elapsed time ($T_{total}$) satisfies the following upper and lower bounds,*

$$T_s \times N^3 + T_{mem} \times b^2 \times (N^3+N) + T_{comp} \times 2b^3 \times N^3 \leq T_{total} \leq T_s \times N^3 + T_{mem} \times b^2 \times (2N^3) + T_{comp} \times 2b^3 \times N^3 \quad (4.6)$$

*Proof:* The upper bound is obtained by considering no reuse of the data present in the local memory. As shown in Figure 4.3(a), single buffering of *A* and *B* permits no overlap of data transfer and computation, the total elapsed time is at most the sum of the times required for computation ($N^3$) and data transfer ($2N^3$, Theorem 4.1). However, the set up time for transfer of a block of *B* can be overlapped with the data transfer of block of *A*, the setup need be done at most $N^3$ times. A similar argument holds for the lower bound as well with the lower bound on the number of data transfers (see Figure 4.3(b)) given by Theorem 4.1.

Case II: Single buffer for matrix *A* and double buffer for matrix *B*

The bound on the number of data transfers of matrix *A* is same as Case I since only a single buffer is used for matrix *A*. However, the number of data transfers of matrix *B* can potentially be reduced due to the double buffering of *B*. Moreover, the double buffering allows for overlap of data transfer and computation times, potentially reducing the total elapsed time.

**Theorem 4.2** *Let A and B be two N×N matrices each consisting of $N^2$ blocks. If the local memory (cache) is large enough to hold not more than a single block of A and two blocks of B, the upper and lower bounds for the total number of block-level data transfers $N_{data}$ is given by*

$$N^3 - N^2 + 2N \leq N_{data} \leq 2N^3 \quad (4.7)$$

*Proof:* The upper bound on the number of data transfers for matrix *A* and *B* considering no reuse of data is same as Theorem 4.1. The lower bound on the number of data

transfers for matrix $A$ considering data reuse is $N^2$ and follows the same argument given in the proof of Theorem 4.1. Double buffering for matrix $B$, enables the sharing of two blocks of data for every row of $B$. Since each row of matrix $B$ is required $N$ times for multiplying with one column of matrix $A$, at most $2\times(N-1)$ blocks can be re-used across all operations involving a single row of matrix $B$. There are $N$ rows of $B$, so at most $2\times N\times(N-1)$ blocks can be reused. Hence the total number of block data transfer operations for matrix $B$ is at least $N^3 - (2\times N\times(N-1)) = N^3 - 2N^2 + 2N$. Thus, the total number of data transfer operations for both $A$ and $B$ is at least $N^2 + (N^3 - 2N^2 + 2N) = N^3 - N^2 + 2N$.

**Theorem 4.3** *Let A and B be two N×N matrices each consisting of $N^2$ blocks. If the local memory (cache) is large enough to hold not more than a single block of A and two blocks of B, the lower bound for the total number of operations $N_{oper}$ that cannot be overlapped is given by*

$$N_{oper} \geq N^3 + N^2 + 1 \tag{4.8}$$

*Proof:* We note that double buffering of $B$ possibly allows for overlapping of all data transfer operations of $B$ with computations operations except for the first block. However, since $A$ is single buffered at least $N^2$ block data transfer operations are required for $A$. Since there are a total of $N^3$ computations, the total number of operations is at least $N^3 + N^2 + 1$.

**Corollary 4.2** *If $T_s$ is the set up time for each data transfer operation, $T_{mem}$ the time required to transfer one data from the main memory to the local memory (cache), and $T_{comp}$ the time required for a single floating point operation, then the total elapsed time ($T_{total}$) satisfies the following upper and lower bounds,*

Data Transfer Bound: $b^2 \times T_{mem} > 2b^3 \times T_{comp} + T_s$

$$T_s \times N^2 + T_{mem} \times b^2 \times (N^3 - N^2 + 2N) + T_{comp} \times 2b^3 \times (2N^2 - 2N+1) \leq T_{total} \leq T_s \times N^3 + T_{mem} \times b^2 \times (2N^3) + 2b^3 \times T_{comp} \quad (4.9)$$

Compute Bound: $2b^3 \times T_{comp} > b^2 \times T_{mem} + Ts$

$$T_s \times N^2 + T_{mem} \times b^2 \times (N^2+1) + T_{comp} \times 2b^3 \times N^3 \leq T_{total} \leq T_s \times N^3 + T_{mem} \times b^2 \times (N^3 + 1) + 2b^3 \times T_{comp} \times N^3 \quad (4.10)$$

*Proof:* The upper bound for the total elapsed time is obtained by considering no reuse of the data present in the local memory. For the data transfer bound case, at most $N^3 - 1$ of a total of $N^3$ block multiplications can be overlapped with the $2N^3$ data transfer operations. The total elapsed time is thus at most the sum of the times required for the maximum number of data transfer operations (upper bound of Theorem 4.2), the set up time for the single buffered blocks of $A$**,** and the computation time for one block. In the compute bound case, at most $N^3$ $-1$ data transfer operations can be overlapped with the $N^3$ computations. Thus, the total elapsed time consists of at least $2N^3 - (N^3 - 1) = N^3 + 1$ data transfers, $N^3$ computations, and $N^3$ set up time for blocks of $A$.

The lower bound is obtained by considering reuse of the data present in the local memory. From Theorem 4.2 the number of block-level data transfer operations is at least $N^3 - N^2 + 2N$, while from Theorem 4.3 the total number of non-overlapped operations is at least $N^3 + N^2 + 1$. For the data transfer bound case (see Figure 4.4(a)), the number of compute operations that cannot be overlapped with is at least $(N^3 + N^2 + 1) - (N^3 - N^2 + 2 \times N) = (2N^2 - 2N + 1)$. On the other hand, in the compute bound case (see Figure 4.4(b)), the number of data transfer operations that cannot be overlapped with computations is $(N^3 + N^2 + 1) - N^3$. In either case, the single buffering of $A$ requires at least $N^2$ setup times.

And in data transfer bound case, $N^2–N$ additional setup times are required. The lower bound is given by the sum of the setup time, non-overlapped (compute bound) data transfer time and non-overlapped (data transfer) compute time.

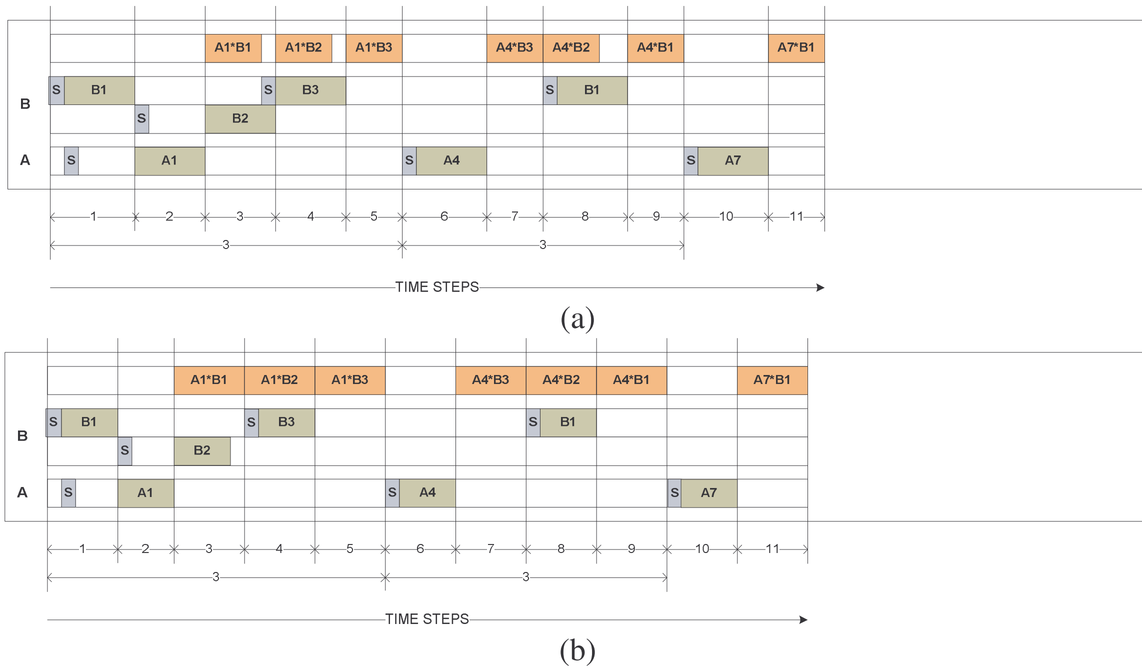



FIGURE 4.4: Simultaneous computing and data transfer for single buffer for matrix $A$ and double buffer for matrix $B$ obtained by considering reuse of the data present in the local memory: (a) Data transfer bound case; (b) Compute bound case.

Case III: Double buffers for matrix $A$ and $B$

The bound on the number of data transfers is same as given by Theorem 4.2 since the added double buffering of $A$ cannot reduce the total number of data transfer operations required to less than $N^2$. However, double buffering of $A$ the increases possibility of overlap between computations and the data transfer reducing the total number of operations that cannot be overlapped.

**Theorem 4.4** *Let A and B be two N×N matrices each consisting of $N^2$ blocks. If the local memory (cache) is large enough to hold at most two blocks each of A and B, the lower bound for the total number of operations $N_{oper}$ that cannot be overlapped is given by*

$$N^3 + 2 \leq N_{oper} \tag{4.11}$$

*Proof:* We note that double buffering of *A* and *B* possibly allows for overlapping of all data transfer operations of *A* and *B* with computations operations except for the first block of *A* and *B*. Since there are a total of $N^3$ computations, the total number of operations is at least $N^3 + 2$.

**Corollary 4.2** *If $T_s$ is the set up time for each data transfer operation, $T_{mem}$ the time required to transfer a single data from the main memory to the local memory (cache), and $T_{comp}$ the time required for a single floating point operation, then the total elapsed time ($T_{total}$) satisfies the following upper and lower bounds,*

Data Transfer Bound: $(b^2T_{mem} > 2b^3T_{comp} + T_s)$

$$T_s + T_{mem}\times b^2\times(N^3 - N^2 + 2N) + T_{comp}\times 2b^3\times(N^2 - 2N+2) \leq T_{total} \leq T_s + T_{mem}\times b^2\times(2N^3) + 2b^3\times T_{comp} \tag{4.12}$$

Compute Bound: $(2b^3T_{comp} > b^2T_{mem} + T_s)$

$$T_s + 2\times T_{mem}\times b^2 + T_{comp}\times 2b^3\times N^3 \leq T_{total} \leq T_s + T_{mem}\times b^2\times(N^3 + 1) + 2b^3\times T_{comp} \times N^3 \tag{4.13}$$

*Proof:* The upper bound for the total elapsed time is obtained by considering no reuse of the data present in the local memory. For the data transfer bound case, at most $N^3 - 1$ of a total of $N^3$ block multiplications can be overlapped with the $2N^3$ data transfer operations. Also, the double buffering of both *A* and *B* implies that only the initial set time involved in the data transfer of the first block cannot be overlapped. The total elapsed time is thus

at most the sum of the times required for the maximum number of data transfer operations (upper bound of Theorem 4.2), the set up time for a single block, and the computation time for one block. In the compute bound case, at most $N^3 - 1$ data transfer operations can be overlapped with the $N^3$ computations. Thus, the total elapsed time consists of at least $2N^3 - (N^3 - 1)$ data transfers, $N^3$ computations, and the set up time for a single block.

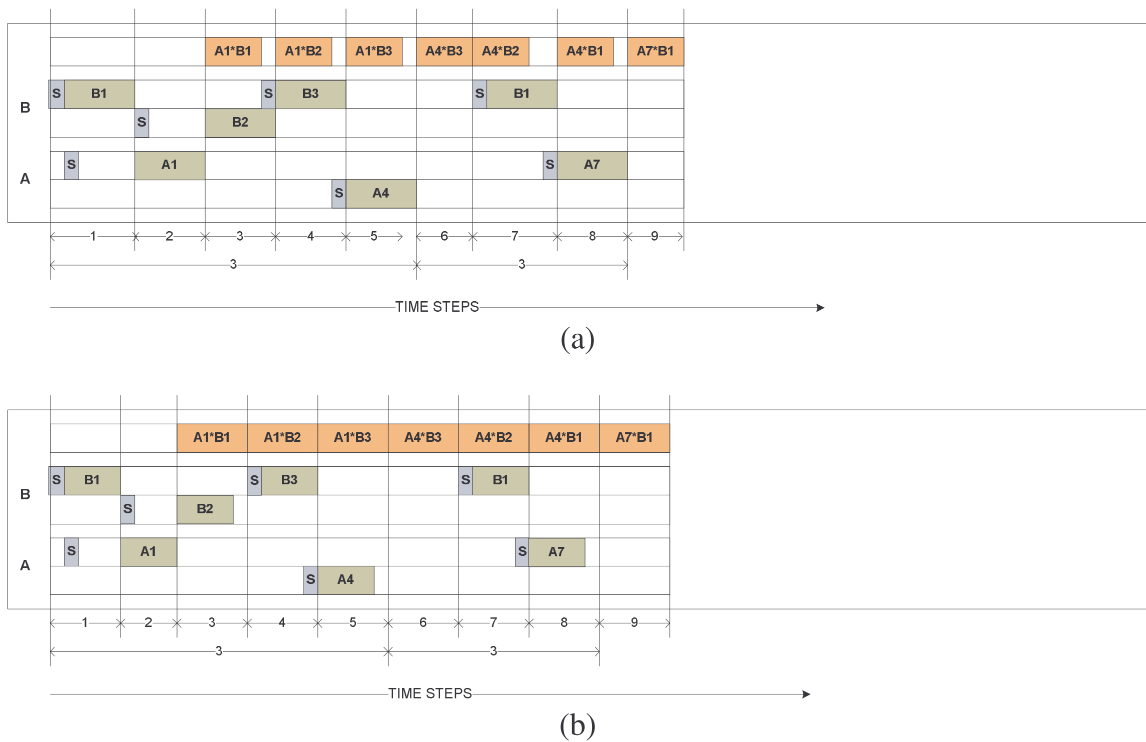



FIGURE 4.5: Simultaneous computing and data transfer for double buffer for both matrix *A* and *B* obtained by considering reuse of the data present in the local memory: (a) Data transfer bound case; (b) Compute bound case.

The lower bound is obtained by considering reuse of the data present in the local memory. From Theorem 4.2 the number of data transfer operations is at least $N^3 – N^2 + 2N$ while from Theorem 4.4 the total number of non-overlapped operations is at least $N^3 + 2$. For the data transfer bound case (see Figure 4.5(a)), the number of compute operations

that cannot be overlapped with data transfer is at least $(N^3+2) - (N^3 - N + 2N) = (N^2 - 2N + 2)$. On the other hand, in the compute bound case (see Figure 4.5(b)), the number of data transfer operations that cannot be overlapped with computations is $(N^3 + 2) - N^3$. In either case, the double buffering of $A$ and $B$ requires a setup time only for the data transfer of the first block. The lower bound is given by the sum of the setup time, non-overlapped (compute bound) data transfer time and non-overlapped (data transfer) compute time.

### 4.5.2 Discussion

A fair comparison between the different buffering schemes presented above requires the expressing the total elapsed time in terms of the problem size $n$ (number is single precision floats) and the size of the local storage $M$ (expressed in terms of number of single precision floats). Although use of a higher order buffering scheme enables a better overlap between the computations and the data transfer, with a fixed size memory, the block size is smaller, resulting in lesser opportunities for exploiting temporal locality. Note that buffer size is same as the block size. We assume the maximum size of a single buffer to be $M$ and scale the buffer size down by the number of buffers that needs to be maintained in memory. The theoretical lower bounds for the elapsed time can then be expressed as follows –

Case I: Single buffer each for matrix $A$ and $B$

$$T_s \times \left(\frac{\sqrt{2}}{\sqrt{M}}\right)^3 \times n^3 + T_{mem} \times \left(\frac{\sqrt{2}}{\sqrt{M}} \times n^3 + \frac{\sqrt{M}}{\sqrt{2}} \times n\right) + T_{comp} \times (2 \times n^3) \quad (4.14)$$

Case II: Single buffer for matrix *A* and double buffer for matrix *B*

Data transfer bound:

$$T_s \times \left( \left(\frac{\sqrt{3}}{\sqrt{M}}\right)^2 \times n^2 \right) + T_{mem} \times \left( \frac{\sqrt{3}}{\sqrt{M}} \times n^3 - n^2 + 2 \times \frac{\sqrt{M}}{\sqrt{3}} \times n \right)$$

$$+ T_{comp} \times \left( 2 \times \left( 2 \times \frac{\sqrt{M}}{\sqrt{3}} \times n^2 - 2 \times \left(\frac{\sqrt{M}}{\sqrt{3}}\right)^2 \times n + \left(\frac{\sqrt{M}}{\sqrt{3}}\right)^3 \right) \right) \quad (4.15)$$

Compute bound:

$$T_s \times \left(\frac{\sqrt{3}}{\sqrt{M}}\right)^2 \times n^2 + T_{mem} \times \left( n^2 + \left(\frac{\sqrt{M}}{\sqrt{3}}\right)^2 \right) + T_{comp} \times (2n^3) \quad (4.16)$$

Case III: Double buffers for matrix *A* and *B*

Data transfer bound:

$$T_s + T_{mem} \times \left( \frac{\sqrt{4}}{\sqrt{M}} \times n^3 - n^2 + 2 \times \frac{\sqrt{M}}{\sqrt{4}} \times n \right)$$

$$+T_{comp} \times \left( 2 \times \left( \frac{\sqrt{M}}{\sqrt{4}} \times n^2 - 2 \times \left(\frac{\sqrt{M}}{\sqrt{4}}\right)^2 \times n + 2 \times \left(\frac{\sqrt{M}}{\sqrt{4}}\right)^3 \right) \right) \quad (4.17)$$

Compute bound:

$$T_s + T_{mem} \times \left( 2 \times \left(\frac{\sqrt{M}}{\sqrt{4}}\right)^2 \right) + T_{comp} \times (2n^3) \quad (4.18)$$

*Performance Evaluation on the Cell B.E.:*

We evaluate the lower bound performance for the three buffering schemes discussed above on the Cell BE. For the Cell/B.E. the setup time is on the order of *130 ns* [20], while $T_{mem}$ is the order of *0.018 ns* per single precision float (the theoretical peak data bandwidth of *204.8* GB/s). Note that Equation 4.1 is an approximation of the DMA transfer time of the Cell/B.E. since the setup time depends on the data alignment in memory and the transfer time depends on the congestion in the network. $T_{comp}$ is the order

of *0.036 ns* for one single precision floating operation calculated from theoretical peak 25.6 GFLOPS performance per SPU. We assume the local memory size *M* available for computation is *0.25 million single precision floats* (100 KB) per LS. For these machine parameters matrix multiplication is a compute bound application. Figure 4.6 shows the theoretical lower bound for the three different buffering schemes. As shown in Figure 4.6, all three buffering schemes show similar performance with scaling of problem size with double buffering of *A* and *B* (Case III) showing less than 1% performance improvement than the single buffering of *A* and double buffering of *B* (Case II). This surprising result can be explained by the fact that on the Cell/B.E. the compute time is about 99% of the total elapsed time.

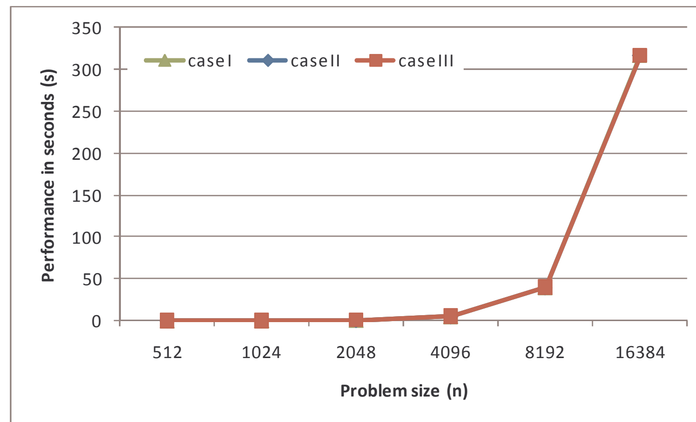



FIGURE 4.6: Theoretical lower bounds for matrix multiplication on IBM Cell/B.E.

## 4.6 Finite Difference Time Domain (FDTD)

### 4.6.1 Theoretical Bounds

We consider the **E**-field computation in cubes of size $n^3$. The cubes are partitioned into blocks of size $b^3$ such that all **E**-field operations are carried out at the block level. As

described in Chapter 3, blocking promotes cache efficient computation. Let the number of blocks in a cube be $N^3$ where $N$ is $n/b$.

We analyze the integrated caching and pre-fetching involving the reading of the nine parameters $E_x$, $E_y$, $E_z$, $H_x$, $H_y$, $H_z$, *epx*, *epy* and *epz*. To simplify our analysis we only consider sharing of boundary data within a block and not between blocks. Note that the constant parameters $\Delta t$, $\Delta x$, $\Delta y$ *and* $\Delta z$ are not considered in our analysis since they need only be fetched once and stored in the local memory. Let $M$ be the size of memory available for the input buffers and let $T_{comp}$ be the computation time required for a floating point operation (double or single precision). We consider the data transfer operation from the main memory to the local memory (cache) to involve a setup time and a data transfer time and use Equation 4.1 to model the time taken to transfer a block of data. We derive upper and lower bounds for the total elapsed time for different buffering strategies. Here the total elapsed time is defined as the total time taken to both **E**-filed computations and the data transfer time.

Case I: Single buffer for **E**-field computations

Initially at least four data blocks of data have to be fetched before computation of either $E_x$, or $E_y$ or $E_z$ can proceed. Since only a single buffer is used for each data set, we need to complete the computation involving the data set before the next data set transfer operation is scheduled. Theorem 4.5 provides bounds for the total number of data transfers involved and Corollary 4.3 provides bounds on the total elapsed time.

**Theorem 4.5** *Let $E_x$, $E_y$, $E_z$, $H_x$, $H_y$, $H_z$, epx, epy and epz each be $b^3$ sized blocks with a total of $9N^3$ such blocks. If the local memory (cache) is large enough to hold not more*

*than a single block each of* $E_x$*,* $E_y$*,* $E_z$*,* $H_x$*,* $H_y$*,* $H_z$*, epx, epy and epz, the upper and lower bounds for the total number of block-level data transfers* $N_{data}$ *is given by*

$$9N^3 \leq N_{data} \leq 12N^3 \tag{4.19}$$

*Proof:* The upper bound is obtained by considering no reuse of the data present in the local memory to compute $E_x$, $E_y$ and $E_z$ (see Figure 4.7(a)). Here four blocks of data needs to be fetched for each computation. Since there are at most $N^3$ such computations in computing each of $E_x$, $E_y$ and $E_z$, a total of not more than $12N^3$ block level data transfers are required. The lower bound is obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$ and $E_z$ computations (see Figure 4.7(d)). We note that each of $H_x$, $H_y$ and $H_z$ data blocks are used at most twice to compute $E_x$, $E_y$ and $E_z$. Hence, at most $3N^3$ blocks can be reused. Thus, the total number of block data transfer operations for **E**-field computations is at least $12N^3 - 3N^3 = 9N^3$.

**Corollary 4.3** *If* $T_s$ *is the set up time for each data transfer operation,* $T_{mem}$ *the time required to transfer one data from the main memory to the local memory (cache), and* $T_{comp}$ *the time required for a single floating point operation***,** *then the total elapsed time (*$T_{total}$*) satisfies the following upper and lower bounds,*

$$N^3 \times (T_s + 9b^3 \times T_{mem} + 27b^3 \times T_{comp}) \leq T_{total} \leq N^3 \times (3T_s + 12b^3 \times T_{mem} + 27b^3 \times T_{comp}) \tag{4.20}$$

*Proof:* The upper bound is obtained by considering no reuse of the data present in the local memory. As shown in Figure 4.7(a), single buffering of data transfer permits no overlap of data transfer and computation, the total elapsed time is at most the sum of the times required for computation ($9 \times 3N^3$) and data transfer ($12N^3$, Theorem 4.5). However, the setup time for transfer of a set of blocks can be overlapped with the data transfer of

blocks, the setup need be done at most $3N^3$ times. A similar argument holds for the lower bound as well with the lower bound on the number of data transfers (see Figure 4.7(d)) given by Theorem 4.5. A set time of at least $N^3$ is required if all the data required for computing $E_x$, $E_y$, and $E_z$ are fetched initially.

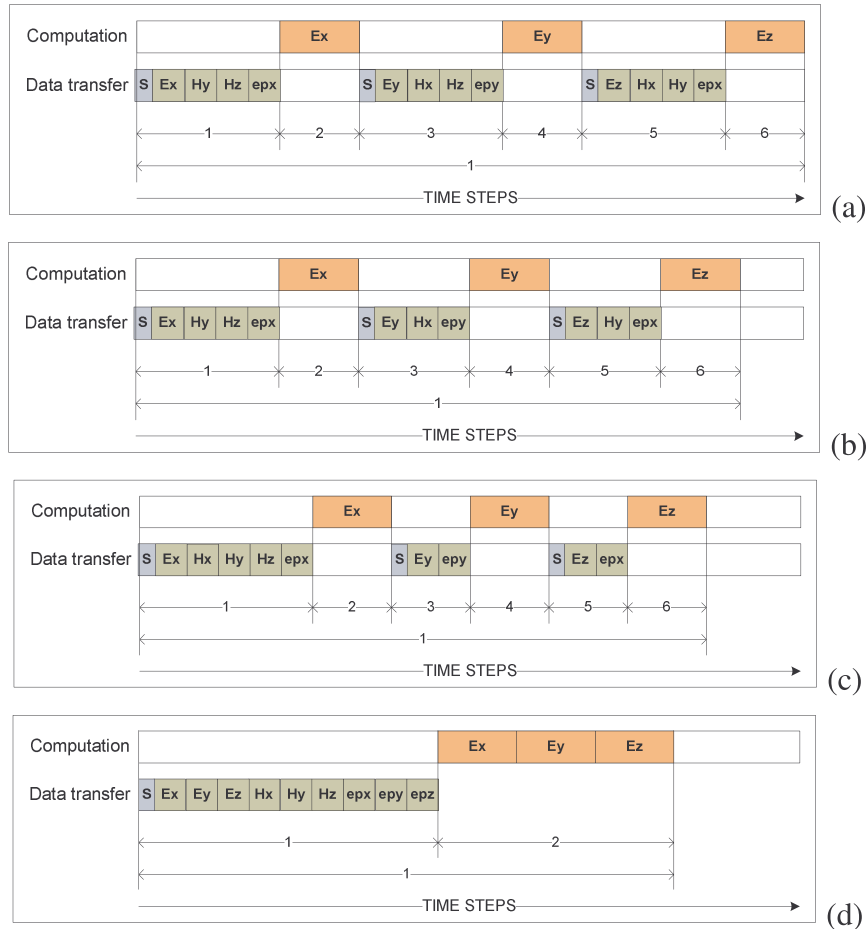
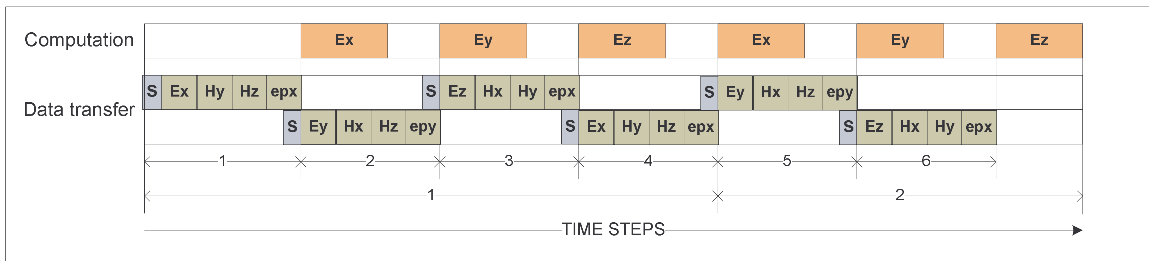


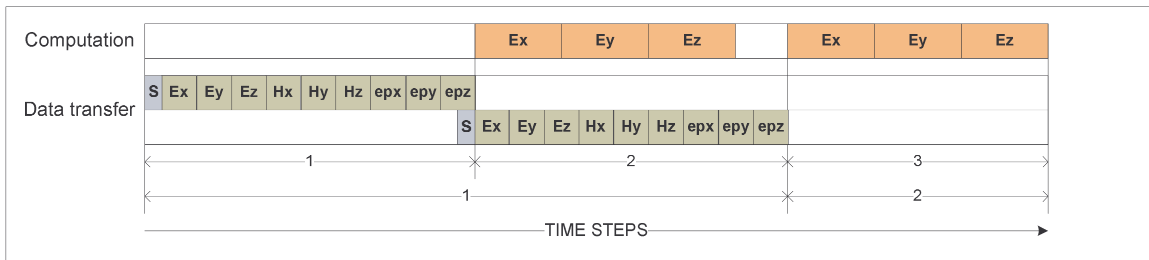

FIGURE 4.7: Simultaneous computing and data transfer for single buffer for **E**-field computation: (a) Execution sequence obtained by considering no reuse of the data present in the local memory. This scheme requires the storage of *4* blocks of data in the local memory; (b) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$ and $E_y$, $E_z$. This scheme requires the storage of *4* blocks of data in the local memory; (c) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$, and $E_z$. This scheme requires the storage of *5* blocks of data in the local memory; (d) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$, and $E_z$ but with all the data fetched initially. This scheme requires the storage of *9* blocks of data in the local memory.

Case II: Double buffers for **E**-field computations

The bound on the number of data transfers is same as given by Theorem 4.5 since the added double buffering cannot reduce the total number of data transfer operations required to less than $N^3$. However, double buffering increases possibility of overlap between computations and the data transfer reducing the total number of non-overlapped operations.




(a)




(b)

FIGURE 4.8: Simultaneous computing and data transfer for double buffers for **E**-field computation as data transfer bound cases: (a) Execution sequence obtained by considering no reuse of the data present in the local memory. This scheme requires the storage of *4* blocks of data in the local memory; (b) Execution sequence obtained by considering reuse of the data present in the local memory between $E_x$, $E_y$, and $E_z$ but with all the data fetched initially. This scheme requires the storage of *9* blocks of data in the local memory.

**Theorem 4.6** *Let $E_x$, $E_y$, $E_z$, $H_x$, $H_y$, $H_z$, epx, epy and epz each be $b^3$ sized blocks with a total of $9N^3$ such blocks. If the local memory (cache) is large enough to hold not more*

*than two blocks each of $E_x$, $E_y$, $E_z$, $H_x$, $H_y$, $H_z$, epx, epy and epz, the lower bound for the total number of operations $N_{oper}$ that cannot be overlapped is given by*

$$N^3 + 1 \leq N_{oper} \tag{4.21}$$

*Proof:* We note that double buffering possibly allows for overlapping of all data transfer operations with computations except for the initial set of blocks. Since there are a total of $N^3$ **E**-field computations, the total number of operations with maximal overlap between computing and data transfer is at least $N^3 + 1$.

**Corollary 4.4** *If $T_s$ is the set up time for each data transfer operation, $T_{mem}$ the time required to transfer data from the main memory to the local memory (cache), and $T_{comp}$ the time required for a single floating point operation***,** *then the total elapsed time ($T_{total}$) satisfies the following upper and lower bounds,*

Data Transfer Bound:

*Case A (max. data sharing): ($9b^3T_{mem} > 27b^3T_{comp} + T_s$)*

$$T_{total} \geq T_s + T_{mem} \times 9b^3 \times N^3 + T_{comp} \times 27b^3 \tag{4.22}$$

*Case B (no data sharing): ($4b^3T_{mem} > 9b^3T_{comp} + T_s$)*

$$T_{total} \geq T_s + T_{mem} \times 12b^3 \times N^3 + T_{comp} \times 9b^3 \tag{4.23}$$

Compute Bound:

*Case A: ($27b^3T_{comp} > 9b^3T_{mem} + T_s$)*

$$T_{total} \geq T_s + T_{mem} \times 9b^3 + T_{comp} \times 27b^3 \times N^3 \tag{4.24}$$

*Case B: ($9b^3T_{comp} > 4b^3T_{mem} + T_s$)*

$$T_{total} \geq T_s + T_{mem} \times 4b^3 + T_{comp} \times 27b^3 \times N^3 \tag{4.25}$$

*Proof:* To achieve maximal data sharing between the blocks (Case A), the memory must be capable of holding at least *18* blocks needed to compute $E_x$, $E_y$, and $E_z$ under the

double buffering scheme. If the total elapsed time is data transfer bound (see Figure 4.8(b)), then the ($N^3 + 1$) non-overlapped operations given by Theorem 4.6 consists of $N^3$ data transfers of *9* blocks each and **E**-field computations for a single block. On the other hand if the total elapsed time is compute bound, then the ($N^3 + 1$) non-overlapped operations given by Theorem 4.6 consists of **E**-field computations for $N^3$ blocks and an initial transfer of *9* blocks. When no data sharing between the blocks is considered, the memory need only hold the minimum of 8 blocks needed to compute $E_x$ (or $E_y$, or $E_z$) under the double buffering scheme. However, without data sharing at least ($3N^3+1$) operations (computation or data transfer) cannot be overlapped. If the total elapsed time is data transfer bound (see Figure 4.8(a)), then the ($3N^3+1$) operations consists of ($4b^3 \times 3N^3$) data transfers and the $E_x$ (or $E_y$, or $E_z$) computation for a single block. For the compute bound case, the ($3N^3 + 1$) operations consist of $9b^3 \times 3N^3$ **E**-field computations, and initial data transfer of *4* blocks required to calculate $E_x$ (or $E_y$, or $E_z$). In all cases, all setup times except for the initial one can be overlapped with either data transfer or computation.

### 4.6.2 Discussion

A fair comparison between the theoretical lower bounds presented above requires the expressing the total elapsed time in terms of the problem size *n* and the size of the local storage *M*. Although use of a higher order buffering scheme enables a better overlap between the computations and the data transfer, with a fixed size memory, the block size is smaller, resulting in lesser opportunities for exploiting temporal locality. Also, unlike matrix multiplication in the FDTD algorithm, the number of blocks to be held in memory depends on the degree of data sharing between the blocks. Note that buffer size is same

as the block size. We assume the maximum size of a single buffer to be *M* and scale the buffer size down by the number of buffers that needs to be maintained in memory. The theoretical lower bounds for the elapsed time can then be expressed as follows –

Case I: Single buffer

$$T_{total} \geq T_s \times \frac{9}{M} \times n^3 + T_{mem} \times (9 \times n^3) + T_{comp} \times (27 \times n^3) \qquad (4.26)$$

Case II: Double buffers

Data transfer bound:

*Case A:* $(9b^3 T_{mem} > 27b^3 T_{comp} + T_s)$ *for 9 blocks space*

$$T_{total} \geq T_s + T_{mem} \times 9n^3 + T_{comp} \times 27(M/9) \qquad (4.27)$$

*Case B:* $(4b^3 T_{mem} > 9b^3 T_{comp} + T_s)$ *for 4 blocks space*

$$T_{total} \geq T_s + T_{mem} \times 12n^3 + T_{comp} \times 9(M/4) \qquad (4.28)$$

Compute Bound:

*Case A:* $(27b^3 T_{comp} > 9b^3 T_{mem} + T_s)$

$$T_{total} \geq T_s + T_{mem} \times (M/2) + T_{comp} \times 27n^3 \qquad (4.29)$$

*Case B:* $(9b^3 T_{comp} > 4b^3 T_{mem} + T_s)$

$$T_{total} \geq T_s + T_{mem} \times (M/2) + T_{comp} \times 27n^3 \qquad (4.30)$$

*Performance Evaluation on the Cell B.E.:*

We evaluate the lower bound performance for the three buffering schemes discussed above on the Cell BE. For the Cell/B.E. the setup time is on the order of *130 ns* [20], while $T_{mem}$ is the order of *0.018 ns* per single precision float (the theoretical peak data bandwidth of *204.8* GB/s). $T_{comp}$ is the order of *0.036 ns* for one single-precision floating-point operation calculated from theoretical peak 25.6 GFLOPS performance per SPU. We assume the local memory size *M* available for computation is *0.25 million*

*single precision floats* (100 KB) per LS. For these machine parameters FDTD is a compute bound application with the compute time constituting about 80% of the total elapsed time. Figure 4.9 shows the theoretical lower bound for the two buffering schemes. As shown in Figure 4.9, double-buffering with integrated caching shows *14%* better performance over single buffering with integrated caching.

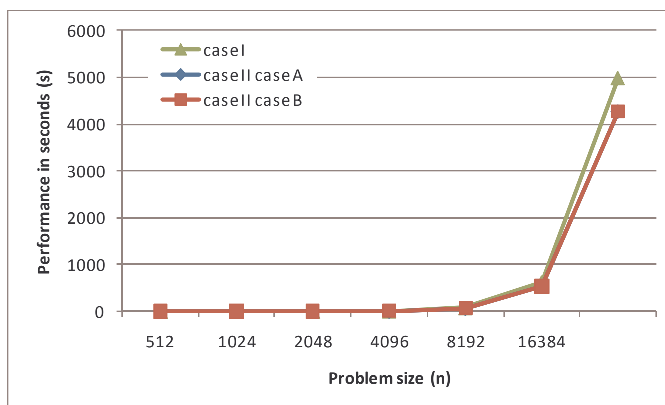



FIGURE 4.9: Theoretical lower bounds for FDTD on IBM Cell/B.E.

## 4.7 Conclusion

In order to derive efficient data transfers for multicore, we propose algorithm specific integrated prefetching and caching strategies for realizing compute-transfer parallelism. The goal of our analysis is to determine the best buffering strategy for limited memory size while simultaneously exploiting data locality. Higher performance improvement is expected on data transfer bound problem with prefetching while a lesser performance improvement is expected for compute bound problems. For example, for matrix multiplication on the IBM Cell BE, where computing is 99% of the total elapsed time, we predict less than 1% performance improvement for prefetching with double

buffering, as compared to the case when prefetching is not used. On the other hand, for the FDTD algorithm on the IBM Cell BE, where computing is about 80% of the total elapsed time, the double-buffering with prefetching is expected to show a 14% performance improvement over the case when prefetching is not used. We conclude that even in compute bound problems prefetching can result in improvement in the overall performance. Note that we have considered system peak performance of the IBM Cell/B.E. platform in our analysis. Measuring the actual system parameters using micro-kernels that capture the compute and data transfer characteristics of the algorithms under consideration can result in better accuracy. For example, the data transfer time $T_{mem}$ depends on data transfer size [45] while $T_{comp}$ is independent with data transfer size.

# CHAPTER 5: EXPERIMENTAL STUDIES IN COMPUTING ON COMMERCIAL MULTICORE PROCESSORS

## 5.1 Introduction

In this chapter, we investigate multicore efficient implementations of data parallel algorithms on commercial multicore platforms. We initially identify a set of in-core optimization techniques that allow us to improve the sequential performance of an algorithm on a single core. While the exact implementation of these in-core optimization techniques depends on the architecture, compiler and the parallel programming tools, most of the commercial multicores architectures, compilers and parallel programming tools support these techniques is some fashion. We utilize the effective blocking and data prefetching techniques discussed in Chapters 3 and 4 to obtain multicore efficient implementations of the algorithms under considerations. The algorithms considered are data parallel algorithms drawn from scientific computing and includes matrix multiplication, FDTD, LU decomposition and Gauss-Seidel power flow solver. We present extensive measurements of the performance of these algorithms on the Intel Colvertown and IBM Cell BE multicores. The two architectures represent two ends of the multicore architecture design philosophies. The Intel Colvertown is a shared cache quad-core processor with complex out-of-order processors and hardware controlled cache coherence. On the other hand, the IBM Cell BE has 8 self contained vector processors with programmer controlled local memory.

The chapter is organized as follows – we briefly discuss about our experimental systems in Section 5.2. We illustrate in-core optimization techniques in Section 5.3. We discuss about case studies in Section 5.4. The conclusion is in Section 5.5.

## 5.2 Experimental Systems

In this section, we briefly discuss about our experimental systems based on multicore processors. For our experimental studies, we consider two commercial platforms: (1) Intel Clovertown platform: Dell Precision-690 as a homogenous multicore platform and (2) IBM Cell/B.E. platform: SONY PlayStation3 as a heterogeneous multicore platform.

*Intel Clovertown Platform:*

Intel Clovertown platform consists of two Intel Xeon E5345 quad-core processors (Clovertown) on a dual-socket shown in Figure 5.1.

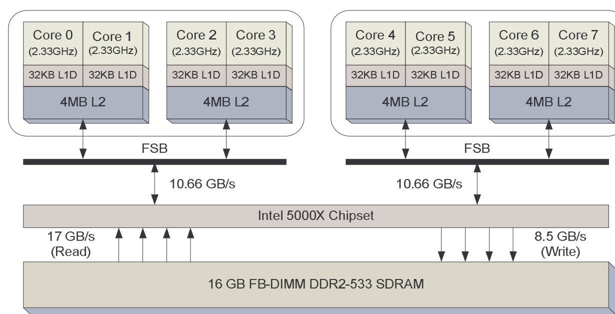



FIGURE 5.1: Dell Precision 690 with dual Intel quad-core Xeon E5345.

The characteristics of our system are that a) each core runs at *2.33* GHz, b) it is capable of fetching and decoding four instructions per cycle, and fully support *128*-bit SSE for the theoretical peak performance of *9.32* GFLOPS per core for single-precision floating-point operations, c) each socket provides the theoretical peak memory bandwidth

of *10.66* GB/s, and d) all *8* cores share *16* GB off-chip memory interfaced to four FB-DIMM DDR2-533 SDRAM channels providing *17* GB/s of the theoretical memory bandwidth. Each core has a private *32* KB L1 cache, and each chip (two cores) has a shared *4* MB L2 cache.

*IBM Cell/B.E. Platform:*

IBM Cell/B.E. platform consists of IBM PowerPC-based Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs) shown in Figure 5.2. The characteristics of the platform are that a) each SPE runs at *3.2* GHz, b) it is capable of fetching and decoding four instruction per cycle, and fully support *128*-entry *128*-bit SIMD organization for the theoretical peak performance of *25.6* GFLOPS per SPE for single-precision floating-point operations, c) Element Interconnect Bus (EIB) provides the theoretical peak data bandwidth of *204.8* GB/s, and d) all *8* SPEs share *200* MB DRAM off-chip memory interfaced to EIB. Each SPE has an efficient software-controlled DMA engine which transfers data between DRAM and the private *256* KB Local-Store (LS) from execution. The LS holds both instructions and data.

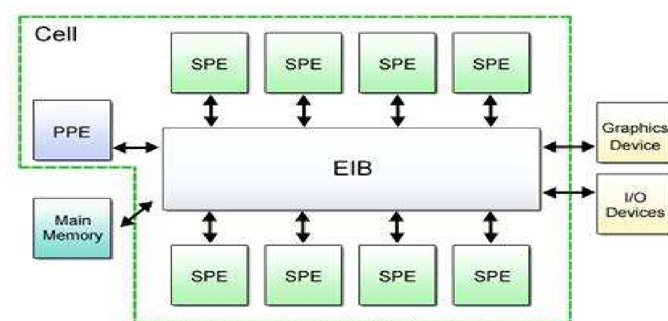
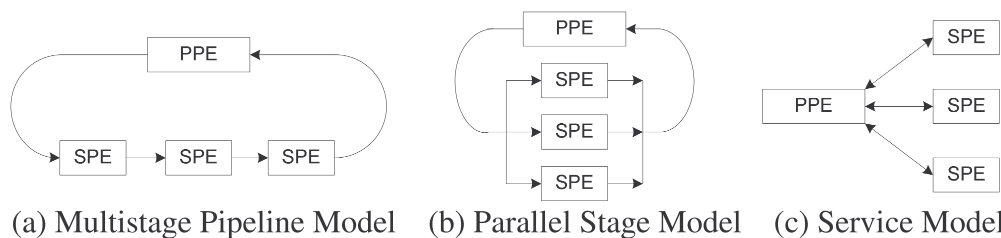



FIGURE 5.2: SONY PlayStation3 with one PPE and eight SPEs.

As shown in Figure 5.3, there are three ways in which the SPEs can be used in the PPE-centric model. Figure 5.3(a) shows the multistage pipeline model, the parallel stages model is shown in Figure 5.3(b) and the services model is shown in Figure 5.3(c). The multistage pipeline model is typically avoided because of the difficulty of load balancing. In addition, the multistage model increases the data-movement requirement because data must be moved for each stage of the pipeline. The parallel stages model is used for a task which has a large amount of data that can be partitioned and acted on at the same time. In the services model, the PPE assigns different services to different SPEs, and the PPE's main process calls upon the appropriate SPE when a particular service is needed. We use parallel stages model for our IBM Cell/B.E. implementations.




(a) Multistage Pipeline Model (b) Parallel Stage Model (c) Service Model

FIGURE 5.3: The PPE-centric programming models.

*Operating Systems, Compilers and Performance Analysis Tools:*

The Intel Clovertown platform runs Fedora-9 with version *2.6* of the Linux kernel, and the IBM Cell/B.E. platform runs Yellow Dog Linux-6.1 with version *2.6* of the Linux kernel. All of our applications use single-precision floating-point across both architectures. Intel compiler *icc-11.0* is used with *–O3* compiler optimization option for all implementations on the Intel platform, and IBM *spu-gcc* compiler is used with *–O3* compiler optimization option for all implementations on the IBM Cell/B.E. platform.

Additionally, we use Intel Vtune™ Performance Analyzer with Intel Thread Profiler to analyze multi-thread and cache performance for Intel Clovertown platform.

## 5.3 In-core Optimization Techniques

Many data parallel scientific and engineering algorithms spend most of their execution time on loop iterations and use multi-dimensional arrays as the principal data structure.

When the referenced data is reused in an algorithm, the deep cache hierarchy of multicore processor allows the exploitation of data locality [48, 52, 58, 77]. The two forms of data reuse are *temporal* and *spatial* reuse. Temporal reuse (*temporal locality*) occurs when the same data is reused in a short time period. Spatial reuse (*spatial locality*) occurs when data in the same cache line or a block of memory at same level of the memory hierarchy is used (unit-stride memory access is the most common type of spatial locality). Wolf and Lam provide a concise definition and summary of important types of data locality [77].

The performance of optimization techniques depends on both the algorithm and the machine architecture. Recent research has shown ways to improve performance using optimization techniques to exploit spatial and temporal locality on multicore architectures [25, 26, 28, 41, 76]. However, there exists no such universal way to utilize these optimization techniques. Therefore, understanding the use of these optimization techniques in developing programs for classes of algorithms and machines is essential in achieving high performance on multicore architectures.

In this section, we describe practical optimization techniques based on data transformation, loop transformation and vectorization to improve the single thread

performance through the standard dense matrix-vector multiplication with single-precision floating-point.

The performance of the in-core optimization techniques described in this section has been experimentally evaluated on single core of the Intel Clovertown platform shown in Figure 5.1. The platform runs Fedora-9 with version *2.6* of the Linux kernel. In this section, we implement all algorithms in C program language and compile using GNU *gcc-4.3* with optimization level *–O3*. In general, use of a higher level compiler optimization increases the compile time and the resulting code size. Although compilers with optimization flags attempt to generate optimized version of the code, compilers often fail at effective optimization. Therefore, additional improvements in performance are possible through manual optimizations.

### 5.3.1 Matrix-Vector Multiplication

Matrix-vector multiplication is an important computational kernel used in scientific computation, signal and image processing, and many other applications. As shown in Equation 5.1, the matrix-vector multiplication algorithm multiplies an $m \times n$ matrix $A$ and $n$ vector $b$ to get a result $m$ vector $c$**.** The computational and data read/write aspects of the algorithm is shown in Figure 5.4. The computational complexity is $O(m \times n)$ while the data space requirement is $O(m \times n)$.

$$c = A \times b \tag{5.1}$$

As shown in Figure 5.4, each element of the matrix $A$ is only read once while each element of both vectors $b$ and $c$ are used $m$ times and $n$ times respectively. Therefore, only spatial locality is critical for matrix $A$, but both temporal and spatial localities are important for $c$ and $b$ vectors.

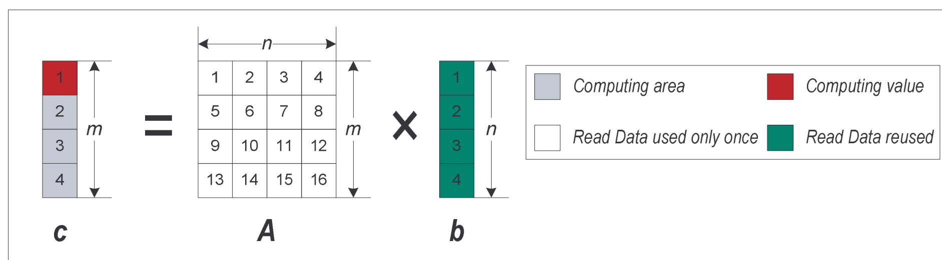



FIGURE 5.4: Matrix-vector multiplication with *n=4* and *m=4*.

### 5.3.2 Data Transformation: Data Layout Scheme

Programming languages that offer support for multi-dimensional arrays generally use one of two linear layouts – row-major or column-major layout, to translate from multi-dimensional array indices to locations in the memory space. C/C++ and Pascal uses the row-major layout scheme while Matlab and Fortran uses the column-major layout scheme.

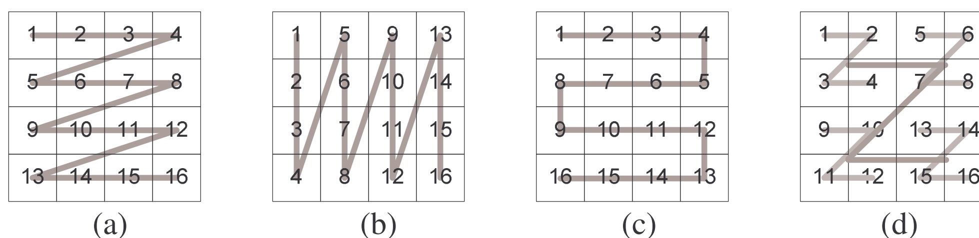



FIGURE 5.5: Data Layout Schemes of 4×4 Matrix: (a) Row-major order; (b) Column-major order; (c) Space-filling-curve order; (d) Z-Morton order.

A computational order (also known as scheduling) that traverses an array in the same order as it is laid out in memory leads to better spatial locality. However, traversing a row-major order layout in column-major computational order or vice-versa, can lead to reduced performance. Computation orders that seek to exploit data locality also rely on

an appropriate data layout scheme [19, 65]. Some of the well-known data layout schemes of two-dimensional arrays are shown in Figure 5.5.

The row-major and column-major order layout schemes shown in Figure 5.5(a) and (b), respectively, are generally easy to implement and no extra time is required for re-structuring the data layout. The space-filling-curve order layout scheme shown in Figure 5.5(c) potentially achieve better locality than row-major or column-major order layout schemes for some applications, such as matrix multiplication. However, since extra effort is required to perform with this layout, it may lead to degradation in the overall performance. The Morton order layout schemes such as Z-Morton shown in Figure 5.5(d) can be useful with loop blocking optimization technique in a limited space. Any of the layout schemes can be combined if necessary. Efficient data layout schemes for an algorithm should be selected by matching data layout with computational order to achieve good performance.

TABLE 5.1: Pseudo code of the matrix-vector multiplication for different data layout schemes.

| 1: `for (i=0;i<m;i++)`<br>2: `for (j=0;j<n;j++)`<br>3: `c[i]+=A[i][j]*b[j];` | 1: `for (i=0;i<m;i++)`<br>2: `for (j=0;j<n;j++)`<br>3: `c[i]+=A[j][i]*b[j];` | 1: `for (i=0;i<m;i++)`<br>2: `for (j=0;j<n;j++) {`<br>3: `if (i%2==0)`<br>4: `c[i] += A[i][j]*b[j];`<br>5: `if (i%2==1)`<br>6: `c[i] += A[i][j]*b[n-j]; }` |
|---|---|---|
| (a) Row-major layout | (b) Column-major layout | (c) Space-filling-curve layout |

The pseudo code describing the nested loop of the matrix-vector multiplication algorithm is shown in Table 5.1. In Table 5.1(a) the algorithm traverses matrix $A$ in row-major computing order. As shown in Table 5.1(b) and (c) respectively, the computations

are re-ordered in both column-major order layout and space-filling-curve order layout for correctness of the original row-major computational order algorithm.

*Performance Analysis:*

Figure 5.6 shows the performance of the layout schemes described above for matrix-vector multiplication with varying problem size $n$ with $m$ fixed at *1024*. The computational order is same as the row-major order for all layout schemes. In this study, we vary the layout schemes for matrix $A$ while vectors $c$ and $b$ are stored in memory as unit-stride fashion.

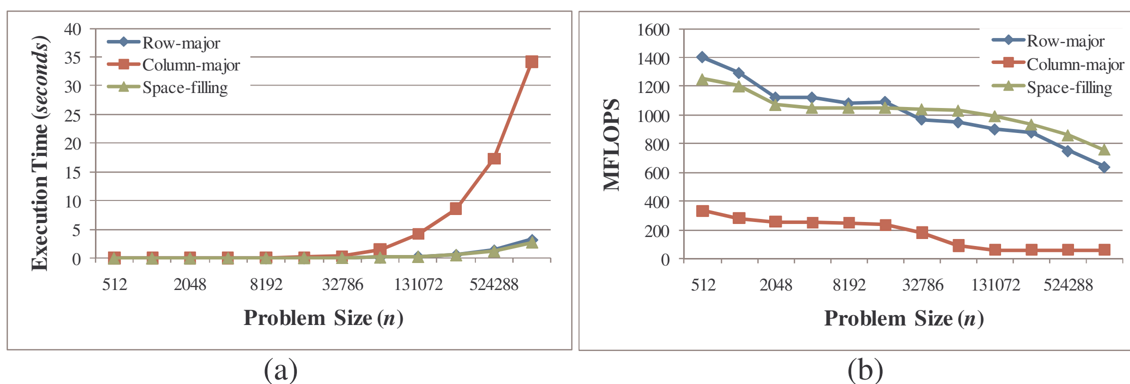



FIGURE 5.6: Performance of data layout schemes for matrix-vector multiplication with fixed $m = 1024$: (a) The performance in seconds; (b) The performance in MFLOPS.

The row-major order layout scheme incurs ($m$-$1$)-times non-unit-stride memory access penalty in accessing vector $b$ while matrix $A$ and vector $c$ are accessed in unit-stride fashion. The non-unit-stride memory access can lead poor locality. An example memory access pattern for a $4 \times 4$ array laid out in a row-major layout scheme is shown in Table 5.2.

TABLE 5.2: The memory access pattern obtained by following a row-major computational order in the nested loop with the *4×4* matrix *A* laid out in a row-major layout scheme.

| | Memory address by each column of matrix $A$ | | | |
|---|---|---|---|---|
| | Column 1 | Column 2 | Column 3 | Column 4 |
| Matrix $A$ | $1 \to 2 \to 3 \to 4 \to$ | $5 \to 6 \to 7 \to 8 \to$ | $9 \to 10 \to 11 \to 12 \to$ | $13 \to 14 \to 15 \to 16$ |
| Vector $b$ | $1 \to 2 \to 3 \to 4 \to$ | $1 \to 2 \to 3 \to 4 \to$ | $1 \to 2 \to 3 \to 4 \to$ | $1 \to 2 \to 3 \to 4$ |
| Vector $c$ | $1 \to 1 \to 1 \to 1 \to$ | $2 \to 2 \to 2 \to 2 \to$ | $3 \to 3 \to 3 \to 3 \to$ | $4 \to 4 \to 4 \to 4$ |

From Figure 5.6 we observe that the column-major order layout is more than twice slower compared to the row-major order layout. Note that the matrix *A* laid out in the column-major order is traversed following the row-major computation order of the nested loop algorithm. The column-major order layout scheme has ($n \times m+m-2$)-times non-unit-stride memory access penalty in accessing vector *b* and matrix *A*. An example memory access pattern for a *4×4* array laid out in a column-major layout scheme is shown in Table 5.3.

TABLE 5.3: The memory access pattern obtained by following a row-major computational order in the nested loop with the *4×4* matrix *A* laid out in a column-major order layout scheme.

| | Memory address by each column of matrix $A$ | | | |
|---|---|---|---|---|
| | Column 1 | Column 2 | Column 3 | Column 4 |
| Matrix $A$ | $1 \to 5 \to 9 \to 13 \to$ | $2 \to 6 \to 10 \to 14 \to$ | $3 \to 7 \to 11 \to 15 \to$ | $4 \to 8 \to 12 \to 16$ |
| Vector $b$ | $1 \to 2 \to 3 \to 4 \to$ | $1 \to 2 \to 3 \to 4 \to$ | $1 \to 2 \to 3 \to 4 \to$ | $1 \to 2 \to 3 \to 4$ |
| Vector $c$ | $1 \to 1 \to 1 \to 1 \to$ | $2 \to 2 \to 2 \to 2 \to$ | $3 \to 3 \to 3 \to 3 \to$ | $4 \to 4 \to 4 \to 4$ |

From Figure 5.6 we note that the space-filling-curve layout scheme shows slightly better performance than the row-major order scheme for large problem sizes. In this case, the temporal locality of vector *b* is increased by the triangular-stride memory access

pattern when all of vector *b* does not fit in the cache due to space limitations. An example memory access pattern for a *4×4* array laid out in a column-major layout scheme is shown in Table 5.4. As shown in Table 5.4, all arrays are accessed in unit-stride fashion. However, the performance is slightly worse than row-major order layout scheme with small problem size ($n < 32786$) as shown in Figure 5.6. In this case, the size of vector *b* which has temporal locality is less than 128 KB. Here a trade off exists between the gain due to temporal locality and the extra computation required for the triangular stride access pattern [48].

TABLE 5.4: The memory access pattern obtained by following a row-major computational order in the nested loop with the *4×4* matrix *A* laid out in a space-filling curve order layout scheme.

| | Memory address by each column of matrix *A* | | | |
|---|---|---|---|---|
| | Column 1 | Column 2 | Column 3 | Column 4 |
| Matrix *A* | 1 → 2 → 3 → 4→ | 5 → 6 → 7 → 8→ | 9 → 10 → 11 → 12→ | 13 → 14 → 15 → 16 |
| Vector *b* | 1 → 2 → 3 → 4→ | 4 → 3 → 2 → 1→ | 1 → 2 → 3 → 4 → | 4 → 3 → 2 → 1 |
| Vector *c* | 1 → 1 → 1 → 1→ | 2 → 2 → 2 → 2→ | 3 → 3 → 3 → 3 → | 4 → 4 → 4 → 4 |

### 5.3.3 Loop Transformation: Loop Blocking (Loop Tiling)

Loop blocking (tiling) is a well-known compiler optimization that helps improve cache performance by dividing the loop iteration space into smaller blocks (tiles) [56]. Loop blocking has been shown to be useful for many algorithms in linear algebra. For example, the Basic Linear Algebra Library (BLAS) provides high-level matrix operations using blocked algorithms. Previous research has shown the utility of multi-level blocking techniques such as cache blocking and register blocking (also known as *unrolling-and-jam*), when applied to multicore architectures with deep memory hierarchy. The optimal

block sizes can be determined by the cache-efficient and space-efficient data parallel algorithm design methods discussed in Chapter 3.

Advantages of loop blocking include improvement in the data locality (temporal and spatial) when memory is limited and better utilization of the memory bandwidth by reducing communication cost. However, loop blocking may require extra index computations and an increase in non-unit-stride memory access penalties.

TABLE 5.5: An example of matrix vector multiplication with $m \times n$ matrix $A$ using loop blocking ($c = A \times b$).

| (a) Without loop blocking | (b) Loop blocking with 2×2 blocks |
|---|---|
| 1: *for (i=0;i<m;i++)*<br>2: *for (j=0;j<n;j++)*<br>3: *c[i] = c[i] + A[i][j]*b[j];* | 1: *for (i=0;i<m;i+=2)*<br>2: *for (j=0;j<n;j+=2)*<br>3: *for (ii=i;ii<i+2;ii++)*<br>4: *for (jj=j;jj<j+2;jj++)*<br>5: *c[ii] = c[ii] + A[ii][jj]*b[jj];* |

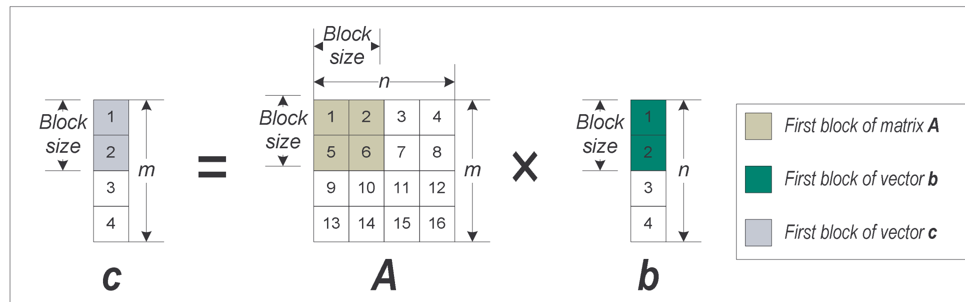


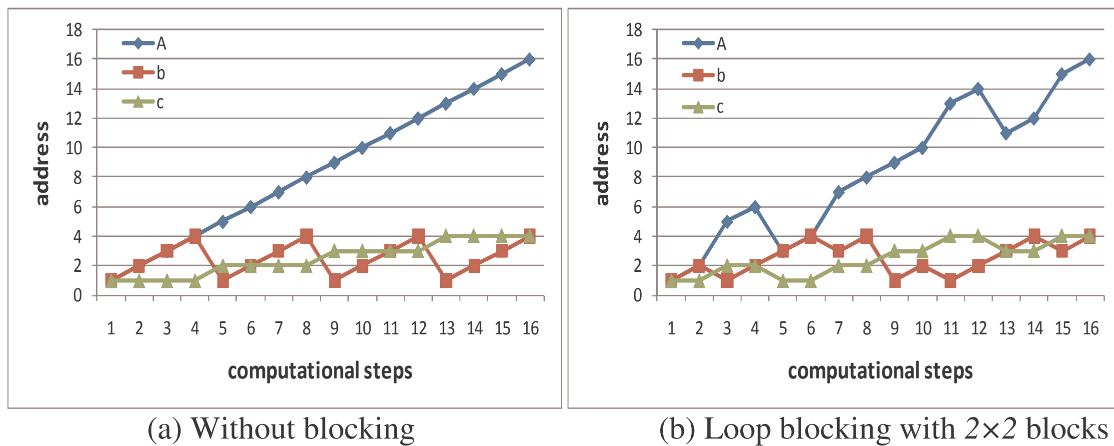

FIGURE 5.7: Implementation of loop blocking algorithm in row-major layout scheme with $n=4$, $m=4$ and $2\times2$ blocks.

For the matrix-vector multiplication algorithms shown in Table 5.5 matrix $A$ is laid out in a row-major order. The loop blocking algorithm with a block size of $2\times2$ is shown in Table 5.5(b). The loop blocking algorithm uses a row-major computing order

both within a block and in traversing the individual blocks of matrix *A*. An example implementation of the loop blocking algorithm in row-major order layout scheme is also shown in Figure 5.7.

*Performance Analysis:*

Figure 5.8 shows the memory access pattern with and without blocking for doubly nested matrix-vector multiplication algorithm. Both implementations use a row-major order layout for $4 \times 4$ matrix *A*. We note that the loop blocking algorithm leads to a memory access pattern of matrix *A* in the same order as the Z-Morton order while the algorithm with blocking traverses matrix *A* in the same order as laid out in memory space.




(a) Without blocking (b) Loop blocking with $2 \times 2$ blocks

FIGURE 5.8: Memory access pattern of matrix-vector multiplication with $n=4$ and $m=4$.

Traversing matrix *A* using the loop blocking algorithm reduces spatial locality by increasing the non-unit-stride memory access penalties. As shown in Figure 5.6, accessing vector *b* in the loop blocking algorithm improves temporal locality, but reduces spatial locality if the block size of vector *b* is greater than *2* ($n>2$). If due to space

limitations, blocked vector $c$ does not fit in memory, accessing vector $c$ in the loop blocking algorithm leads to poor temporal locality compared to the non-blocked algorithm.

The performance of the loop blocking algorithm for matrix-vector multiplication with varying block size is shown in Table 5.6. We use a fixed problem size with $m=1024$ and $n=1048576$ and double precision floating point.

TABLE 5.6: The performance of loop blocking algorithm with varying block size.

| Block size | 1 × 1 | 4 × 4 | 8 × 8 | 16 × 16 | 32 ×32 | 64 × 64 | 128×128 |
|---|---|---|---|---|---|---|---|
| Vector $b$ in Byte | 4 | 16 | 32 | 64 | 128 | 256 | 512 |
| Execution time (s) | 6.4 | 4.5 | 5.2 | 10.6 | 15.6 | 10.5 | 7.8 |

| Block size | 256×256 | 512×512 | 1k × 1k | 2k × 2k | 2k × 4k | 2k × 8k | 2k × 16k |
|---|---|---|---|---|---|---|---|
| Vector $b$ in Byte | 1 K | 2 K | 4 K | 8 K | 16 K | 32 K | 64 K |
| Execution time (s) | 5.7 | 4.6 | 4.3 | 4.0 | 3.9 | 3.9 | 4.0 |

| Block size | 2k× 32k | 2k × 64k | 2k×128k | 2k×256k | 2k×512k | 2k×1M |
|---|---|---|---|---|---|---|
| Vector $b$ in Byte | 128 K | 256 K | 512 K | 1 M | 2 M | 4 M |
| Execution time (s) | 4.0 | 3.9 | 4.0 | 4.4 | 5.4 | 6.4 |

As shown in Table 5.6, the loop blocking algorithm reduces the execution time in most cases. The highest speed-up of loop blocking algorithm over the non-blocked version is about 1.4 and is achieved for several different block sizes. However, some block sizes of the block algorithm lead to a worse performance than the non-blocked case. For example, the performance with $32{\times}32$ block size shows 2.4 times slower worst case performance. Note that even if the block size is small enough as compared to the cache size, the other factors such as replacement policy and set-associative can lead to poor locality by replacing useful entries leading to degradation in cache performance. Also, other optimization techniques, such as choice of layout, padding (for alignment)

and computational reordering may be required to realize the potential benefits of loop blocking.

### 5.3.4 Loop Transformation: Loop Unrolling

Loop unrolling is a well-known compiler optimization technique to convert a loop into straight-line code. This technique helps in the elimination of branch instructions and enables the implementation of a scheduling for efficient cache usage. However, it can lead to an increase in the code size and extra computations with a compiler that does not optimize well. Table 5.7 shows pseudo codes for multiplying ($m \times n$) matrix $A$ by $n$ vector $b$ to get a result $m$ vector $c$. The pseudo code describing the nested loop of the matrix-vector multiplication algorithm is shown in Table 5.7. The nested loop implementation is shown in Table 5.7(a), and the partial loop unrolling implementation with *unrolling_factor* = *2* is shown in Table 5.7(b).

TABLE 5.7: An example of matrix-vector multiplication with $m \times n$ matrix $A$ using loop unrolling ($c = A \times b$).

| | |
|---|---|
| 1: *for (i=0; i<m; i++)*<br>2: *for (j=0; j<n; j++)*<br>3: *c[i] = c[i] + A[i][j]*b[j];* | *1: for (i=0; i<m; i++)*<br>*2: for (j=0; j<n; j+=2) {*<br>*3: c[i] = c[i] + A[i][j]*b[j];*<br>*4: c[i] = c[i] + A[i][j+1]*b[j+1]; }* |
| (a) Nest loop implementation | (b) Partial loop unrolling implementation with *unrolling_factor*=2 |

*Performance Analysis:*

We investigate the performance of different loop unrolling factors with the compiler optimization level *–O3*. We use a fixed problem size of *m=1024* and *n=1024* which performs in *7.6* milliseconds (260 MFLOPS) for the nested loop implementation, *unrolling_factor=0*, with single-precision floating-point. Note that all implementations

use row-major layout scheme. As shown in Table 5.8, the highest performance of 365 MFLOPS is achieved with *unrolling_factor=8*. The performance does not vary much when *unrolling_factor* is greater than *4*.

TABLE 5.8: The performance of loop unrolling algorithm with varying unrolling factor.

| *unrolling_factor* | 0 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| Performance in MFLOPS | 260 | 302 | 353 | 365 | 360 | 364 |
| Execution time (ms) | 7.6 | 6.6 | 5.6 | 5.4 | 5.5 | 5.4 |

### 5.3.5 Loop Transformation: Loop Interchange (computational reordering)

Loop interchange also known as loop permutation is the process of exchanging the order of multiple loop iterations. This technique is useful in achieving simple computational reordering [37]. For the matrix-vector multiplication shown in Table 5.9, the outer loop becomes the inner loop or vice versa. This implies that the loop interchange technique simply changes the computational ordering for matrix-vector multiplication between depth first (1DF) and breadth first (1BF) ordering (see Chapter 3). The effectiveness of loop interchange depends on the behavior of an algorithm and layout scheme.

TABLE 5.9: Examples of matrix-vector multiplication with $m \times n$ matrix $A$ using loop interchange($c = A \times b$).

| `for (j=0; j<n; j++)`<br>`    for (i=0; i<m; i++)`<br>`        c[i] = c[i] + A[i][j]*b[j];` | `for (j=0; j<n; j++)`<br>`    for (i=0; i<m; i++)`<br>`        c[i] += A[j][i]*b[j];` |
|---|---|
| (a) Loop interchange in row-major order layout | (b) Loop interchange in column order layout |

As shown in Table 5.9, both implementations follow 1BF computational ordering, but each implementation uses different layout schemes. The implementation shown in Table 5.9(a) uses row-major order layout, and the implementation shown in Table 5.9(b) uses column-major order layout scheme. The memory access pattern for the implementation, shown in Table 5.9(b), for matrix multiplication with $4\times4$ matrix $A$ is shown in Table 5.10.

TABLE 5.10: The memory access pattern for the loop interchange algorithm with $4\times4$ matrix $A$ in column-major order layout scheme shown in Table 5.9 (b).

| | Memory address by each column of matrix $A$ | | | |
|---|---|---|---|---|
| | Column 1 | Column 2 | Column 3 | Column 4 |
| Matrix $A$ | 1 → 2 → 3 → 4→ | 5 → 6 → 7 → 8→ | 9 → 10 → 11 → 12→ | 13 → 14 → 15 → 16 |
| Vector $b$ | 1 → 1 → 1 → 1→ | 2 → 2 → 2 → 2→ | 3 → 3 → 3 → 3 → | 4 → 4 → 4 → 4 |
| Vector $c$ | 1 → 2 → 3 → 4→ | 1 → 2 → 3 → 4→ | 1 → 2 → 3 → 4 → | 1 → 2 → 3 → 4 |

*Performance Analysis:*

The performance of the loop interchange algorithm for matrix-vector multiplication with varying problem size $n$ is shown in Table 5.11. We use single precision floating point and fix $m=1024$, and vary $n$. The performance of the nest loop algorithm with row-major and column-major order layouts is shown in Table 5.11. The implementation with loop interchange, which changes the computational ordering from 1DF to 1BF for matrix multiplication algorithm, shows better performance for the column-major order layout scheme.

TABLE 5.11: The performance of loop interchange algorithm with varying problem size *n*.

(a) The performance in row-major order layout with loop interchange

| *n* | *512* | *1024* | *2048* | *4096* | *8192* | *16384* |
|---|---|---|---|---|---|---|
| MFLOPS | 477 | 270.16 | 214.89 | 183.39 | 154 | 154 |
| *ms* | 2.1 | 7.4 | 18.61 | 43.62 | 103.7 | 207.2 |

| *n* | *32768* | *65536* | *1311072* | *262144* | *524288* | *1048576* |
|---|---|---|---|---|---|---|
| MFLOPS | 30.57 | 30.1 | 29.56 | 30 | 29 | 28 |
| *ms* | 2093.4 | 4252.3 | 8659.4 | 16847.2 | 35208 | 72877.2 |

(b) The performance in column-major order layout with loop interchange

| *n* | *512* | *1024* | *2048* | *4096* | *8192* | *16384* |
|---|---|---|---|---|---|---|
| MFLOPS | 1377 | 1276 | 1153 | 1087 | 1087 | 1085 |
| *ms* | 0.73 | 1.56 | 3.5 | 7.35 | 14.7 | 29.4 |

| *n* | *32768* | *65536* | *1311072* | *262144* | *524288* | *1048576* |
|---|---|---|---|---|---|---|
| MFLOPS | 1080 | 1087 | 1086 | 1088 | 1086 | 1085 |
| *ms* | 59.24 | 117.73 | 235.57 | 470.23 | 942.28 | 1886.75 |

### 5.3.6 Vectorization

Vectorization is the process of converting a program from a scalar implementation to a vector implementation. While the scalar implementation operates on a pair of operands at a time, the vector implementation can perform multiple operations on a pair of vector (series of adjacent values) operands at a time. Most of general purpose commercial multicores support vectorization using Single Instruction Multiple Data (SIMD) vector extensions to achieve high performance. However, vectorization is a machine dependent optimization and depends on the alignment of data in memory. In order to understand some of the possible vectorization implementation techniques on our experimental setups, several SIMD vectorization (SIMDize) examples using *pragmas* or Intel x86_64 SSE2 intrinsics are shown in Table 5.12. The *pragmas* are machine- or operating system-specific by definition, and are usually different for every compiler. The

*pragmas* directive offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages. The *pragmas* can be used in conditional statements, to provide new preprocessor functionality, or to provide implementation-defined information to the compiler.

In the Table 5.12, we show six sample implementations of the SIMD vectorization techniques for matrix-vector multiplication algorithm. The implementations shown in Table 5.12(a), (b) and (c) use the *pragmas* directive and the loop unrolling technique with *unrolling_factor=4* (see Section 5.3.2) for performing four single-precision floating-point operations simultaneously. We use the *pragmas vector aligned* to support vectorization with unit-stride fashion. We also use the row-major order layout scheme since the computational order is row-major. Note that the row-major order for matrix-vector multiplication is same as following 1DF scheduling. The implementation shown in Table 5.12(a) is a vector implementation without using any local variables where as 5.12(b) and (c) employs local variables. The use of local variables enables register level of intermediate results without incurring memory accesses. The implementation shown in Table 5.12(b) uses one local parameter *t* to store the intermediate computing value and updates output vector *c* in the inner loop. The implementation shown in Table 5.12(c) uses four local parameters to store the intermediate computing values, and updates output vector *c* in the outer loop.

TABLE 5.12: The example of matrix-vector multiplication with $m \times n$ matrix $A$ using vectorization ($c = A \times b$).

```
for (i=0; i<m; i++)
   for (j=0; j<n; j+=4)
      #pragma vector aligned
      c[i] += A[i][j]*b[j];
      c[i] += A[i][j+1]*b[j+1];
      c[i] += A[i][j+2]*b[j+2];
      c[i] += A[i][j+3]*b[j+3];
   end for
end for
```

(a) Row-major using *pragma*

```
for (i=0;i<m;i++)
   for (j=0;j<n;j+=4)
      #pragma vector aligned
       t = A[i][j]*b[j] + A[i][j+1]*b[j+1]
           + A[i][j+2]*b[j+2] + A[i][j+3]*b[j+3];
      c[i] += t;
   end for
end for
```

(b) Row-major using *pragma* and a local variable

```
for (i=0;i<m;i++)
   t[0]=0.0; t[1]=0.0; t[2]=0.0; t[3]=0.0;
   for (j=0;j<n;j+=4)
      #pragma vector aligned
      t[0] += A[i][j]*b[j];
      t[1] += A[i][j+1]*b[j+1];
      t[2] += A[i][j+2]*b[j+2];
      t[3] += A[i][j+3]*b[j+3];
  end for
   c[i] += t[0] + t[1] + t[2] + t[3];
end for
```

(c) Row-major using *pragma* and local variables

```
for (i=0;i<m;i++)
   for (j=0;j<n;j+=4)
      __m128 va = _mm_load_ps(&A[i][j]);
      __m128 vb = _mm_load_ps(&b[j]);
      __m128 vt = _mm_mul_ps(va,vb);
      _mm_store_ps(&t[0],vt);
      c[i] +=  t[0] +t[1]+t[2]+t[3];
   end for
end for
```

(d) Row-major using Intel x86_64 SSE2 intrinsics

```
for (i=0;i<m;i++)
   __m128 vt = _mm_load_ps1(0.0);
   for (j=0;j<n;j+=4)
      __m128 va = _mm_load_ps(&A[i][j]);
      __m128 vb = _mm_load_ps(&b[j]);
      vt = _mm_add_ps(_mm_mul_ps(va,vb),vt);
   end for
   _mm_store_ps(&t[0],vt);
   c[i] += t[0] + t[1] + t[2] + t[3];
end for
```

(e) Row-major using Intel x86_64 SSE2 intrinsics and local variables in outer loop

```
for (j=0;j<n;j++)
   for (i=0;i<m;i+=4)
      __m128 va = _mm_load_ps(&A[j][i]);
      __m128 vb = _mm_load_ps1(&b[j]);
      __m128 vc = _mm_load_ps(&c[i]);
      vc=_mm_add_ps(_mm_mul_ps(va,vb),vc);
      _mm_store_ps(&c[i],vc);
   end for
end for
```

(f) Colum-major with loop interchange using Intel x86_64 SSE2 intrinsics

The implementations shown in Table 5.12(d), (e) and (f) use Intel x86_64 SSE2 intrinsics with loop unrolling technique with *unrolling_factor=4* for four single-precision floating-point operations simultaneously. Similar to the *pragma* implementations, Table 5.12(d) and (e) uses local variables. The implementation shown in Table 5.12(d) uses row-major scheme and updates output vector *c* in the inner loop. The implementation shown in Table 5.12(e) uses row-major scheme and updates output vector *c* in the outer loop. The implementation shown in Table 5.12(f) uses column-major scheme and loop interchange, and updates output vector *c* in the inner loop.

*Performance Analysis:*

The performance of vectorization techniques for matrix-vector multiplication with varying problem size *n* is shown in Table 5.13. In this study, we use a fixed *m=1024,* while varying the problem size *n*. All implementations for vecctorization techniques use single precision floating point.

As shown in Table 5.13(a) and (b), the *pragma* implementations using intermediate local variables to update output vector *c* in the inner loop, shows at most 2 times speed-up (*n=512*) compared with the simple implementation shown in 5.12(a). As shown in Table 5.13(c), we observe similar performance for the implementations irrespective of whether the vector *c* is updated in the inner or outer loop.

As shown in Table 5.13(d) and (e), when the vectorization is implemented using the Intel x86_64 SSE2 intrinsics, the implementation using intermediate local variables to update output vector *c* in inner loop, (Table 5.12(d)), shows similar performance as the simple implementation (Table 5.12(a)). However, as shown in Table 5.13(f) the implementation using intermediate local variables to update output vector *c* in the outer

loop, (Table 5.12(e)), performs at most 4.3 times speed-up (*n=512*) as compared to the simple implementation (Table 5.12(a)).

TABLE 5.13: The performance of vectorization algorithms with a varying problem size *n* and a fixed *m=1024*.

(a) The performance in row-major layout scheme using *pragma*

| *n* | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 1311072 | 262144 | 524288 | 1048576 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MFLOPS | 956 | 900 | 868 | 885 | 873 | 883 | 858 | 848 | 836 | 779 | 733 | 645 |
| *ms* | 1.04 | 2.22 | 4.6 | 9.03 | 18.3 | 36.2 | 74.57 | 150.87 | 306.19 | 656.54 | 1396.49 | 3172.56 |

(b) The performance in row-major layout scheme using *pragma* and a local variable

| *n* | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 1311072 | 262144 | 524288 | 1048576 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MFLOPS | 1901 | 1464 | 1196 | 1162 | 1142 | 1146 | 1086 | 1074 | 1040 | 907 | 786 | 707 |
| *ms* | 0.52 | 1.36 | 3.34 | 6.88 | 14.01 | 27.92 | 58.9 | 119.07 | 246.03 | 564.25 | 1301.77 | 2895 |

(c) The performance in row-major layout scheme using *pragma* and local variables

| *n* | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 1311072 | 262144 | 524288 | 1048576 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MFLOPS | 1893 | 1567 | 1205 | 1154 | 1148 | 1162 | 1074 | 1056 | 1015 | 908 | 785 | 662 |
| *ms* | 0.528 | 1.27 | 3.31 | 6.92 | 13.93 | 27.52 | 59.56 | 121.12 | 252.00 | 563.53 | 1303.14 | 3092 |

(d) The performance in row-major layout scheme SSE2 intrinsics

| *n* | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 1311072 | 262144 | 524288 | 1048576 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MFLOPS | 959 | 928 | 895 | 890 | 882 | 886 | 886 | 871 | 859 | 810 | 757 | 678 |
| *ms* | 1.04 | 2.15 | 4.46 | 8.97 | 18.12 | 36.09 | 72.23 | 126.88 | 297.88 | 632.09 | 1352.61 | 3019.65 |

(e) The performance in row-major layout scheme using SSE2 intrinsics and local variables

| *n* | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 1311072 | 262144 | 524288 | 1048576 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MFLOPS | 4149 | 2328 | 1419 | 1275 | 1285 | 1315 | 1262 | 1270 | 1205 | 1100 | 897.9 | 739 |
| *ms* | 0.24 | 0.85 | 2.81 | 6.27 | 12.44 | 24.32 | 50.67 | 100.72 | 212.42 | 465.1 | 1140.43 | 2769.2 |

(f) The performance in column-major layout scheme with loop interchange using SSE2 intrinsics

| *n* | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 1311072 | 262144 | 524288 | 1048576 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MFLOPS | 3597 | 2249 | 1548 | 1361 | 1321 | 1317 | 1322 | 1324 | 1317 | 1327 | 1324 | 1323 |
| *ms* | 0.27 | 0.88 | 2.58 | 5.87 | 12.1 | 24.28 | 48.39 | 96.66 | 194.31 | 385.79 | 772.9 | 1547.06 |

Use of row-major order for both layout and computational order for vectorization requires scalar operation to the update output vector *c*. The implementation shown in Table 12(f) uses column-major layout and computational order to eliminate scalar

computations to update output vector $c$. The performance of this scheme, shown in Table 5.12(f), shows the highest performance for large problem sizes ($n > 2048$).

Therefore, for a given algorithm, efficient vectorization techniques depend on both the layout and the computational order.

## 5.4 Case Studies: Experimental Results and Performance Analysis

In this section, we demonstrate the effectiveness of the proposed parallel programming design methodology using several algorithms as benchmarks: Dense Matrix Multiplication (DMM), Finite Difference Time Domain (FDTD), LU Decomposition, and Power Flow Solver with Gauss-Seidel (PFS-GS). The algorithms are popular computational methods in science and engineering. Moreover, the applications have different memory access patterns on multi-dimensional arrays. The experimental results and performance analysis of each algorithm are summarized in the following sections. The performance analysis can be applied for other applications which have similar memory access patterns.

### 5.4.1 Dense Matrix Multiplication (DMM)

In this subsection, we discuss the parallel implementations of the matrix multiplication algorithm for multiplying two $n \times n$ square matrices $A$ and $B$ to get a result $n \times n$ square matrix $C = A \times B$ where $n$ is a power of $2$ on both the Intel Clovertown and the IBM Cell/B.E. platforms. The matrix multiplication is an important kernel in science and engineering problems. Also it is closely related to other linear algebra algorithms and is one of the most-studied algorithms in high performance computing [5, 6, 33, 38]. The computational complexity of the conventional serial matrix multiplication algorithm shown in Table 5.14 is $O(n^3)$ while the data access time and space requirements are

$O(n^2)$. The data dependency of the standard matrix multiplication is shown in Figure 5.9. In the standard matrix multiplication operation, each output element of matrix $C$ is updated with the dot product of one row of matrix $A$ and one column of matrix $B$. Therefore, with $O(n) = O(n^3/n^2)$ times of data reused for each element of the three matrices, ensuring efficient memory access is an important challenges on multicores. Moreover, for cache- and space-efficient computing, integrated prefetching and caching, and the use of appropriate in-core optimization are also important factors in improving the parallel performance on multicore platforms.

TABLE 5.14: The conventional serial algorithm for multiplying of two $n \times n$ square matrices.

```
1:  for (i=0;i<n;i++)
2:    for (j=0;j<n;j++)
3:      for (k=0;k<n;k++)
4:          C[i][j] = C[i][j] +A[i][k]×B[k][j];
```

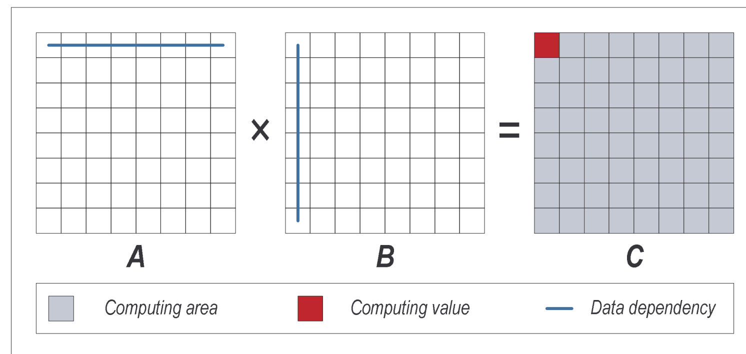



FIGURE 5.9: The data dependency of the standard matrix multiplication with $n \times n$ square matrices.

5.4.1.1 Multicore-efficient Implementation

We develop our multicore-efficient implementations on both the Intel Clovertown and IBM Cell/B.E. platforms following our parallel programming methodology discussed in Chapter 1. For benchmarking, we compare the effectiveness of our approach to that of the naïve parallel implementation which uses three nested loops on the both platforms. Additionally, on the Intel Clovertown platform we compare the effectiveness of our approach to that of the General Matrix Multiply (GEMM) implementation from the Intel Math Kernel Library (MKL). Intel MKL is a library of highly optimized, extensively threaded math routines including Basic Linear Algebra Subprograms (BLAS) for science, engineering, and financial applications that require high performance. Note all implementations in this subsection are based on the conventional serial algorithm shown in Table 5.14.

*Naïve Parallel Implementation:*

The naïve parallel implementation uses row-major order of the array layout for matrix $A$ and $C$, and column-major order of the array layout for matrix $B$. For parallelizing the data among the cores, the row-wise array distribution [34] based on one-dimensional partitioning of the output matrix $C$ is used. Then the computing order of each output partition follows depth first scheduling which is the computing order of the conventional serial algorithm. We use the OpenMP parallel programming library for the Intel Clovertown platform and the IBM *libspe* parallel programming library for the IBM Cell/B.E. platform (see Chapter 2). Since DMA transfer of data is required on the IBM Cell/B.E. platform, we use single buffer each for all matrices without considering caching and prefetching (see Chapter 4). A total of 4 buffers are used (three reads and one write).

The each buffer size is chosen to be 16 KB corresponding to the maximum size of the DMA transfer. The total buffer size of 64 KB is less than the available size of the SPE local store. The GEMM implementation of Intel MKL uses the same data layout as our naïve implementation. GEMM can be parallelized using OpenMP.

*Mutlicore-efficient Implementation:*

We design our multicore-efficient implementation based on our parallel programming methodology (see Chapter 1) to improve the performance on both the Intel Clovertown and the IBM Cell/B.E. platforms. The design steps for our multicore-efficient implementation are as follows –

First, we determine the architecture characteristics of the target platforms to find the model parameters (see Chapter 3) which include the depth of memory hierarchy ($d$), the effective number of processing components ($P_i$) at each level, and the size of the available memory on a component ($M_i$) at level-$i$ where $0 \leq i \leq d$. Note that level-$d$ is the main memory level and level-$0$ is the register level.

Our Intel Clovertown platform shown in Figure 5.1 has $d=3$, $P_1=1$, $P_2=2$, $P_3=4$, $M_1= 32$KB, $M_2=4$MB and $M_3=16$GB. Our IBM Cell/B.E. platform shown in Figure 5.2 has that $d=2$, $P_1=1$, $P_2=8$, $M_1=256$KB and $M_2=200$MB.

Next, we use the Unified Multicore computational model, the weighted-vertex parallel pebble strategy and data-aware scheduling on hierarchical DAGs (see Chapter 3) to analyze the algorithm and to find the block sizes and scheduling at each level. Additionally, we use integrated data prefetching and caching model discussed in Chapter 4 to analyze overlaps between computation and data transfer and determine the multi-

buffering scheme for data transfer. We then modify the block sizes based on the choice of the multi-buffering scheme.

For our Intel Clovertown platform, we choose a three levels of blocking – L2-block for L2 cache, L1-block for L1 cache and register-block for registers. Then, the entire problem of $n \times n$ of each matrix (*A*, *B* and *C*) is partitioned into smaller L2-blocks of size $B_2=|b_2 \times b_2|$. Each L2-block is further partitioned into L1-blocks of size $B_1=|b_1 \times b_1|$. Then each L1-block is partitioned into register-block $B_0=|b_0 \times b_0|$. We then determine the size of block at each level using the weighted-vertex parallel pebble strategy based on CONTROLLED-PDF scheduling illustrated in Chapter 3. The size of the blocks at each level is shown in Table 5.15. The scheduling of the blocks among the effective number of components is based on the CONTROLLED-PDF schedule at each level.

For our IBM Cell/B.E. platform, we choose a two-level blocking, LS-block for LS, and register-block for registers. The entire problem of $n \times n$ of each matrix (*A*, *B* and *C*) is partitioned into smaller LS-blocks of size $B_1=|b_1 \times b_1|$. Then, each LS-block is further partitioned into register-blocks of size $B_0=|b_0 \times b_0|$. We determine the size of block at each level using the weighted-vertex parallel pebble strategy based on CONTROLLED-PDF scheduling. Since we use DMA transfer between main memory and LS, the size of LS-block is modified according to the double-buffering scheme used. The LS-block size is chosen such that eight LS-blocks, three inputs (matrix *A, B* and *C*) and one output (matrix *C*), is less than the available size of each LS. Note that we consider the matrix *C* for both input and output. The size of the blocks at each level is

shown in Table 5.15. The distribution of blocks among the effective number of components is based on CONTROLLED-PDF scheduling at each level.

In the third step, we design our multicore-efficient implementation with the optimal block sizes and scheduling scheme determined in the second step. We use parallel threading model libraries (see Chapter 2) and in-core optimization techniques (see Section 5.3) to achieve close to theoretical performance of the machine. The following in-core optimization techniques are used –

(1) To avoid the penalties of non-unit-stride memory access in multi-level blocking, we determine optimal data layout scheme of the input/output arrays at each level.

(2) We use loop tiling technique to implement multi-level blocking and we reorder computations using such as loop unrolling and loop interchange techniques to achieve our computational scheduling at each level.

(3) We use vectorization technique for computation of the register-block to deliver better performance since both Intel Clovertown and IBM Cell/B.E. support *128-bit* SIMD intrinsics. Although compiler with optimization flags attempt to generate "SIMDized" version of the code, compilers often fail at effective vectorization. We modify the scheduling at the register level as shown in Figure 5.10. Then we use loop unrolling and SIMD intrinsics techniques to implement effective vectorization for register level as shown in Figure 5.10. Further details of the scheduling and vectorization techniques at the register level are provided in Section 5.4.1.2.

For our Intel Clovertown platform, we use Z-Morton order layout scheme of the input/output arrays for 3-level blocking. We use Intel x86_64 SSE2 intrinsics for vectorization and OpenMP for threading model library.

For our IBM Cell/B.E. platform, we use Z-Morton order layout scheme of the input/output arrays for 2-level blocking. We use IBM SPU intrinsics for vectorization and IBM *libspe* for threading model library.

The summary of our multicore-efficient implementation techniques is shown in Table 5.15.

TABLE 5.15: The summary of our implementation techniques of matrix multiplication used for our platforms.

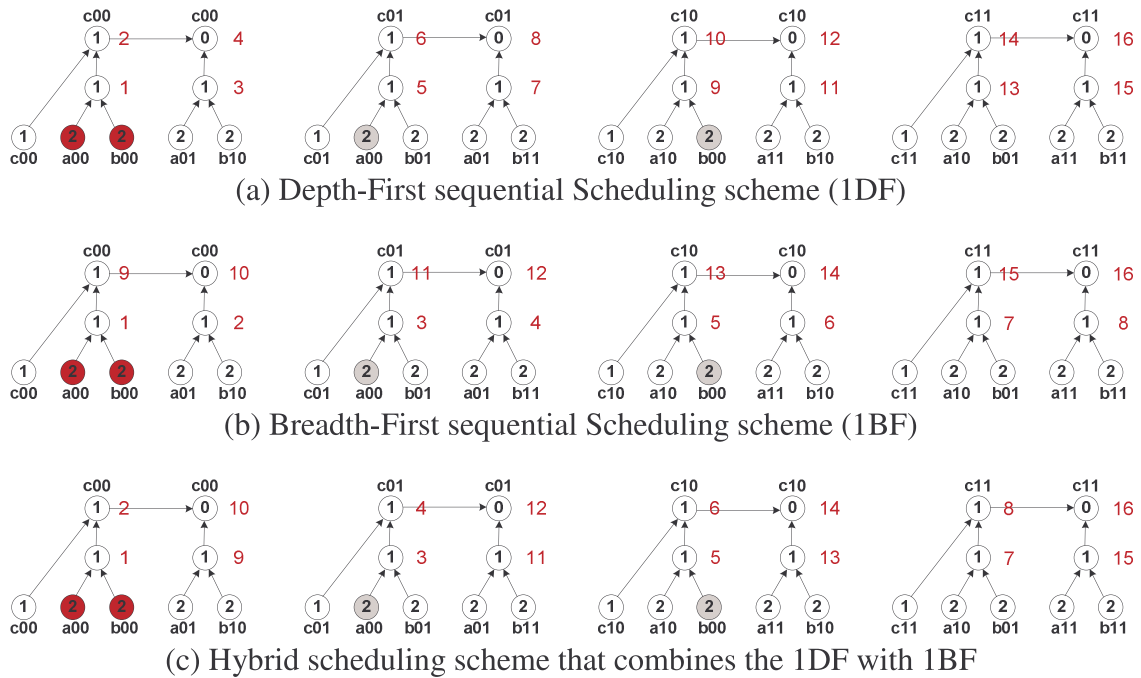
| Optimization Techniques | Multicore Platforms | |
|---|---|---|
| | Intel Clovertown | IBM Cell/B.E. |
| Multi-level blocking | 3-level blocking ($b_0$=4, $b_1$=64, $b_2$=512) | 2-level blocking ($b_0$=4, $b_1$= 64) |
| Scheduling | CONTROLLED-PDF (except at register level) | CONTROLLED-PDF (except at register level) |
| Layout Scheme | Z-Morton ordering | Z-Morton ordering |
| Multi-buffering | Single buffering | Double buffering |
| Vectorization for register level | Intel x86_64 SSE2 intrinsics with *128-bit* registers | IBM Cell/B.E. SPU intrinsics with *128-bit* registers |
| Loop unrolling for Vectorization | *unrolling factor*=4 for register-block | *unrolling factor*=4 for register-block |
| Threading | OpenMP | IBM *libspe* |

5.4.1.2 Optimization at Register Level

*Scheduling at Register Level:*

As mentioned in Section 5.4.1, we use CONTROLLED-PDF scheduling at each level of the memory hierarchy. The CONTROLLED-PDF schedule (see Chapter 3) uses

a 1DF scheduling scheme is used at register level blocking. However, we modify the scheduling at register level blocking to better accommodate vectorication. We show the example of scheduling schemes on the weighted DAGs for register level blocking with $b_0$=2 in Figure 5.10.




(a) Depth-First sequential Scheduling scheme (1DF)

(b) Breadth-First sequential Scheduling scheme (1BF)

(c) Hybrid scheduling scheme that combines the 1DF with 1BF

FIGURE 5.10: The example of the scheduling schemes on the weighted DAGs at register level blocking with $b_0$=2; Note, the number on right side of each computational vertex represents the sequential scheduling order.

In Figure 5.10, the number on the right side of computational vertices for each scheduling scheme indicates the sequential scheduling order. Figure 5.10(a) shows 1DF scheduling scheme which is used in the CONTROLLED-PDF scheduling at register level blocking. Figure 5.10(b) shows 1BF scheduling scheme and Figure 5.10(c) shows hybrid scheduling scheme that combines 1DF and 1BF on weighted DAGs at the register level. To motivate the choice of the scheduling scheme that supports vectorization efficiently,

consider the computation of the elements of the matrix *C* as shown in Figure 5.11. The vectorization based on 1DF and 1BF the vector multiplication computes partial products for the same element of matrix *C* (such as $c00{+}{=}a00{\times}b00$ and $c00{+}{=}a01{\times}b10$) as shown in Figure 5.11(a). Scalar addition of the elements of the output vector is required to obtain the final result. On the other hand, vectorization based on hybrid scheduling scheme computes partial products for different elements of matrix *C* (such as $c00{+}{=}a00{\times}b00$ and $c01{+}{=}a00{\times}b0$) as shown in Figure 5.11(b). The final result is then obtained by vector addition of the output vectors. The hybrid scheme thus allows for vector pipelining and hence is the preferred scheduling method at the register level.

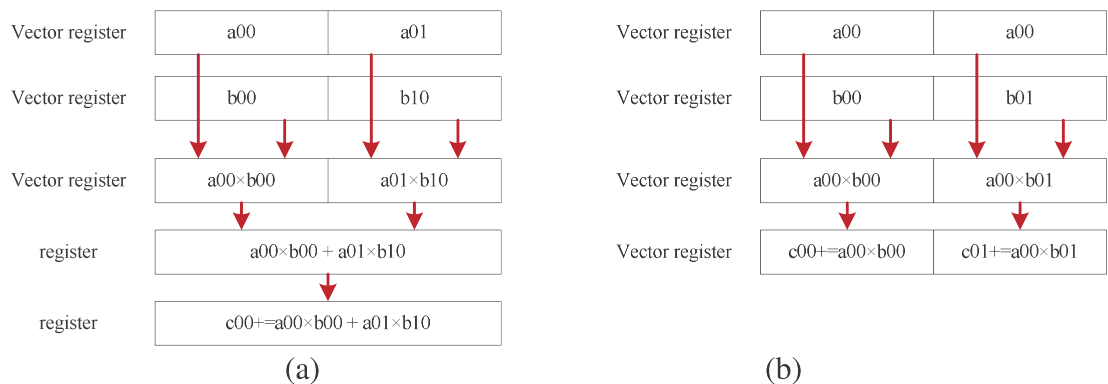



FIGURE 5.11: The example of vector computations for two multiplications following by two addition operations simultaneously: (a) Based on 1DF or 1BF scheduling scheme; (b) Based on hybrid scheduling scheme.

*Vectorization at Register Level:*

We now present the implementation details of vectorization using hybrid scheduling at the register level on the Intel Clovertown platform. Intel x86_64 SSE2 intrinsics are used at with a register level block size of $b_0{=}4$.

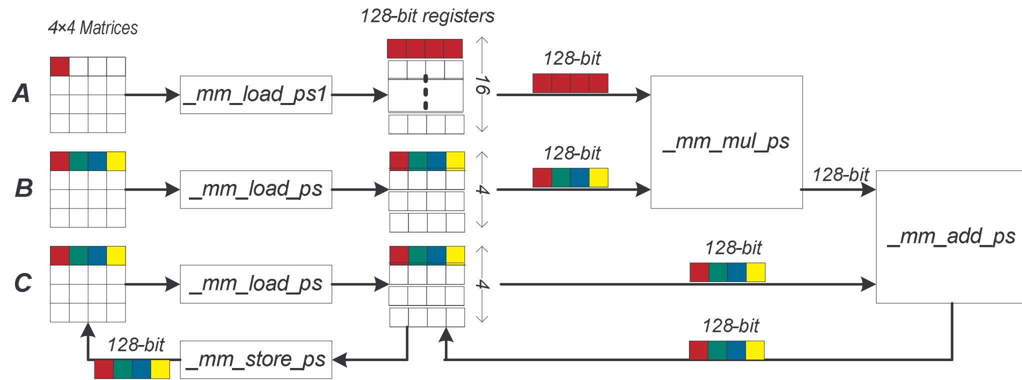



FIGURE 5.12: Vectorization implementation at register level blocking with $b_0=4$ using hybrid scheduling scheme and Intel x86_64 SSE2 intrinsics for Intel Clovertown platform.

In Figure 5.12, first we load each element of the matrix *A* into a *128*-bit register using *_mm_load_ps1* which is an instruction supported by Intel x86_64 SSE2 intrinsics to load one single-precision floating-point data and copy it into all four words of a *128*-bit register. Note that each 128-bit register of matrix *A* represents four pebbles cloning on the weighted-vertex pebble strategy. Then one corresponding row of the matrix *B*, is loaded into a 128-bit register using *_mm_load_ps* which is an instruction supported by Intel x86_64 SSE2 intrinsics to load four aligned single-precision floating-point into a 128-bit register. Also one-row of the matrix *C*, which is product one element of matrix *A* and one-row of matrix *B*, is loaded into a 128-bit register using *_mm_load_ps*. Then we multiply two 128-bit registers of matrix *A* and *B* using *_mm_mul_ps* which is an instruction supported by Intel x86_64 SSE2 intrinsics to multiply two 128-bit registers with single-precision floating-point data at a time, and we add into the 128-bit register of matrix *C* using *_mm_add_ps* which is an instruction supported by Intel x86_64 SSE2 intrinsics to add two 128-bit registers with single-precision floating-point data at a time. We repeat these operations until the multiplication of the two register-blocks with $b_0=4$

of matrix *A* and *B* complete. We then store the matrix *C* into the upper level memory using *_mm_store_ps* which is an instruction supported by Intel x86_64 SSE2 intrinsics to store four single-precision floating-point data into the upper level memory. For our IBM Cell/B.E. platform, we use the IBM SPU intrinsics instead of the Intel x86_64 SSE2 following the same processes as shown in Figure 5.12. Note that we also use loop unrolling technique with *unrolling_factor=4* (see Section 5.3) to implement vectorization at register level blocking with $b_0=4$.

5.4.1.3 Performance Analysis

Now, we compare the effectiveness of the multicore-efficient implementation to that of the naïve parallel implementation on both platforms. Additionally for the Intel Clovertown platform, we compare our multicore-efficient implementation to that of the Intel MKL GEMM implementation.

*Performance on Intel Clovertown Platform:*

First, we show the effect of the L1 block size and register level scheduling on the performance on a single core of the Intel Clovertown platform. The problem size is fixed at $n = 4096$ (corresponding to 16 GB), L2 block size is fixed at $b_2 = 512$ and the register block size is fixed at $b_0 = 4$. 1DF scheduling is used at the L1 and L2 cache levels.

TABLE 5.16: The performance (GFLOPS) for varying schedules and sizes of L1-block ($b_1$) with fixed size of L2-block ($b_2=512$) and register-block ($b_0=4$) on a single core of Intel Clovertown platform. We use 1DF scheduling scheme for L1-level and L2-level blocking, and vary the scheduling scheme at the register level.

| *The size of L1-block $b_1$* | | *4* | *8* | *16* | *32* | *64* | *128* | *256* |
|---|---|---|---|---|---|---|---|---|
| *The size of $3b_1^2$ in bytes* | | *0.18 KB* | *0.75 KB* | *3 KB* | *12 KB* | *48 KB* | *192 KB* | *768 KB* |
| *Scheduling scheme at register level* | *1DF* | 5.78 | 5.35 | 5.18 | 5.95 | 5.22 | 5.69 | 5.02 |
| | *1BF* | 5.88 | 5.75 | 5.78 | 5.85 | 5.32 | 5.39 | 4.92 |
| | *Hybrid* | 5.91 | 5.90 | 5.99 | 6.01 | 6.19 | 5.82 | 5.64 |

As shown in Table 5.16, the highest performance of *6.19* GFLOPS on a single core is achieved with the L1-block size of *64* and a hybrid scheduling scheme at the register level. With an L1-block size of *64*, the memory space required for all three L1-blocks of matrix *A*, *B* and *C* is *48* KB. Unfortunately, this exceeds the 32 KB capacity of the L1 cache of the Intel Clovertown processor. However, the maximum size of the L1-block is $b_1=90$ for a 32 KB L1 cache size. Thus the weighted vertex pebbling strategy allows for larger block size compared to the nominal pebbling strategy (See Chapter 3 for details).

To study the impact of multi-level blocking on performance, we show the cache miss rates (%) and the system bus utilization (%) with respect to scaling of the problem size (problem-scaling) and the number of cores (core-scaling) for both naïve parallel and multicore-efficient implementations on the Intel Clovertown platform.

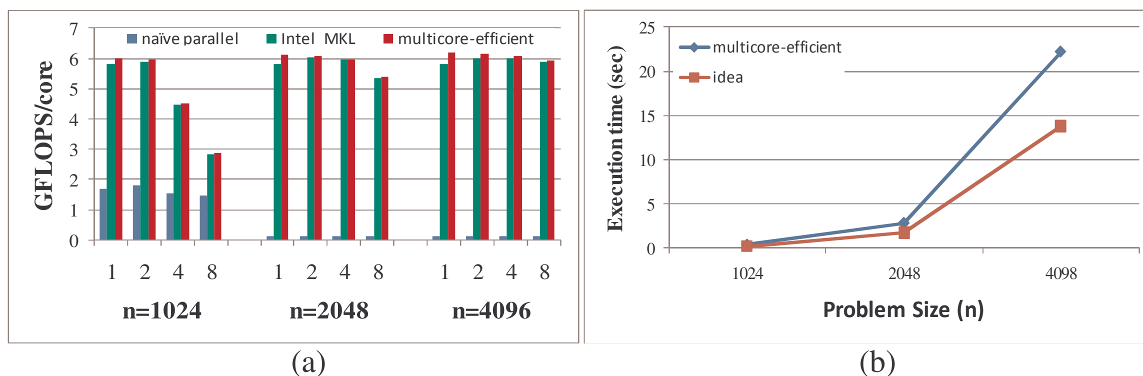
As shown in Table 5.17, our multicore-efficient implementation has negligibly low miss rates demonstrating the benefits of data sharing (temporal locality) among the block computation at each level of the memory hierarchy of the Intel Clovertown platform. Moreover, the Z-Morton order layout minimizes the non-unit-stride access penalties due to multi-level blocking. As seen in Table 5.17 the cache miss rates of the multicore-efficient implementation is independent of core-scaling and problem-scaling with low cache miss rates of *0.001%* for the L2 cache miss rate and *1.7%* for the L1 cache miss rate. On the other hand, the naïve parallel implementation shows almost a linear increase in miss rate with problem scaling while showing no clearly identifiable trend in core-scaling.

TABLE 5.17: Cache miss rate (%) and system bus bandwidth utilization (%) on Intel Clovertown platform.

| Problem size ($n$) | # cores | L2 cache miss rate (%) | | L1 cache miss rate (%) | | Bus utilization (%) | |
|---|---|---|---|---|---|---|---|
| | | naïve parallel | Multicore-efficient | naïve parallel | Multicore-efficient | naïve parallel | Multicore-efficient |
| *1024* | *1* | *0.7* | *0.001* | *2.5* | *1.7* | *69.77* | *1.23* |
| | *2* | *1.1* | *0.001* | *1.5* | *1.7* | *48.24* | *1.5* |
| | *4* | *1.7* | *0.001* | *2.1* | *1.7* | *30.14* | *1.34* |
| | *8* | *1.6* | *0.001* | *2.2* | *1.7* | *34.14* | *1.14* |
| *2048* | *1* | *3.1* | *0.001* | *5.1* | *1.7* | *73.36* | *0.71* |
| | *2* | *4.7* | *0.001* | *4.9* | *1.7* | *65.95* | *0.95* |
| | *4* | *3.7* | *0.001* | *5.3* | *1.7* | *66.53* | *1.11* |
| | *8* | *2.6* | *0.001* | *4.6* | *1.7* | *71.96* | *2.04* |
| *4096* | *1* | *4.2* | *0.001* | *10.3* | *1.7* | *72.7* | *0.82* |
| | *2* | *8.1* | *0.001* | *10.6* | *1.7* | *71.1* | *1.12* |
| | *4* | *5.2* | *0.001* | *10.4* | *1.7* | *87.09* | *1.52* |
| | *8* | *4.2* | *0.001* | *10.2* | *1.7* | *76.2* | *0.69* |

The bus utilization is another concern because memory access times worsen with increasing amounts of traffic on the bus. The low bus utilization (<*2%*) of our approach correlates with the low L2 cache miss rate. On the contrary, the maximum bus utilization is over *80%* for the naïve parallel implementation. From Intel internal measurements and experiments, the memory latencies increase at a rapid rate after ~*60%* FSB utilization [51].

In Figure 5.13, we show the overall performance in GFLOPS per for problem-scaling and core-scaling for both naïve parallel, Intel MKL, and our multicore-efficient implementation on the Intel Clovertown platform.




(a) (b)

FIGURE 5.13: Overall performance on Intel Clovertown platform: (a) Performance in GFLOPS per core; (b) Execution time in seconds on single core.

In Figure 5.13(a), the naïve parallel implementation shows performance of *1.7* GFLOPS with small size of problem (*n=1024*). Here the total problem size of *12* MB for each matrix *A*, *B* and *C* can fit into the four L2 caches. However, for larger problem sizes (*n=2048* and *n=4096*), the naïve parallel approach achieves only *0.2* GFLOPS corresponding to the increased cache miss rates shown in Table 5.17. Our multicore-efficient implementation performs at a performance of *6.2* GFLOPS/core for *n=4096*. Similar performances are attained for smaller problems sizes (*n=1024, n=2048*). These performance figures correspond to the low cache miss rates independent of the problem size (see Table 5.17). The Intel MKL GEMM shows a peak performance of *6* GFLOPS/core. In all of our implementations, we observe an almost linear scaling of performance with respect to the number of cores (core-scaling) when the problem size is large (*n=2048* and *n=4096*). However, for a smaller problem size (*n=1024*), the performance per core gets reduced beyond four cores for both the GEMM implementation of Intel MKL and our multicore-efficient implementation. We believe that for small problem sizes (*n=1024*, *12*MB) where the data fits into the four L2 caches,

while the benefits of data sharing between level-2 blocks is not prominent, the L2 latencies increase with core scaling due to contention on the shared bus.

In Figure 5.13(b), we compare the performance of the naïve and the multicore-efficient implementation with to the ideal computing performance on the Intel Clovertown processor. The ideal execution time for matrix multiplication is the time required for $2n^3$ floating point operations with the theoretical peak core performance of *9.32* GFLOPS on the Intel Clovertown processor. As seen in Figure 5.13(b) the multicore-efficient implementation performs close ideal.

*Performance on IBM Cell/B.E. Platform:*

First, we show the effect of the LS block size ($b_1$) and multi-buffering on the performance on a single SPE of the IBM Cell/B.E. platform. The problem size is fixed at $n = 2048$, the register block size is fixed at $b_0 = 4$. 1DF scheduling is used at both levels.

TABLE 5.18: The performance (GFLOPS) for different multi-buffering schemes and size of LS-block ($b_1$) with fixed size of the register-block ($b_0$=*4*) on a single SPE of IBM Cell/B.E. platform. We use the 1DF scheduling scheme for both level blocking.

| *The size of LS-block $b_1$* | *4* | *8* | *16* | *32* | *64* |
|---|---|---|---|---|---|
| *The size of $4b_1^2$ in bytes* | *0.24 KB* | *1 KB* | *4 KB* | *16 KB* | *64 KB* |
| *Single-buffering with caching and prefetching* | 5.9 | 6.3 | 6.5 | 6.6 | 6.6 |
| *Double-buffering without caching and prefetching* | 6.3 | 6.6 | 6.7 | 6.7 | 6.8 |
| *Double-buffering with caching and prefetching* | 6.8 | 7.0 | 7.1 | 7.1 | 7.1 |

As shown in Table 5.18, the highest performance of *7.1* GFLOPS on a single SPE is achieved with the LS-block sizes ($b_1$=*16*, *32*, and *64*) and double-buffering with caching and prefetching scheme at the LS-block level. With an LS-block size of *64*, the

memory space required for all four LS-blocks of matrix *A*, *B* and *C* is *64*-KB for single-buffering scheme. However, with the block size the double-buffering scheme requires *128* KB (See Chapter 4). For a fixed memory size, the implementation using double-buffering with caching and prefetching scheme shows an *8%* improvement in performance (in GFLOPS) compared to the single-buffering scheme with caching and pre-fetching. This suggests that the gain performance due to overlapping of computation and communication when double buffering offsets the reduced temporarily locality due to smaller block sizes.

In Figure 5.14 we show the overall performance in GFLOPS per for problem-scaling and core-scaling for both naïve parallel and our multicore-efficient implementation on the IBM Cell/B.E. platform.

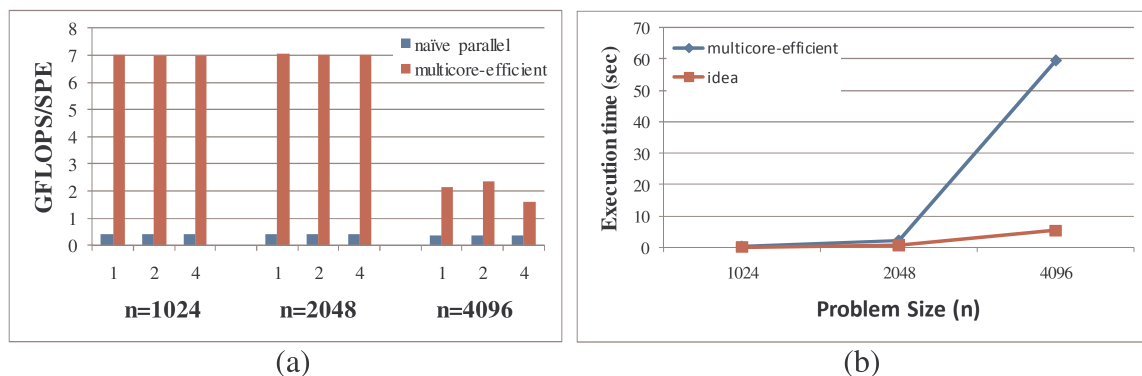



FIGURE 5.14: Overall performance on IBM Cell/B.E. platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on single SPE.

In Figure 5.14(a), the naïve parallel implementation shows a performance of *0.43* GFLOPS/SPE with the problem size (*n=1024*, *n=2048*). However, for a larger problem size (*n=4096*), the performance per SPE reduces to *0.35* GFLOPS/SPE. For *n=4096* the

total memory space requires is *192* MB which exceeds the available size of main memory (*180* MB) of our IBM Cell/B.E. platform. Hence swap space is needed on high latency disk which in turn reduces the performance. Our multicore-efficient implementation shows a performance of *7.1* GFLOPS/SPE for small problem sizes (*n=1024*, *n=2048*) and *2.15* GFLOPS for a larger problem size (*n=4096*). In all of our implementations, we observe an almost linear scaling of performance with respect to the number of cores (core-scaling) for all problem sizes.

In Figure 5.14(b), we compare the performance of the multicore-efficient implementation with that the ideal computing performance on the IBM Cell/B.E. platform. The ideal time for matrix multiplication is the time required for $2n^3$ floating point operations with the theoretical peak performance of *25.6* GFLOPS per SPE for single precision floating point data on the IBM Cell/B.E. platform. As seen in Figure 5.14(b) the multicore-efficient implementation performs close ideal with the problem size (*n=1024*, *n=2048*). However, as explained before, the high disk access latencies incurred with *n=4096*, reduces the performance of our multicore efficient implementation.

### 5.4.2 Finite Difference Time Domain (FDTD)

In this section, we discuss the parallel implementations of the three-dimensional Finite Difference Time Domain (3D-FDTD) method which is a numerical technique proposed by Yee to solve Maxwell's equations [79]. The FDTD method is based on Yee Space Grid [1] and computes the electric-field (**E**-field) and magnetic-field (**H**-field) vectors in both time and space domain [64, 71, 78]. **E**-field and **H**-field vectors are updated at alternate half time steps in a *leapfrog* scheme [78] in time domain. Our

computational equations of **E**-field and **H**-field of the 3D-FDTD for analyzing planar microstrip circuits are as follows –

$$E_x^{\left(t+\frac{1}{2}\right)}(i,j,k) = E_x^{\left(t-\frac{1}{2}\right)}(i,j,k) + \frac{\Delta t}{\epsilon_x(i,j,k)}\left(\frac{H_z^t(i,j,k)-H_z^t(i,j-1,k)}{\Delta y} - \frac{H_y^t(i,j,k)-H_y^t(i,j,k-1)}{\Delta z}\right) \quad (5.2)$$

$$E_y^{\left(t+\frac{1}{2}\right)}(i,j,k) = E_y^{\left(t-\frac{1}{2}\right)}(i,j,k) + \frac{\Delta t}{\epsilon_y(i,j,k)}\left(\frac{H_x^t(i,j,k)-H_x^t(i,j,k-1)}{\Delta z} - \frac{H_z^t(i,j,k)-H_z^t(i-1,j,k)}{\Delta x}\right) \quad (5.3)$$

$$E_z^{\left(t+\frac{1}{2}\right)}(i,j,k) = E_z^{\left(t-\frac{1}{2}\right)}(i,j,k) + \frac{\Delta t}{\epsilon_z(i,j,k)}\left(\frac{H_y^t(i,j,k)-H_y^t(i-1,j,k)}{\Delta x} - \frac{H_x^t(i,j,k)-H_x^t(i,j-1,k)}{\Delta y}\right) \quad (5.4)$$

$$H_x^{(t+1)}(i,j,k) = H_x^t(i,j,k) - \frac{\Delta t}{\mu}\left(\frac{E_y^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_y^{\left(t+\frac{1}{2}\right)}(i,j,k+1)}{\Delta z} - \frac{E_z^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_z^{\left(t+\frac{1}{2}\right)}(i,j+1,k)}{\Delta y}\right) \quad (5.5)$$

$$H_y^{(t+1)}(i,j,k) = H_y^t(i,j,k) + \frac{\Delta t}{\mu}\left(\frac{E_x^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_x^{\left(t+\frac{1}{2}\right)}(i,j,k+1)}{\Delta z} - \frac{E_z^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_z^{\left(t+\frac{1}{2}\right)}(i+1,j,k)}{\Delta x}\right) \quad (5.6)$$

$$H_z^{(t+1)}(i,j,k) = H_z^t(i,j,k) - \frac{\Delta t}{\mu}\left(\frac{E_x^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_x^{\left(t+\frac{1}{2}\right)}(i,j+1,k)}{\Delta y} - \frac{E_y^{\left(t+\frac{1}{2}\right)}(i,j,k)-E_y^{\left(t+\frac{1}{2}\right)}(i+1,j,k)}{\Delta x}\right) \quad (5.7)$$

where, the indices *i, j, k* and *t* refer to the space and time of the standard Yee's cell in the *x-, y-, z*-direction and time step, respectively, and $\Delta x$, $\Delta y$, $\Delta z$, and $\Delta t$ represent the unit space interval in the *x-, y-, z*-direction and unit time interval, respectively. The dielectric parameters are *epx (εx), epy(εx), epz(εx)* and $\mu$ is the permeability.

The characteristic features of the 3D-FDTD method are (a) it is a computation and data-intensive problem performing $O(n^3)$ computations with $O(n^3)$ space requirement, (b) there is data dependency between **E**- and **H**-field computation in time domain, (c) there is no risk of a race condition for each field computation in space domain since the Yee cells can be computed independently for the **E**- and **H**-fields, and (d) a cell (*e.g.* $E_x(i,j,k)$) computation of each field in each direction refers to nearest-neighbors following a *2*-points stencil communication pattern in the space domain. For example, as shown in

Figure 5.15, in each cell, the $x$-directed **E**-filed ($E_x$) is updated with one cell of $x$-directed dielectric parameter ($epx$), two cells of $y$-directed **H**-field ($H_y$) and two cells of $z$-directed **H**-field ($H_z$).

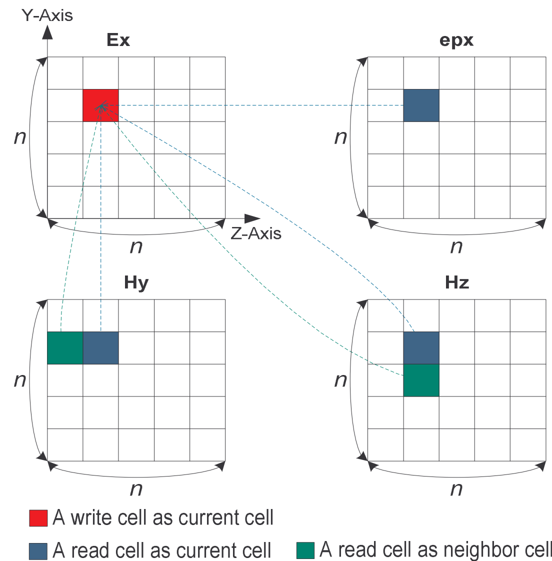



FIGURE 5.15: Example of data dependency in space domain for a cell of $E_x$ computation.

TABLE 5.19: The naïve serial 3D-FDTD algorithm.

| Algorithm**:** The naïve serial 3D-FDTD algorithm |
|---|
| 1: *for t =1 to tmax do* |
| 2: */* E-field computation */* |
| 3: *for i,j,k =1 to imax, jmax, kmax do* |
| 4: *update Electric-field of all-directions using Magnetic-fields* |
| 5: *end for;* |
| 6: */* H-field computation */* |
| 7: *for i,j,k =1 to imax, jmax, kmax do* |
| 8: *update Magnetic-field of all-directions using Electric-fields* |
| 9: *end for* |
| 10: *end for* |

5.4.2.1 Multicore-efficient Implementations

For both the Intel Clovertown and IBM Cell/B.E. platforms, we develop our multicore-efficient implementations following our parallel programming methodology

discussed in Chapter 1. We compare the effectiveness of our approach to that of the naïve parallel implementation based on the naïve serial algorithm shown in Table 5.19.

*Naïve Parallel Implementation:*

The naïve parallel implementation uses row-major order of the array layout for a series of 2D *yz*-slices (*yz*-plane in *x*-direction) of each 3D Yee's cells. The naïve parallel implementation computes all **E**-field computations in space domain first, followed by all **H**-field computations as shown in Table 5.19. The parallelization scheme for *P* cores uses a data partitioning scheme in the *x*-direction (a series of 2D *yz*- slices) as shown in Figure 5.16(a). We implement naïve parallel algorithm using the OpenMP and IBM *libspe* in a straightforward manner on the Intel Clovertown and IBM Cell/B.E. platform respectively, relying mostly on compiler optimizations for performance. Additionally, we synchronize all *P* cores between **E**- and **H**-field computations in the time domain so that adjacent cores can update boundary data.

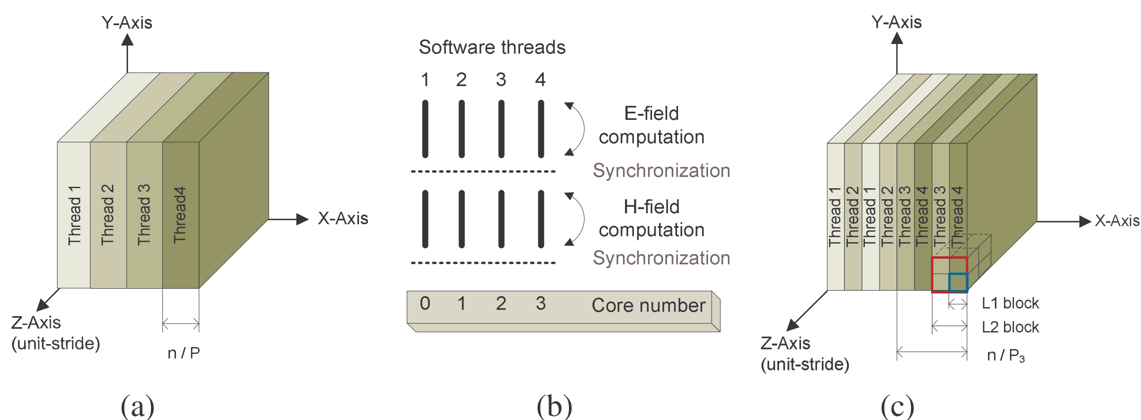



FIGURE 5.16: An example distribution of threads among four cores: (a) Data partitioning scheme for the naïve parallel algorithm; (b) Mapping threads to cores for both (a) and (c) data partitioning schemes; (c) Data partitioning scheme for the multicore efficient algorithm.

*Multicore-efficient Implementation:*

We design our multicore-efficient implementation based on our parallel programming methodology (see Chapter 1) to improve the performance on both the Intel Clovertown and IBM Cell/B.E. platforms. The design steps for our multicore-efficient implementation are the same as for matrix multiplication implementation (see Section 5.4.1).

TABLE 5.20: The summary of our implementation techniques of 3D FDTD for our platforms.

| Optimization Techniques | Multicore Platforms | |
|---|---|---|
| | Intel Clovertown | IBM Cell/B.E. |
| Multi-level blocking | 3-level blocking ($b_0$=4, $b_1$=16, $b_2$=64) | 2-level blocking ($b_0$=4, $b_1$=16) |
| Scheduling | CONTROLLED-PDF (except at register level) | CONTROLLED-PDF (except at register level) |
| Layout Scheme | Row-major ordering | Row-major ordering |
| Multi-buffering | Single buffering | Double buffering |
| Vectorization for register level | Intel x86_64 SSE2 intrinsics with 128-bit registers | IBM Cell/B.E. SPU intrinsics with 128-bit registers |
| Loop unrolling for Vectorization | *unrolling factor*=4 for register-block | *unrolling factor*=4 for register-block |
| Threading | OpenMP | IBM *libspe* |

The summary of our mutlicore-efficient implementation techniques for 3D FDTD is shown in Table 5.20. For both platforms, similar to the naïve parallel implementation, we use a row-major order layout scheme for the series of 2D *yz*-slices in the *x*-direction. Unlike matrix multiplication, the Z-Morton layout for multi-level blockings suffers from performance penalties due to need to access boundary data between nearest-neighbor blocks at each level. As shown in Figure 5.16(c), the 3D blocks are divided into *P* 2D *yz*-slices which are distributed among the *P* cores according to the CONTROLLED-PDF

schedule. Such a parallelization scheme simplifies the data movement between the cores associated with the update of boundary conditions. Within each core a 1DF scheduling scheme is used for blocks at each level. Additionally, for the IBM Cell/B.E. platform, we fetch all the required LS-blocks of data associated with *Ex* (*Hx*), *Ey* (*Hy*), and *Ez* (*Hz*) components initially prior to computation of the **E**-field (see Chapter 4). Similar to matrix multiplication, we modify the scheduling and use vectorization techniques at the register level for both platforms. Further details for register level scheduling are discussed in Section 5.4.2.2.

5.4.2.2 Optimization at Register Level

*Scheduling at Register Level:*

Similar to matrix multiplication, we use a hybrid 1BF-1DF scheduling for register level blocking to better accommodate vectorization. In Figure 5.17, we show the hybrid scheduling scheme on the weighted DAGs for four $E_x$ computations at the register level blocking with $b_0$=*4*. The hybrid scheduling scheme allows for vector pipelining and hence is the preferred scheduling method at the register level.

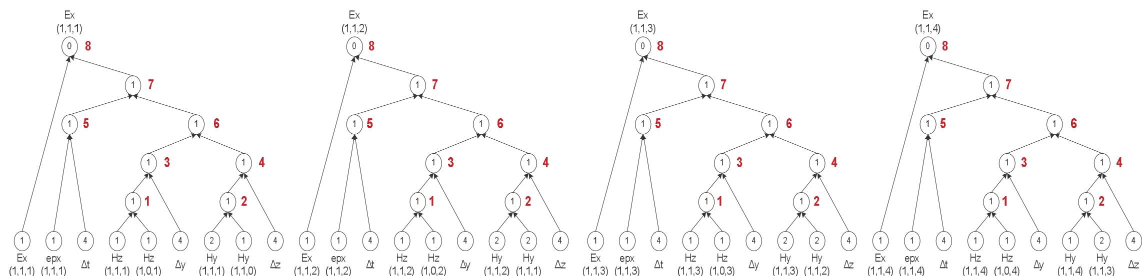



FIGURE 5.17: The hybrid scheduling scheme for the four $E_x$ computations for the register level blocking; the number on the right side of each computational vertices indicates the SIMDize scheduling order.

*Vectorization at Register Level:*

Although SIMD extensions are a cost effective way to exploit data level parallelism, they show poor performance for unaligned (or misaligned) accesses on memory. When there is an attempt to access an unaligned location, it is necessary to perform a realignment process. As shown in Figure 5.18, the access of the boundary values $H_y(i,j,k\text{-}1)$ is un-aligned with respect to the aligned data $H_y(i,j,k)$ where $k$ is in unit-stride direction.

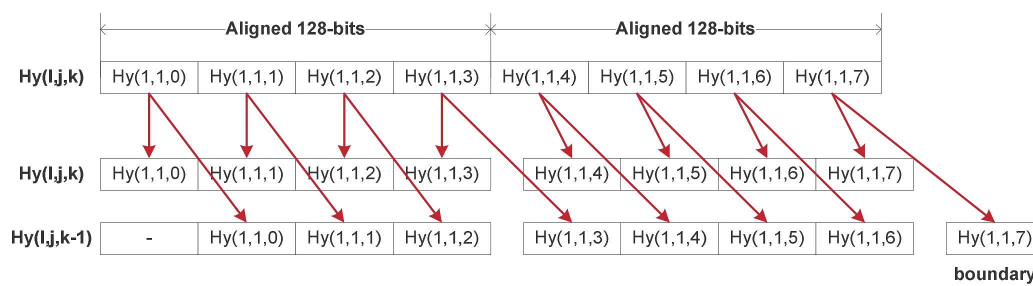



FIGURE 5.18: The example of the conflict alignment of 128-bit vector registers for $H_y(i,j,k)$ and $H_y(i,j,k\text{-}1)$.

Intel x86_64 SSE2 intrinsics supports *_mm_loadu_ps* vector instruction to load unaligned four single-precision floating-point words into a 128-bit register even though it is 4-times slower than *_mm_load_ps* to load aligned four single-precision floating-point into a 128-bit register. For IBM Cell/B.E. platform, we load one row of the aligned $H_y(i,j,k)$ and for the unaligned $H_y(i,j,k\text{-}1)$ data we shift by one in the $k$-direction. The boundary data is stored into the first element of the next register-block in the $k$-direction. Note that we also use loop unrolling technique with *unrolling_factor=4* to implement vectorization at the register block level.

In Table 5.21, we show the pseudo code for the multicore-efficient 3D-FDTD implementation for SPEs. There are a total *15* data sets, including boundary data between LS-blocks, required to compute the **E**-field. A total *13* DMA transfers from the main memory to the LS are required before starting the computation, and *2* block sized spaces for data alignment. Since we use the double-buffering scheme, there are a total *30* LS-blocks in the LS. Three DMA transfers from the LS to the main memory are needed to update the $E_x$, $E_y$, and $E_z$ data in main memory. The boundary values of a current block are stored into the first element of the next LS-block buffer in the *k*-direction as shown in Figure 5.18.

TABLE 5.21: The pseudo code for the SPE 3D-FDTD **E**-field computation using double buffers.

```
Algorithm: SPE thread main()
1:    reserve  tags for MFC (Memory Flow Controller)
2:    initialize   Double buffer for DMA Inputs / outputs
3:    fetch   effective addresses of initial parameters
4:    wait   for a "mailbox" message to start 3D-FDTD computations
5:    for   iter = 1 to ITERATIONS do
6:        // E-field computation
7:        DMA_get() for Ex, Ey, Ez, epx, epy, epz, Hx, Hy, Hz, Hx_j, Hy_i, Hz_i, Hz_j into in-buffers tin
8:        for j=1 to number_blocks do
9:            SWAP_in_buffer() between tin and tin^1
10:           DMA_get() for Ex, Ey, Ez, epx, epy, epz, Hx, Hy, Hz, Hx_j, Hy_i, Hz_i, Hz_j into in-buffers tin
11:           DMA_get_wait() for in-buffer tin
12:           Memcpy (Hx_k, Hx) for aligned Hx_k in k-direction
13:           Memcpy (Hy_k, Hy) for aligned Hy_k in k-direction
14:           DMA_put_wait() for out-buffers tout
15:           Call E-field computation()
16:           DMA_put() for Ex, Ey, Ez into out-buffers tout
17:           SWAP_out_buffer() between out-buffers tout and tout^1
18:       End for
19:       Swap_in_buffer() between tin and tin^1
20:       DMA_get_wait() for in-buffers tin
21:       Memcpy (Hx_k, Hx) for aligned Hx_k in k-direction
22:       Memcpy (Hy_k, Hy) for aligned Hy_k in k-direction
23:       Call  E-field computation()
24:       DMA_put() for Ex, Ey, Ez into out-buffers  tout
25:       DMA_put_wait() for out-buffers tout
26:       synchronize  all SPEs
27:       // H-field computation
28:       // Similar processes as E-field computation
29:       synchronize  all SPEs
30:   end  for
```

5.4.2.3 Performance analysis

We now compare the effectiveness of the multicore-efficient implementation to that of the naïve parallel implementation on both platforms.

*Performance on Intel Clovertown Platform:*

First, we show the effect of the L1 block size and register level scheduling on the performance on a single core of the Intel Clovertown platform. The problem size is fixed at $n = 512$ (corresponding to 16 GB), L2 block size is fixed at $b_2 = 64$ and the register block size is fixed at $b_0 = 4$. 1DF scheduling is used at the L1 and L2 cache levels.

TABLE 5.22: The performance (GFLOPS) for different register level schedules and sizes of L1-block ($b_1$) with fixed size of L2-block ($b_2=64$) and register-block ($b_0=4$) on a single core of the Intel Clovertown platform. We use 1DF scheduling scheme for L1-level and L2-level blocking, and vary the scheduling scheme at the register level.

| | | ***Single core*** | | | | |
|---|---|---|---|---|---|---|
| ***The size of L1-block $b_1$*** | | ***4*** | ***8*** | ***16*** | ***32*** | ***64*** |
| ***The size of $4b_1^3$ in bytes*** | | ***1 KB*** | ***8 KB*** | ***64 KB*** | ***512 KB*** | ***4096 KB*** |
| ***Scheduling scheme at register level*** | ***1DF*** | 0.75 | 0.81 | 0.84 | 0.79 | 0.76 |
| | ***1BF*** | 0.76 | 0.8 | 0.82 | 0.78 | 0.76 |
| | ***Hybrid*** | 0.76 | 0.82 | 0.86 | 0.81 | 0.77 |

As shown in Table 5.22, the highest performance of *0.86* GFLOPS on a single core is achieved with the L1-block size of *16* and a hybrid scheduling scheme at the register level. This is an agreement with the theoretical L1 block size shown in Table 5.20. Note that for *Ex* computations if we attempt to hold all four L1-blocks of cubes *Ex*, *Hy*, *Hz*, and *epx* in the L1 cache as traditional algorithms do, the L1 cache size would have to *64* KB for $b_1=16$.

To study the impact of multi-level blocking scheme on performance, we show the cache miss rates (%) and the system bus utilization (%) with respect to scaling of the

problem size (problem-scaling) and the number of cores (core-scaling) for both naïve parallel and multicore-efficient implementations on the Intel Clovertown platform.

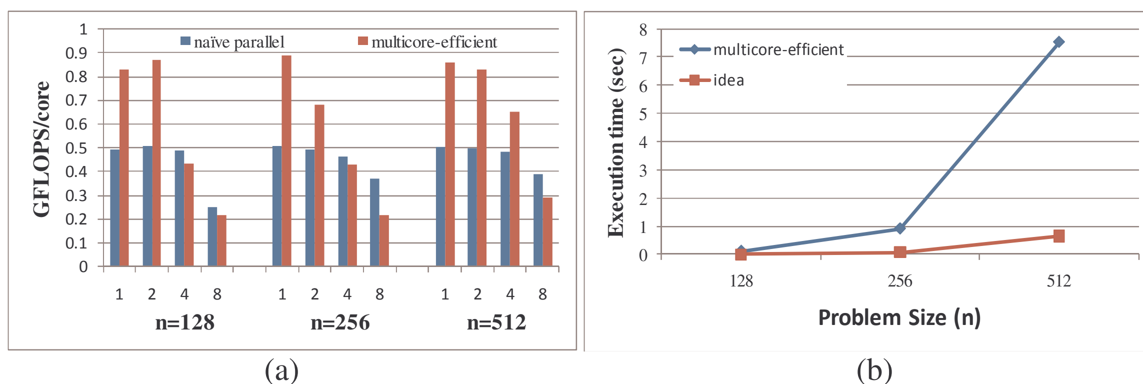
TABLE 5.23: Cache miss rate (%) and system bus bandwidth utilization (%) on Intel Clovertown platform.

| Problem size ($n$) | # cores | L2 cache miss rate (%) | | L1 cache miss rate (%) | | Bus utilization (%) | |
|---|---|---|---|---|---|---|---|
| | | naïve parallel | Multicore-efficient | naïve parallel | Multicore-efficient | naïve parallel | Multicore-efficient |
| *128* | *1* | *0.2* | *1.2* | *0.6* | *1.2* | *25.85* | *15.92* |
| | *2* | *0.5* | *1* | *0.6* | *1.1* | *28.42* | *15.96* |
| | *4* | *0.5* | *0.4* | *0.8* | *0.5* | *27.87* | *16.43* |
| | *8* | *0.4* | *0.4* | *0.8* | *0.4* | *29.15* | *26.92* |
| *256* | *1* | *0.6* | *1.6* | *0.8* | *1.2* | *29.17* | *17.66* |
| | *2* | *0.7* | *1.4* | *0.8* | *1.1* | *27.74* | *33.13* |
| | *4* | *0.7* | *1.1* | *0.8* | *1* | *26.4* | *27.4* |
| | *8* | *0.7* | *0.7* | *0.8* | *0.8* | *29.56* | *31.67* |
| *512* | *1* | *0.9* | *1.5* | *1.2* | *1.2* | *30.61* | *17.35* |
| | *2* | *0.8* | *1.4* | *1.4* | *1.2* | *28.62* | *23.13* |
| | *4* | *0.8* | *1.1* | *1.5* | *1.1* | *27.36* | *26.12* |
| | *8* | *0.7* | *0.9* | *1.5* | *1.1* | *31.91* | *36.87* |

As shown in Table 5.23, our multicore-efficient implementation has a high L2 miss rate although the theoretical analysis of Chapter 3 indicates good temporal locality using multi-level blocking. We postulate that the non-unit-stride penalties due to blocking outweigh the increased temporal locality. The miss rate is highest for the one core case and falls as the number of cores increases since the available L2 cache size increases with the number of cores. The L2 cache miss rate of the naïve parallel implementation on a single core increases with respect to scaling of the problem size while the L2 cache miss rate of the cache-efficient implementation does not vary much. Although we expect the miss rate to be independent of the problem size, we have non-unit stride access along two of the three directions using multi-level blocking for each $E_x$, $E_y$, $E_z$, $H_x$, $H_y$, or $H_z$

computations. The behavior of the L1D cache miss rate is similar to both L2 cache miss rate behavior both for core and problem size scaling. However, note that the L1D cache is more dependent on the L2 blocks and less on the problem size. The higher bus utilization compared to matrix multiplication in Table 5.17 is due to the higher data access to computation ratio for FDTD and possibly more cache coherence traffic due to boundary value sharing.

In Figure 5.19, we show the overall performance in GFLOPS per for problem-scaling and core-scaling for both our naïve parallel and our mutlicore-efficient implementations on the Intel Clovertown platform.




(a) (b)

FIGURE 5.19: Overall performance on Intel platform: (a) Performance in GFLOPS per core; (b) Execution time in seconds on a single core.

In Figure 5.19(a), the naïve parallel implementation shows a performance of about *0.5* GFLOPS on single core which does not vary much with problem scaling. We observe that our multicore-efficient implementation performs almost 1.8 times faster on a single core compared to the naïve algorithm. However, for implementations the performance decreases with core-scaling. For small problems sizes and level-*2* block size $b_2=64$, the parallelism at level-*2* is limited. For each core, we believe that the benefits of

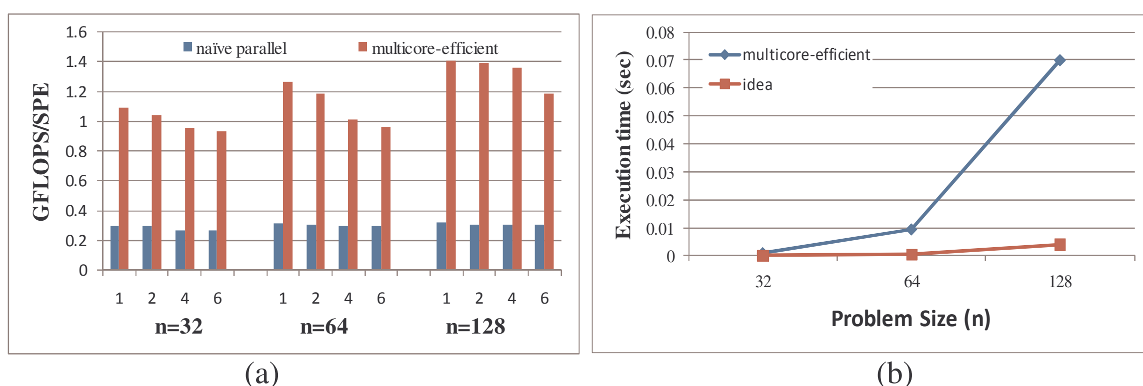
data sharing among the block computations are less than the penalties of non-unit-stride memory access between blocks. Moreover, where the data closely fits into the L2 cache, the benefits of data sharing between level-*2* blocks is not prominent, but the L2 latencies increase with core scaling due to contention on the bus. Since there is only boundary data reused, the performance of both implementation remains almost unchanged as the problem size is scaled on a single core.

In Figure 5.19(b), we compare the performance of the multicore-efficient implementation with an ideal computing performance on the Intel Clovertown processor. The ideal time for 3D-FDTD is the time required for $48n^3$ floating point operations for both **E-** and **H-**field computations at the theoretical peak core performance of the Intel Clovertown processor.

*Performance on IBM cell B.E. Platform:*

Figure 5.20 shows the performance of our implementations on the IBM Cell/B.E. platform. In Figure 5.20(a), we compare the naïve parallel and our multicore-efficient implementations with respect to problem-scaling and core-scaling. The single buffering scheme used in naïve algorithm allows a DMA size per transfer of *32*KB ($b=32$) while the double buffering scheme used in the multicore efficient algorithm limits the block size to *4*KB ($b=16$). Both algorithms require *120* KB of LS for all parameters including boundary data. The naïve parallel implementation shows about *0.3* GFLOPS/SPE with respect to both problem-scaling and core-scaling. We observe that our multicore-efficient implementation achieves *4.6* times speedup over the naïve implementation for $n=128$. Unlike the naïve implementation the multicore-efficient implementation has better performance with problem-scaling as shown in Figure 5.20(a). The increased temporal

locality between blocks increases for large problem sizes. Moreover, unlike the poor scalability on the Intel Clovertown platform with respect to core scaling, we achieve almost linear performance increase with core scaling on the IBM Cell B.E. On the Clovertown processor, the hardware cache coherence policy affects the amount of data that can be shared between the cores. On the other hand, on the IBM Cell BE the explicit control of the boundary data shared between the cores allows for maximal data sharing between the cores.




(a) (b)

FIGURE 5.20: Overall performance on IBM platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on a single SPE.

In Figure 5.20(b), we compare the performance of the multicore-efficient implementation to an ideal computing performance on the IBM Cell/B.E. platform. The ideal computing time for the 3D-FDTD is the time required for $48n^3$ floating point operations for both **E-** and **H-**field computations at the theoretical peak core performance of the SPE.

5.4.3 LU Decomposition

In this section, we describe our multicore efficient implementation of the LU decomposition algorithm. LU is a matrix decomposition algorithm which decomposes a

matrix $A$ into a lower triangular matrix $L$ and an upper triangular matrix $U$ (such that $A=LU$). The computational complexity of the LU decomposition algorithm based on the Gaussian elimination method shown in Table 5.24 is $O(n^3)$ while the data space requirements are $O(n^2)$ for a $n \times n$ square matrix $A$. The data dependency of the algorithm is shown in Figure 5.21. Notice that we use only one matrix $A$, with the triangular matrices $L$ and $U$ overwriting matrix $A$.

TABLE 5.24: The LU decomposition based on Gaussian elimination method with $n \times n$ square matrix $A$.

| Algorithm: LU decomposition based on Gaussian eliminate method |
|---|
| 1: *for (k=0; k<n; k++)*<br>2: *for (i=(k+1); i<n; i++)*<br>3: *A[i][k] = A[i][k] / A[k][k];*<br>4: *for (j=(k+1); j<n; j++)*<br>5: *A[i][j] = A[i][j] – A[i][k]*A[k][j];*<br>6: *end for*<br>7: *end for*<br>8: *end for* |

Let $A^k$ denote the sub-matrix which is the computing domain including only elements *a(i,j)* with $k < i \leq n$ and $k \leq j \leq n$ at $k$ iteration step, where *i, j* and $k$ refer to $i^{th}$ row elements, $j^{th}$ column elements and $k^{th}$ element elimination step, respectively. As shown in Figure 5.21, only the sub-matrix $A^k$ is updated at $k$ iteration step. As shown in Table 5.24, the computation scheduling of 1DF in $k$ iteration step consists of first updating the element in the first column of $A^k$ as shown in Line 3 of the pseudo-code of Table 5.24, and then updating the remaining elements of *a(i,j)* of $A^k$ as shown in Line 5 of the pseudo-code of Table 5.24. Thus, for computing the element a*(i,j)*, the updated $k$ elements of the $i^{th}$ row and $j^{th}$ column are required.

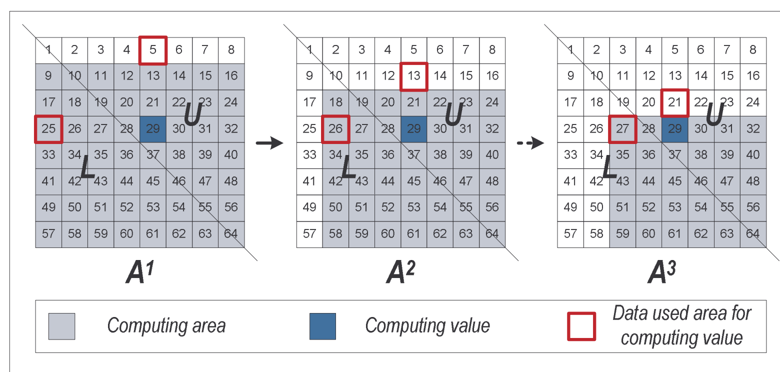



FIGURE 5.21: The data dependency of the LU decomposition based on Gaussian elimination method.

5.4.3.1 Multicore-efficient Implementations

For both the Intel Clovertown and IBM Cell/B.E. platforms, we develop our multicore-efficient implementations following our parallel programming methodology discussed in Chapter 1. We compare the effectiveness of our approaches to that of the naïve parallel implementation based on the Gaussian-elimination method shown in Table 5.24.

*Naïve Parallel Implementation:*

The naïve parallel implementation uses a row-major order of the array layout for matrix $A$. For the parallelization scheme for $P$ cores, we use the row-wise one dimensional partitioning of the sub-matrix $A^k$ in $k$ iteration step. Notice that the $k^{th}$ row in $A^k$ is shared for all $P$ cores at $k$ iteration step. Then the computing order of each partition follows the 1DF scheduling which is the computing order of the serial algorithm shown in Table 5.24. We use *OpenMP* parallel programming library for the Intel Clovertown platform and the IBM *libspe* for the IBM Cell/B.E. platform (see Chapter 2). Since DMA transfer of data is required on the IBM Cell/B.E. platform, we use the single buffer scheme without considering prefetching (see Chapter 4). A total of 4 buffers are used

(three reads and one write). The size of each buffer is chosen to be 16KB corresponding to the maximum size of the DMA transfer. The total buffer size of 64KB is less than the available size of the SPE LS.

*Multicore-efficient Implementation:*

We design our multicore-efficient implementation based on our parallel programming methodology (see Chapter 1) to improve the performance on both platforms. The design steps for our multicore-efficient implementations are the same as for matrix multiplication implementations (see Section 5.4.1). The summary of the multicore-efficient implementation techniques for LU decomposition is shown in Table 5.25.

TABLE 5.25: The summary of our implementation techniques of LU decomposition for our platforms.

| Optimization Techniques | Multicore Platforms | |
|---|---|---|
| | Intel Clovertown | IBM Cell/B.E. |
| Multi-level blocking | 3-level blocking ($b_0$=4, $b_1$=64, $b_2$=256) | 2-level blocking ($b_0$=4, $b_1$=32) |
| Scheduling | CONTROLLED-PDF (except at register level) | CONTROLLED-PDF (except at register level) |
| Layout Scheme | Z-Morton ordering | Z-Morton ordering |
| Multi-buffering | Single buffering | Double buffering |
| Vectorization for register level | Intel x86_64 SSE2 intrinsics with 128-bit registers | IBM Cell/B.E. SPU intrinsics with 128-bit registers |
| Loop unrolling for Vectorization | *unrolling factor*=4 for register-block | *unrolling factor*=4 for register-block |
| Threading | OpenMP | IBM *libspe* |

For both platforms, we use Z-Morton order layout scheme of the matrix $A$ for $d$-level blocking to avoid the penalties of non-unit-stride memory access at each level

blocking. Here $d=3$ and $d=2$ for Intel Clovertown and IBM Cell/B.E. platform, respectively.

Unlike matrix multiplication, the blocks for LU decomposition have different types of computation. Let $A^k$ denote the sub-matrix which is the computing area including only block-$(i,j)$ with $k \leq i,j \leq n$ at $k$ iteration step, where $i$, $j$ and $k$ refer to $i^{th}$ row blocks, $j^{th}$ column blocks and $k^{th}$ block elimination step, respectively. At level-$d$ the four different types of computations (LUD-, L-, U-, and M-block) in sub-matrix $A^k$ is shown in Figure 5.22. Note that the blocks with the same type (same color) can be executed in parallel in each elimination step. Depending on the parent block, the computations of the remaining blocks are a subset of the four types of computations described above. The computations details for the different types of blocks are discussed later. For parallelism the $d$-level blocks among $P$ cores, the following steps are used – 1) First, the algorithm starts by processing the LU-block on one core, 2) then, the U-blocks distributed among $P$ cores are updated, 3) finally, the remaining blocks (L-blocks and M-blocks), which are distributed using the row-wise one-dimensional partitioning of sub-matrix $A^k$ among $P$ cores, are updated in each $k$ iteration step. The $P$ cores synchronize at each step. Note that the algorithm continues to iterate until it processes the last block as a LU-block. Within each core, the computing order of the partitioned blocks at each level follows the 1DF scheduling. Additionally, for the IBM Cell/B.E. platform, the L-block is stored until all the computations of all M-blocks at the same row are completed (refer Chapter 4). Similar to matrix multiplication, we modify the scheduling and use vectorization techniques at the register level for both platforms. Further details for scheduling at the register level are discussed in Section 5.4.3.2.

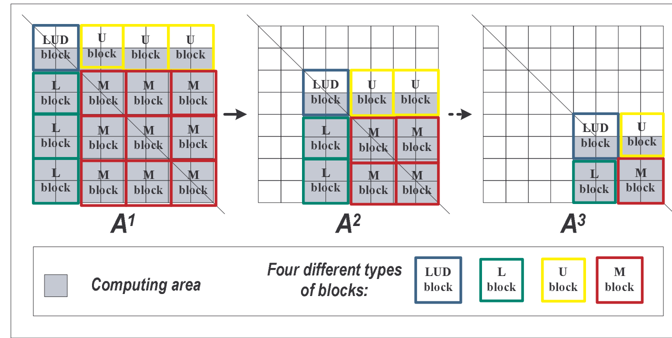



FIGURE 5.22: The block partition with the four different types in sub-matrix $A^k$ at *d*-level.

TABLE 5.26: The different tasks of the four blocks at *d*-level.

| | LUD-block | L-block | U-block | M-block |
|---|---|---|---|---|
| First Row | No computation | Multiplication (×) | No computation | Multiplication (×) |
| First Column | Divide (/) | Divide (/) | Multiplication (×) | Multiplication (×) |
| Remaining elements | Multiplication (×) | Multiplication (×) | Multiplication (×) | Multiplication (×) |
| Data dependency | Local LUD-block | LUD-block<br>Local L-block | LUD-block<br>Local U-block | L-block in same row<br>U-block in same column<br>Local M-block |

As mentioned previously, the four different types of blocks at *d*-level for LU decomposition have different tasks as shown in Table 5.26. The LUD-block performs the same computation as LU decomposition, and it requires only data of the current LUD-block. The L-block performs similar computations as the LUD-block on all rows excluding the first row. The first row elements are computed by multiplying the corresponding element in the LUD block with the element in the previous column (same row) of the L-block. Thus the L-block computation requires data of one LUD-block and the current L-block. Excluding the first row, the U-block elements are computed by multiplying the corresponding element of the LUD block with the element in the previous

row (same column) of the U-block. Thus the U-block computation requires data of one LUD-block and the current U-block. The M-block is computed by multiplying the corresponding element of the L-block with the corresponding element of the U-block. Thus the M-block computation requires one L-block, one U-block, and the current M-block. The example of the data dependency is shown in Figure 5.23.

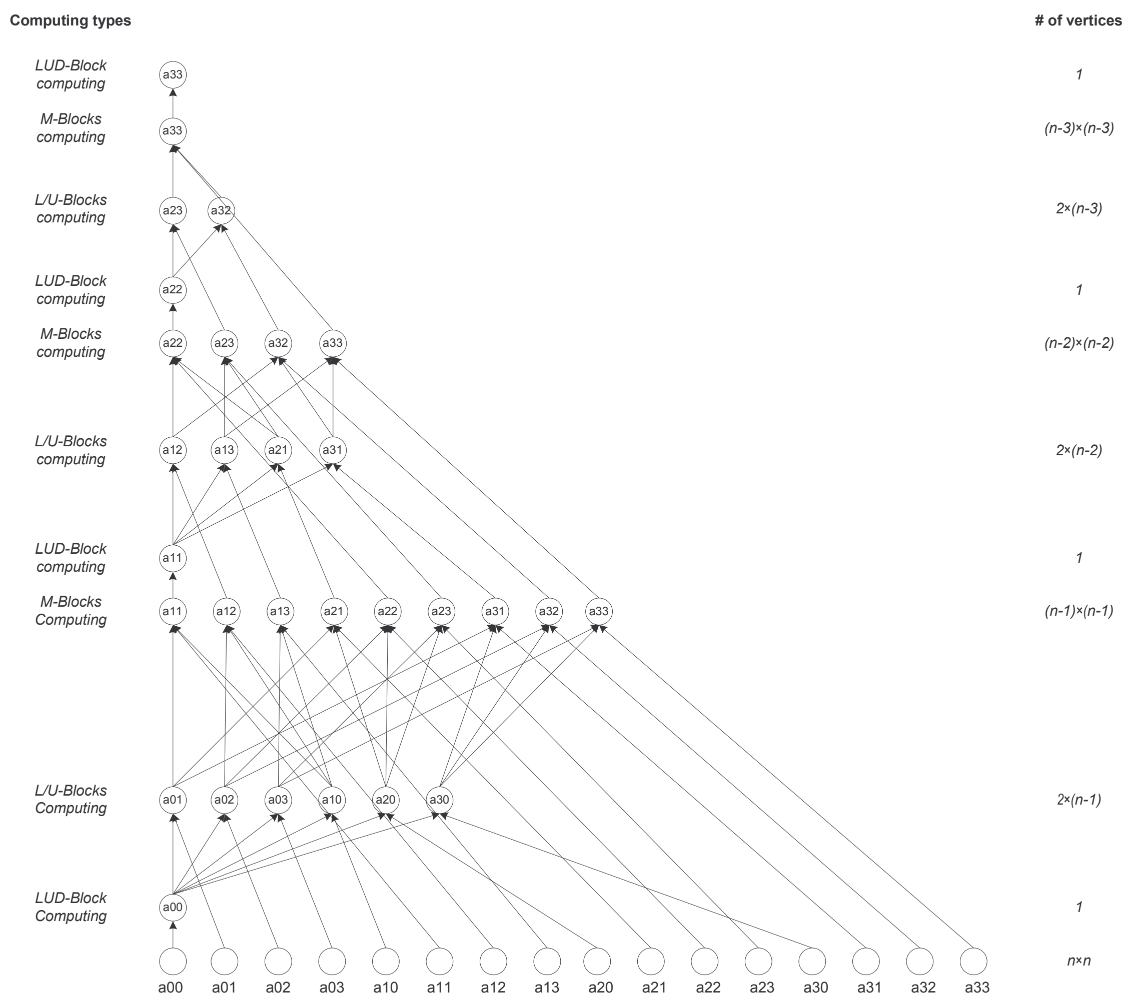



FIGURE 5.23: The example of data dependency of LU decomposition for a matrix $A$ with 4×4 blocks.

As shown in Figure 5.23, for the LU decomposition of the matrix *A,* there are a total of *N* elimination steps with $N \times N$ blocks (elements), where *N* is the number of blocks (elements) in row (column). Each elimination step except the last requires three synchronizations between the computations of the LUD-block, L- and U-blocks, and M-block. Note that the L- and U-blocks can be computed in parallel and require no synchronization between them. The last $k^{th}$ elimination step requires only an LUD-block computation of the sub-matrix $A^k$. The depth of the DAG is $(3 \times (N - 1) + 1)$. Unlike, the matrix multiplication or FDTD algorithms, the LU decomposition can be described with only one DAG. Thus, the parallelism at each level is limited to the breadth of the DAG.

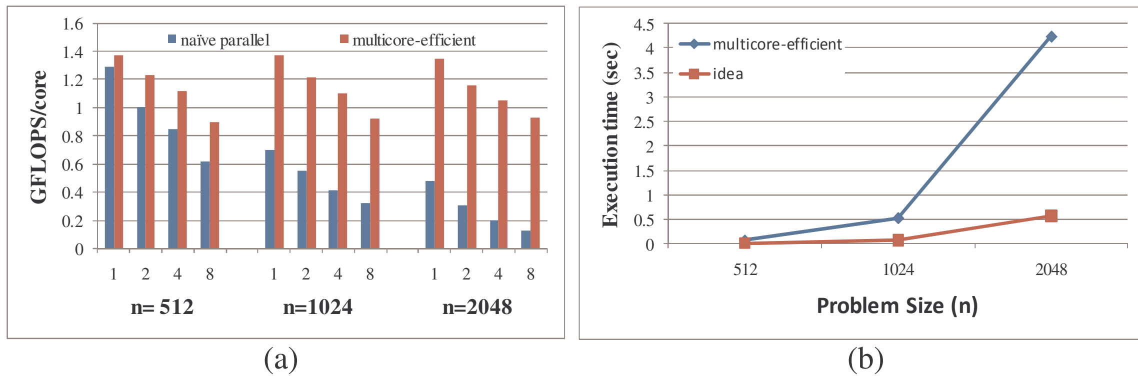
5.4.3.2 Optimization at Register Level

For the register level, we modify the scheduling scheme and use vectorization only for the M-block while other types of blocks use the 1DF scheduling scheme. The implementation of M-block vectorization is similar to matrix multiplication (see Figure 5.12 in Section 5.4.1). The M-block vectorization uses a subtract operation instead of an addition operation. We also use loop unrolling for register level blocking.

5.4.3.3 Performance Analysis

*Performance on Intel Clovertown Platform:*

In Figure 5.24, we show the overall performance in GFLOPS per core for problem-scaling and core-scaling for both our naïve parallel and multicore-efficient implementations on the Intel Clovertown platform.




(a) (b)

FIGURE 5.24: Overall performance on Intel platform: (a) Performance in GFLOPS per core; (b) Execution times in seconds on a single core.

In Figure 5.24(a), for problem-scaling, the naïve parallel implementation shows a performance of *1.3* GFLOPS on single core with the smaller size of problem (*n=512*). Here the total problem size of *1* MB for the matrix *A* can fit into the single L2 cache. However, the naïve parallel implementation shows *0.7* and *0.5* GFLOPS for *n=1024* (*4* MB) and *n=2048* (*16* MB), respectively. Our multicore-efficient implementation shows a performance of *1.4* GFLOPS on single core which does not vary much with the problem-scaling. We expect this performance trend to continue for larger sized ($n > 2048$) problems. Our multicore-efficient implementation achieves *2.8* times speedup over the naïve parallel implementation for *n=2048*. However, for both implementations, the performance with respect to core-scaling shows poor scalability corresponding to the reduced parallelism associated with the shrinking computing area (sub-matrix $A^k$) in each *k* elimination step. Moreover, the number of available blocks in each parallel phase is usually not exactly divisible by the number of cores. This creates a load imbalance at each elimination step, with the associated overhead accumulating over time.

In Figure 5.24(b), we compare the efficiency of the multicore-efficient implementation with the ideal computing performance on the Intel Clovertown platform.

The ideal time for LU decomposition is the time required for $O(n^3)$ floating point operations at the theoretical peak core performance of a single core of the Intel Clovertown.

*Performance on IBM Cell/B.E. Platform:*

In Figure 5.25, we show the overall performance in GFLOPS per SPE for problem-scaling and core-scaling for both naïve parallel and multicore-efficient implementations on the IBM Cell/B.E. platform.

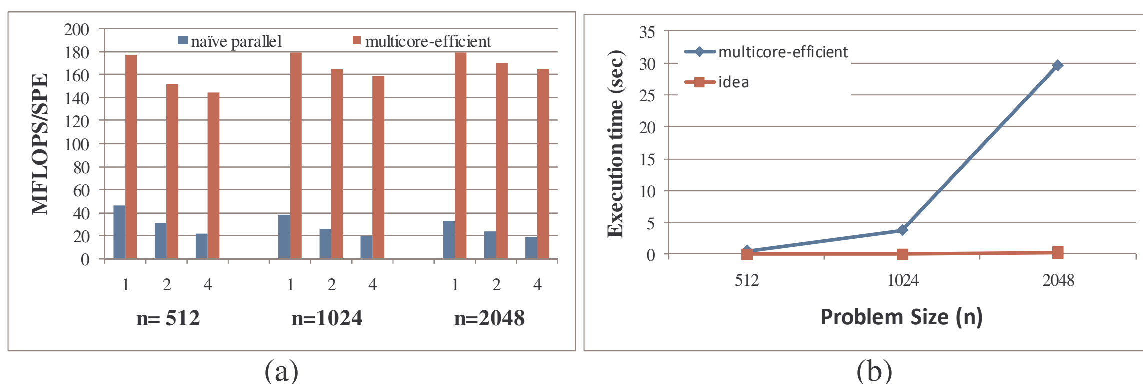



FIGURE 5.25: Overall performance on IBM Cell BE platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on a single SPE.

Figure 5.25(a) shows a single SPE performance of *40* MFLOPS (*n=512*) for the naïve parallel implementation and *180* MFLOPS (*n=512*) for the multicore efficient implementation. For both implementations performance does not vary much with problem-scaling. We expect this performance trend to continue for larger sized (*n>2048*) problems. We observe that our multicore-efficient implementation achieves almost *2.8* times speedup over the naïve parallel implementation for all problem sizes. For both the naïve and multicore efficient implementation, although the performance with respect to core-scaling degrades with increasing number of cores, core-scalability is better than the

Intel Colvertown platform. The smaller LS-blocks (*32×32*) used on the IBM Cell BE allows for better load balance compared to the larger blocks of the Intel Clovertown (*64×64*) platform.

In Figure 5.25(b), we compare the efficiency of the multicore-efficient implementation with that the ideal computing performance on the IBM Cell BE platform. The ideal time for LU decomposition is the time required for $O(n^3)$ floating point operations with the theoretical peak core performance on single core of the IBM Cell BE.

For both platforms, the dynamic repartitioning can be used to reduce load imbalance at each step.

### 5.4.4 Power Flow Solver based on Gauss-Seidel method (PFS-GS)

In this section, we illustrate and analyze our multicore-efficient parallel implementation of the Power Flow Solver based on Gauss-Seidel method (PFS-GS). It determines the voltage magnitude and phase angle for each bus (network node) in a power system network under balanced three-phase steady-state conditions. PFS-GS is modeled as a set of buses (network nodes) interconnected by transmission branches (network links) expressed as [16, 18]:

$$\forall_k \left\{ \overline{V}_k \leftarrow \overline{G}(\overline{V}_k) - \sum_{\substack{n=1 \\ n \neq k}}^{N} \overline{H}(\overline{V}_n) \right\} \tag{5.8}$$

where, $\overline{G}(\overline{x}) = \overline{S_k} / (\overline{Y_{kk}} \times \overline{x}^*)$, $\overline{H}(\overline{x}) = (\overline{x} \times \overline{Y_{kn}}) / \overline{Y_{kk}}$ in which $\overline{V_k}$ and $\overline{S_k}$ represent the complex voltage and the complex power at each bus *k*, respectively, and $\overline{Y_{kn}}$ is *admittance* between bus *k* and *n*. To compute the line current in a branch $\overline{H}$ in the power network, we calculate admittance and line current injections from the source and

destination buses. This calculation depends on the line currents of all incident branches of the source and destination buses.

The buses are categorized as SWING, PQ and PV. The SWING bus is a node that is designated to compensate residual error and is also used for power generators which control both real and reactive power injections. The PQ bus is a node that has both constant real and reactive power injections. The PV bus is a node that has constant real power injection but can control reactive power injections. The voltage and power calculations of a bus do not depend on other bus computations as the line currents are calculated during the branch computations.

The data dependency of a sample power network with *5* buses and *5* branches is shown in Figure 5.26, and the pseudo-code of naïve serial algorithm for PFS-GS is shown in Table 5.27.

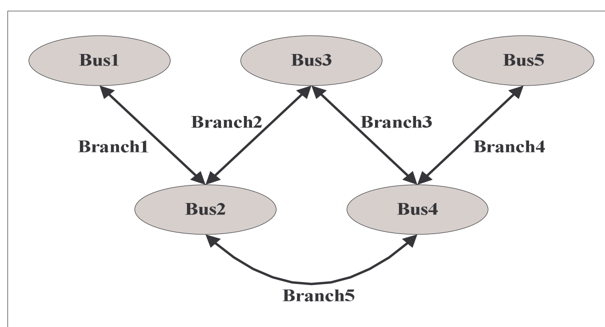



FIGURE 5.26: The sample power network computation with 5 buses and 5 branches.

As shown in Table 5.27, the naïve serial algorithm first performs all branch computations and then proceeds to the bus computations. For each branch computation, data from the two buses connected by the branch is required. The branch computations can be represented as matrix vector computations involving the multiplication of the

TABLE 5.27: Pseudo-code of naïve serial algorithm for the bus and branch computation.

Inputs: Bus voltage $V_{old}$, Bus Power $S_{old,}$ Admittance matrix $Y$, Acceleration factor $ACC$, Reactive power limits $Q_{max}$ and $Q_{min}$, Bus shunt conductance $G$ and reactance $B$ and Bus type (PV/PQ/Swing)

Outputs: Branch: self admittance $Y_s$, self current $I_s$
Bus: new bus voltage $V_{new}$, New bus power $S_{new}$

Prototype Gauss-Seidel Solver Algorithm

```
1:   For iter = 1 to ITERATION Do
2:     Call Branch Function
3:     Call Bus Function
4:     Check CONVERGENCE; Continue if necessary
5:   End for
```

*Branch function()*

```
1:   For each branch between bus n and k on the network Do
2:     Calculate the self admittance vector term Ys with admittance matrix Y and voltage vector V_old
3:     Calculate the self current vector term Is with admittance matrix Y and voltage vector V_old
4:   End for
```

*Bus function()*

```
1:   For each bus on the network Do
2:     If non-zero admittance matrix OR swing bus type Then
3:       Case bus type Of
4:         PV bus:
5:           Calculate new reactive power Q_new using Gauss-Seidel Algorithm
6:           If new reactive power Q_new exceeds Qlimit [Q_min, Q_max] Then
7:             Set new reactive power Q_new with Qlimit = [Q_max,Q_min]
8:             Continue Calculate new voltage V_new as PQ bus
9:           Else
10:            Calculate intermediate voltage V_int with new reactive power Q_new using GS
11:            Calculate the new voltage V_new with Magnitude of V_old and phase angle of V_int
12:            Break
13:          End if
14:        PQ bus:
15:          If non-zero current voltage V_old Then
16:            Calculate the intermediate voltage V_int with reactive power Q_old for PQ or Q_new for PV using GS
17:            Calculate the new voltage V_new = V_old + (V_int – V_old)×ACC
18:            Break
19:          End if
20:        SWING bus:
21:          Calculate the new power injection S_new
22:          Break
23:        OTHERS:
24:          Break
25:      End Case
26:    End if
27:  End for
```

admittance matrix term ***Y*** (current matrix term ***I***) by voltage vector ***V*** to compute the self admittance vector ***Ys*** (current vector ***Is***) for all buses. The bus computations involve update of the voltage and power vectors and are independent of each other. Thus, in principle all buses can be computed in parallel without considering any data dependency as the self currents and admittances of all buses are calculated during the branch computations.

5.4.4.1 Multicore-efficient Implementations

For both the Intel Clovertown and IBM Cell/B.E. platforms, we develop our multicore-efficient implementations following our parallel programming methodology discussed in Chapter 1. We compare the effectiveness of our approach to that of the naïve parallel implementation.

*Naïve Parallel Implementation:*

The naïve parallel implementation uses a row-major order of the array layout for the matrices (admittance matrix ***Y*** and current matrix ***I***). For the parallelization scheme for *P* cores, we use row-wise one-dimensional partitioning of the matrices for the branch computations, and we divide the total number of buses by *P* for the bus computations. Then the computing order of each partition follows the 1DF scheduling. Additionally, we synchronize all *P* cores between branch and bus computations. We use OpenMP parallel programming library for the Intel Clovertown platform and the IBM *libspe* for the IBM Cell/B.E. platform (see Chapter 2). Since DMA transfer of data is required on the IBM Cell/B.E. platform, we use the single buffer scheme without considering caching and prefetching (see Chapter 4). Note that each core (or SPE) performs assigned branch

computation followed by the bus computation. The total numbers of data transfers per core (or SPE) for the branch and bus computations are given by:

$$Branch: \quad (5 \times N_{BUS} + 4) \times N_{BUS}/P \tag{5.9}$$

$$Bus: \quad (18 \times N_{BUS})/P \tag{5.10}$$

And, the total numbers of computations per core (or SPE) for the branch and bus computations are given by:

$$Branch: \quad (14 \times N_{BUS} \times N_{BUS})/P \tag{5.11}$$

$$Bus: \quad (81 \times N_{PV}^{i} + 66 \times N_{PQ}^{i} + 16 \times N_{SWING}^{i}) \tag{5.12}$$

where, $N_{BUS}$ is the number of buses in the network, $P$ is the number of used cores (or SPEs), and $N_{PV}{}^{i}$, $N_{PQ}{}^{i}$, and $N_{SWING}{}^{i}$ represent the number of total PV, PQ, and SWING buses assigned to core$^{i}$ (or SPE$^{i}$), were $1 \leq i \leq P$.

*Multicore-efficient Implementation:*

We design our mutlicore-efficient implementation based on our programming methodology (see Chapter 1) to improve the performance on both platforms. The design steps for our multicore-efficient implementations are similar to the multicore-efficient matrix multiplication (see Section 5.4.1). The summary of the multicore-efficient implementation techniques for PFS-GS is shown in Table 5.28.

For both platforms, we use the Z-Morton order layout scheme of the matrices ***Y*** and ***I*** for *d*-level blocking to avoid the penalties of non-unit-stride memory access. Here *d=3* and *d=2* for Intel Clovertown and IBM Cell/B.E. platform, respectively. We use the same parallelization scheme for *P* cores as the naïve parallel implementation. For IBM Cell/B.E. platform, we use double-buffering scheme with considering caching and prefetching (see Chapter 4) for DMA transfer. However, there is no caching advantage

for bus computation since there is no data reused between buses. Similar to the previously described algorithms, we modify the scheduling and use vectorization techniques at the register level for both platforms. Further details for scheduling and vectorization at the register level are discussed in Section 5.4.4.2.

TABLE 5.28: The summary of our implementation techniques of PFS_GS for our platforms.

| Optimization Techniques | Multicore Platforms | |
|---|---|---|
| | Intel Clovertown | IBM Cell/B.E. |
| Multi-level blocking | 3-level blocking ($b_0$=4, $b_1$=32, $b_2$=128) | 2-level blocking ($b_0$=4, $b_1$=32) |
| Scheduling | CONTROLLED-PDF (except at register level) | CONTROLLED-PDF (except at register level) |
| Layout Scheme | Z-Morton ordering | Z-Morton ordering |
| Multi-buffering | Single buffering | Double buffering |
| Vectorization for register level | Intel x86_64 SSE2 intrinsics with 128-bit registers | IBM Cell/B.E. SPU intrinsics with 128-bit registers |
| Loop unrolling for Vectorization | *unrolling factor*=4 for register-block | *unrolling factor*=4 for register-block |
| Threading | OpenMP | IBM *libspe* |

### 5.4.4.2 Optimization at Register Level

*Scheduling and Vectorization at the Register Level:*

As mentioned previously, the branch computations use matrix-vector multiplication by multiplying the admittance matrix by the voltage vector to compute the self admittance vector and the self current vector for all buses. Hence we use similar scheduling and vectorization scheme as the matrix multiplication algorithm at the register level for branch computations (see Section 5.4.1.2).

For bus computations at register level, all bus types share some of the computation with different input data. For example, the intermediate bus voltage

calculation is computed for both PV and PQ buses using the Gauss-Seidel method. We can therefore take advantage of the shared computations between the different bus types. However, the different types of buses require different computations, which can lead to mispredictions in conditional statements. Each bus type executes conditional statements depending on the voltage and power values. The mispredictions in these conditional executions can lead to poor performance due to hardware pipeline stalls and limited vectorization on the multicore platforms. Therefore, we implement a *vectorized unified-bus-computation module* for all bus types to avoid such undesirable performance degradation and to take advantage of the shared computations. For IBM Cell/B.E. platform with IBM SPU intrinsics, the *vectorized unified-bus-computation module* with single-precision floating-point data is shown in Figure 5.27. Each vector bus computations consist of *4* scalar bus computations drawn from same or different bus types depending on the network.

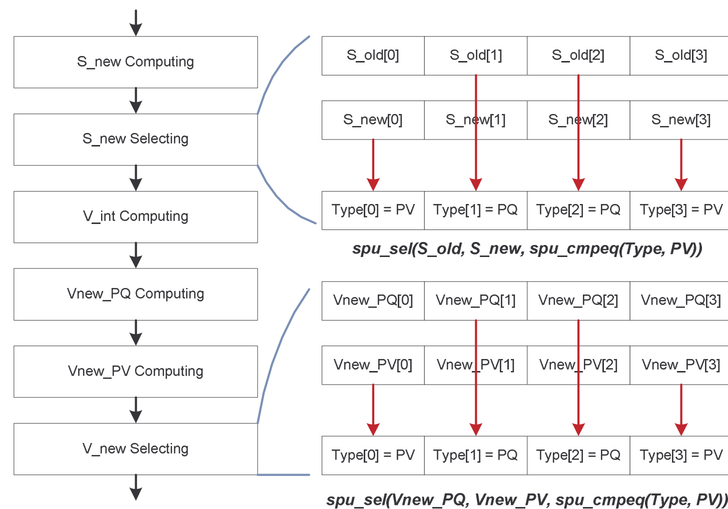



FIGURE 5.27: *Vectorized Unified-Bus-Computation* Module.

As shown in Figure 5.27, we design the *vectorized unified-bus-computation module* to eliminate conditional statements and to take advantage of the shared computations through the following steps – (1) compute intermediate power *S_new* which is used for all three types of buses, (2) update new power *S* for each bus, (3) compute intermediate voltage *V_int* which is used for PV and PQ bus, (4) compute new PQ voltage *Vnew_PQ*, (5) compute new PV voltage *Vnew_PV*, (6) and then update new voltage *V* for each bus. We use *spe_sel* intrinsic to select the individual bus power and voltage from the vector registers. The select-bits instruction is the key in eliminating branches for simple control-flow statements (for example, *if* and *if-then* constructs). An *if-then-else* statement can be made branchless by computing the results of both the branch conditions, and then the select-bits choose the result depending on the evaluation of the *if-then-else* statement. If computing both the results costs less than a mispredicted branch, then we have additional saving. Also, the *select_bits* enables efficient vectorization of *if-then-else* statements. For the Intel Clovertown platform with Intel x86_64 SSE2 intrinsics, the design steps of the *vectorized unified-bus-computation module* are same as those for the IBM Cell/B.E. platform. For our multicore-efficient implementation using the *vectorized unified-bus-computation* at register revel, the total number of all bus computations per core is given by:

$$Bus: \qquad 86 \times (N_{PV}^{i} + N_{PQ}^{i})/4 \ + (8 \times N_{SWING}^{i}) \tag{5.13}$$

where, $N_{PV}{}^{i}$, $N_{PQ}{}^{i}$, and $N_{SWING}{}^{i}$ represent the number of total PV, PQ, and SWING buses assigned to $core^{i}$ ($SPE^{i}$), were $1 \leq i \leq P$. Note that the numbers of data requirement and branch computation are same as the naïve parallel implementation.

TABLE 5.29: Pseudo code of the multicore-efficient implementation for PFS-GS on the IBM Cell/B.E. platform.

```
Algorithm: PPE main()
1:    Initialize   Branch and Bus data
2:    Create   SPE threads for PFS-GS computations
3:    Send   "mailboxes" to instruct SPEs to SPE threads
4:    Wait   until PFS-GS computation of all SPEs is done
5:    Terminate   SPE threads
```

```
SPE thread main()
1:    Reserve  tags for MFC (Memory Flow Controller)
2:    Initialize   Double buffer for DMA Inputs/outputs
3:    Fetch   effective addresses of initial parameters
4:    Wait   for a "mailbox" message to start PFS-GS computations
5:    For   iter = 1 to ITERATIONS Do
6:        Call   SPE Branch Function
7:        Synchronize  all SPEs
8:        Call   SPE Bus Function
9:        Synchronize  all SPEs
10:       Check  CONVERGENCE; Continue if necessary
11:   End  for
```

```
SPE Branch ()
1:    For i=0 to BLOCKS Do
2:        DMA FETCH for M input data (Ys, YVs) of i block
3:        DMA WAIT for input data (Ys, YVs) of M buses
4:        For j=0 to M Do
5:            DMA FETCH for M input data (Ymatrix, Bmatrix, V)
6:            For k=1 to (TOTAL_BUSES/M) Do
7:                SWAP input buffers of M input data (Y, B, V)
8:                DMA FETCH for M input data (Y, B, V) of k
9:                DMA WAIT for M input data (Y, B, V) of k-1
10:               Computing Branch of k-1
11:               Update M output data (Ys, Is) of i block
12:           End for
13:           DMA STORE for M output data (Ys, Is) of i block
14:           DMA WAIT for M output data to main memory
15:       End for
16:   End for
```

```
SPE Bus ()
1:    DMA FETCH for M input data (Type, V, S, G, B, Qmax, Qmin, Ys, Is)
2:    For i=1 to BLOKCS Do
3:        SWAP input buffers of M input data (Type, V, S, G, B, Qmax, Qmin, Ys, Is) of i block
4:        DMA WAIT for input data (Type, V, S, G, B, Qmax, Qmin, Ys, Is) of (i-1) block
5:        DMA WAIT for output data (V, S, Ys, Is) of (i-1) block
6:        Computing  the vectorized unified-bus-computation of (i-1) block
7:        DMA STORE for M output data (V, S, Ys, Is) of (i-1) block
8:        SWAP output buffers
9:    End for
10:   SWAP input buffers
11:   DMA WAIT for M input data (Type, V, S, G, B, Qmax, Qmin, Ys, Is) of BLOCKS block
12:   Computing  the vectorized unified-bus-computation of (BLOCKS) block
13:   DMA STORE for M output data (V, S) of (BLOCKS) block
```

*Pseudo Code for Multicore-efficient Implementation:*

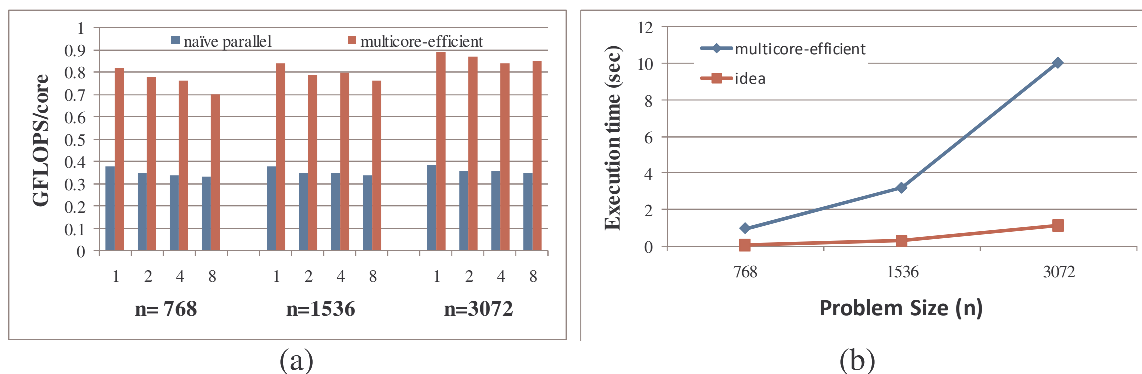
In Table 5.29, we show the pseudo code for multicore-efficient implementation on the IBM Cell/B.E. platform.

5.4.4.3 Performance Analysis

For all implementations, we setup a network configuration with *n* number of buses and $n \times n$ branches. The network includes *60%* of PQ buses, *40%* of PV buses, and one SWING bus. The network is simulated for *100* iterations.

*Performance on Intel Clovertown Platform:*

In Figure 5.28, we show the overall performance in GFLOPS per core for problem-scaling and core-scaling for both the naïve parallel and our multicore-efficient implementation on the Intel Clovertown platform.




(a) (b)

FIGURE 5.28: Overall performance on Intel Clovertown platform: (a) Performance in GFLOPS per core; (b) Execution time in seconds on a single core.

In Figure 5.28(a), the naïve parallel implementation shows performance of *0.38* GFLOPS on single core which does not vary much with respect to problem-scaling. We observe that for *n=3072*, our multicore-efficient implementation performs *2.3* times faster on a single core compared to the naïve parallel algorithm. In all of our

implementations, we observe an almost linear scaling of performance with respect to the core-scaling.

In Figure 5.28(b), we compare the efficiency of the multicore-efficient implementation with an ideal computing performance on the Intel Clovertown platform. The ideal time for PFS-GS is the time required for only computations following Equations 5.11 and 5.12 assuming theoretical peak core performance on a single core of the Intel Clovertown.

*Performance on IBM Cell/B.E. Platform:*

In Table 5.30, we show the effect of the LS-block size on the performance of single SPE of the IBM Cell/B.E. platform for branch and bus computations. The problem size is fixed at $n=768$. The best case performance is achieved with LS-block size of $b_1=32$ and $b_1=16$ for branch and bus computations respectively.

TABLE 5.30: GFLOPS with varying DMA transfer size in bytes on single SPE.

| LS-block size $b_1$ | Performance in GFLOPS | | |
|---|---|---|---|
| | Bus | Branch | Bus + Branch |
| *8* | *1.97* | *2.11* | *2.11* |
| *16* | *2.06* | *2.37* | *2.37* |
| *32* | *2.02* | *2.46* | *2.47* |
| *64* | *2.04* | *2.42* | *2.44* |
| *128* | *2.05* | *2.37* | *2.37* |

Table 5.31 shows the speedup and percentage computation times for four different implementations of the PFS-GS algorithm on a single SPE. Here the problem size is $n=768$. Our multicore-efficient implementation, which combines double-buffering scheme and *vectorized unified-bus-computation* module, achieves the highest performance improvement of *9.2* times speedup for bus computations, and *5.3* times

speedup in branch computations with respect to the naïve serial implementation. Both bus and branch computations are compute bound. An interesting point here is that for the naïve implementation (97% compute time) of the bus computations with double-buffering shows poorer performance compared to single buffering. However, for the *vectorized unified-bus-computation* technique (75% of compute bound) double buffering (multicore-efficient) performs better than single buffering by about 20%. Here double buffering enables overlap of DMA transfers with the vectorized operations in the SPE dual pipeline.

TABLE 5.31: Distributed speedup and % of computation on single SPE; Note our multicore-efficient implementation combines both double-buffering scheme and *vectorized unified-bus-computation* module.

| Optimization | Performance | Bus | Branch | Bus + Branch |
|---|---|---|---|---|
| The naïve serial implementation | Speedup | *1* | *1* | *1* |
| | % Computation | *97 %* | *77 %* | *87 %* |
| Double-buffering scheme | Speedup | *0.96* | *1.13* | *1.15* |
| | % Computation | *99 %* | *95 %* | *96 %* |
| Vectorized unified-bus-computation | Speedup | *7.74* | *4.80* | *7.7* |
| | % Computation | *75 %* | *62 %* | *72 %* |
| Multicore-efficient | Speedup | *9.25* | *5.31* | *9.2* |
| | % Computation | *90 %* | *79 %* | *87 %* |

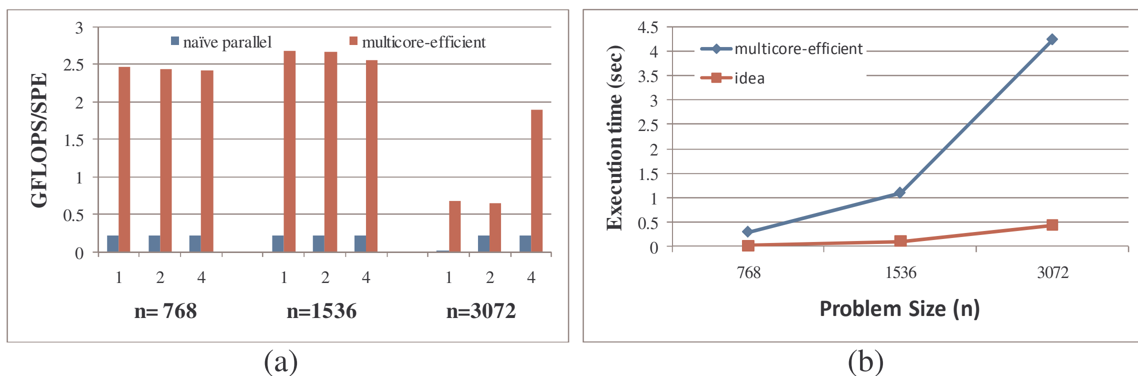



FIGURE 5.29: Overall performance on IBM Cell/B.E. platform: (a) Performance in GFLOPS per SPE; (b) Execution time in seconds on a single SPE.

In Figure 5.29, we show the overall performance in GFLOPS per core for problem-scaling and core-scaling for both naïve parallel and our multicore-efficient implementation on the IBM Cell/B.E. platform.

In Figure 5.29(a), the naïve parallel implementation shows a performance of *0.25* GFLOPS/SPE with the problem size (*n=768* and *n=1536*). However, for a larger problem size (*n=3072*), the performance on single SPE reduces to *0.05* GFLOPS. Our multicore-efficient implementation shows performs well with problem scaling and achieves a performance of *2.7* GFLOPS/SPE with the problem size *n=1536*. For both the naïve parallel and multicore-efficient implementations, we observe an almost linear scaling of performance with respect to the core-scaling for the problem sizes with *n=768* and *n=1536*. For *n=3072*, the total required memory space exceeds the available size of main memory of out IBM Cell/B.E. platform. Thus, the high disk access latencies incurred with *n=3072* reduces the performance of our implementations.

In Figure 5.29(b), we compare the efficiency of the multicore-efficient implementation with that the ideal computing performance on the IBM Cell/B.E. platform. The ideal time for PFS-GS is the time required for only computations with the theoretical peak core performance on single core of the IBM Cell/B.E. platform.

## 5.5 Conclusion

In this chapter, we have presented experimental studies based on our effective data parallel design methodology for two commercial available mutlicore platforms, Intel Clovertown and IBM Cell/B.E. platform. Based on a weighted-vertex pebbling strategy, data-aware scheduling, data prefetching and caching strategies (see Chapter 3 and 4), we discuss multicore-efficient implementations of four algorithms, matrix multiplication,

FDTD, LU decomposition and power flow solver based on GS method. Note that the theoretical case studies for matrix multiplication and FDTD algorithms are illustrated in Chapter 3 and 4. For multicore efficient implementation, we present the in-core optimization techniques of data transformation, loop transformations and vectorization in Chapter 5. From theoretical bounds, we determine the size of each the block at each level of the memory hierarchy, parallel scheduling strategy, and data buffering schemes considering both the architecture and algorithm. At the register level, we illustrate the computational ordering based on weighted-vertex pebbling strategy to achieve efficient vectorized implementations. Also, the effects of data layout on performance are investigated.

Our multicore efficient implementations seek to aggressively exploit data locality to achieve good performance. For Intel Clovertown platform, our measurement results of multicore-efficient implementations indicate a speed-up per core of *31*x, *1.8*x, *2.8*x and *2.4*x for matrix multiplication (*n=4096*), the FDTD algorithm (*n=256*), the LU decomposition (*n=2048*) and the PFS-GS algorithm (*n=3072*), respectively, compared to compiler optimized naïve parallel implementations. For IBM Cell/B.E. platform, our measurement results of multicore-efficient implementations indicate a speed-up per SPE of *16*x, *4.3*x, *4.7*x and *10.6*x for matrix multiplication (*n=2048*), the FDTD algorithm (*n=128*), the LU decomposition (*n=2048)* and the PFS-GS algorithm (*n=1536*), respectively, compared to compiler optimized naïve parallel implementations. We observe good performance scalability both with the number cores (core-scaling) and the problems size (problem-scaling) for both platforms. We also note that in algorithms

where data locality is limited, efficient vectorization can result in an overall improvement in performance.

# CHAPTER 6: CONCLUSION AND FUTURE WORK

## 6.1 Conclusion

Multicore architectures attempt to achieve power efficient performance by exploiting data locality, and data and task level parallelism in an application through multiple processing cores and deep memory hierarchies integrated on a single chip. Often, this performance is only realized by designing code that effectively maps the application to the underlying architecture. A variety of multicore architectures exists in the market today subscribing to different philosophies regarding the processing complexity of the core, hardware control of the memory hierarchy and the nature of the on-chip interconnect.

In this dissertation, we argue that robust portable multicore software is best designed by focusing on designing algorithms that are parallel, cache friendly, and are capable of exploiting compute-transfer parallelism. Further, optimization techniques that are well established across a wide variety of architectures and programming platforms need to be integrated into this design process to obtain the highest performance. In this regards, we have presented an efficient software design process that combines algorithm analysis with practical optimization techniques for data parallel algorithms. The resulting code shows high performance on diverse commercial multicore platforms, and scales well both with problem size and the number of cores. Among the algorithm analysis techniques are a) weighted vertex pebbling game for designing space- and cache-

efficient algorithms targeting shared memory hierarchies b) parallel scheduling for computations at different levels of the memory hierarchy and c) integrated data prefetching and caching schemes for overlapping data transfer and computation. Among the optimization techniques are a) efficient cache-friendly data transformation, b) loop transformations and c) vectorization.

We developed multicore-efficient software developments based on weighted-vertex pebbling strategy, integrated data prefetching and caching and in-core optimization techniques for commercial available multicore platforms.

We present detailed case studies that highlight the approach listed above for different data parallel kernels. In general our multicore efficient implementations outperform naïve parallel implementation. In particular our multicore efficient implementation scales well both with respect to problem size and the number of cores. The multicore efficient matrix multiplication algorithm outperforms the Intel MKL matrix multiplication library without the use of assembly code. For LU decomposition, there are trade-offs between larger block sizes and load balancing and less parallel partitioning. Large block size improve data locality but may result in reduced parallel data partitions which resulting in poor load balance with core-scaling. For FDTD, we observe trade-offs between spatial and temporal locality. Layouts such as Z-Morton that promotes spatial locality within a block adversely affects spatial locality between blocks. For PFS-GS, we note that although locality of data is limited exploiting data parallelism through efficient vectorization can improve the overall performance. Also, in all the above kernels we note that the choice of the scheduling scheme at the register level is critical in promoting efficient vectorization.

## 6.2 Future work

Our design process identifies algorithm (for example block size) and implementation parameters (for example loop unrolling depth) that can be tuned to improve performance on a given multicore platform. For our multicore efficient implementations on the Intel Clovertown and IBM Cell BE platforms, these parameters were tuned manually to obtain the highest performance. An extension of our work is to incorporate an autotuning framework that can automatically identify the best combination of setting for these parameters that can result in the highest performance on the target multicore computing platform. Note that since there are a large number of such parameters with a large range of values combined with the execution time for each iteration, an exhaustive search of the design space is computationally prohibitive. Recently auto-tuning frameworks have been proposed for stencil based algorithms on multicore platforms. Our design process can help identify the best set of tuning parameters and their nominal values such that the design space exploration time is minimized.

On the theoretical side, we have considered only static blocking at each level of the memory hierarchy. In algorithms where there exists a trade-off between block size and the number of blocks that can be processed in parallel (for example LU decomposition), dynamically adjusting the block size may result in a better load balance with good data locality. Such dynamic adjustment of block size may also be important in virtualized environments where more than one operating system shares the processor. In this dissertation our focus was on modeling the temporal data locality. In the future we seek to model spatial data locality as well.

Experimentally, we have verified the applicability of the proposed design flow in developing high performance kernels on the Intel Clovertown and the IBM Cell BE platforms. In future projects, we will extend the applicability of the design process to other commercial multicore platforms such as the Sun UltraSPARC T2 and GPUs (for example Nvidia Tesla). Also, we have applied the integrated caching and prefetching scheme only to the IBM Cell BE platform. Other multicores such as the Intel Clovertown and AMD Barcelona have both hardware and software support for prefetching. However, the cache is hardware controlled and it remains to be seen if the proposed integrated prefetching and caching schemes will work well on these processors.

# REFERENCES


[1] S. Adams, J. Payne, and R. Boppana, "Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors", In Proc. DoD High Performance Computing Modernization Program Users Group conference, Pittsburgh, PA, June 2007.

[2] S. Albers and M. Buttner, "Integrated Prefetching and Caching in Single and Parallel Disk Systems", Proc. 15th Ann. ACM Symp. Parallel Algorithms and Architectures, June 2003.

[3] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Rudy, F. Thomas, and C. Almond, "Programming the Cell Broadband Engine Architecture Examples and Best Practices", IBM Redbooks, IBM International Support Organization, 2008.

[4] [Arge02] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro, "Cache-Oblivious Priority Queues and Graph Algorithm Applications", In Proceedings of the 34th Annual ACM Symposium on Theory of Computing, p. 268-276, 2002.

[5] D. Bader, V. Kanade, and K. Madduri, "SWARM: A Parallel Programming Framework for Multicore Processors", In: First Workshop on Multithreaded Architectures and Applications (MTAAP) at IEEE IPDPS 2007.

[6] M. Bader and C. Zenger, "Cache oblivious matrix multiplication using an element ordering based on the Peano curve", Linear Algebra and its Applications, Elsevier, 2006.

[7] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems", In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (Big Sky, Montana, USA, October 11 - 14, 2009). SOSP '09. ACM, New York, NY, 29-44. DOI= http://doi.acm.org/10.1145/1629575.1629579.

[8] M. A. Bender, E. D. Demaine, M. Farach-Colton, "Cache-Oblivious B-trees", In Proceedings of the 41st Annual Symposium on Foundations of Computer Science, p.399-409, 2000.

[9] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton, "Scanning and Traversing: Maintaining Data for Traversals in a Memory Hierarchy", In Proceedings of the 10th Annual European Symposium on Algorithms, vol. 2461, Lecture Notes in Computer Science, p. 139-151, 2002.

[10] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably Efficient Scheduling for Languages with Fine-Grained Parallelism", Journal of the ACM, Vol. 46, No.2, pp 281-321, March 1999.

[11] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithm", In Proceedings of the Nineteenth Annual ACM-SLAM Symposium on Discrete Algorithms, Jan. 2006.

[12] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low Depth Cache-Oblivious Algorithms", Carnegie Mellon University Computer Science Technical Report, CMU-CS-09-134, 2009.

[13] R. D. Blumofe, and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations", In the Proceedings of the 25th Annual ACM Symposium on Theory of Computing, p. 362-371, 1993.

[14] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-Consistent Distributed Shared Memory". In Proceedings of the 10th International Parallel Processing Symposium, pages 132– 141, Honolulu, Hawaii, Apr. 1996.

[15] D. Bonachea, "GASNet specification v1.1", U.C Berkeley Tech Report. UCB/CSD-0201207, U.C. Berkeley, 2002.

[16] J. Byun, A. Ravindran, A. Mukherjee, B. Joshi, and D. Chassin, "Accelerating the Gauss-Seidel power flow solver on a high performance reconfigurable computer", In Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), p.227-230, 2009.

[17] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies", In Proceedings of the ACM International Conference on Measurement and Modeling of Computer System (SIGMETRICS), p. 188-196, 1995.

[18] D. Chassin, P. Armstrong, D. Chavarria-Miranda, R. Guttromson, "Gauss-Seidel Accelerated: Implementing flow solvers on field programmable gate arrays", Power Engineering Society General Meeting, IEEE, 2006.

[19] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems", In Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99), 1999.

[20] T. Chen, Z. Sura, K. O'Brien, and K. O'Brien, "Optimizing the Use of Static Buffers for DMA on a CELL Chip", Workshop on Language and Compiler for Parallel Computing (LCPC), 2006.

[21] S. Chen, 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", Symposium on Parallel Algorithms and Architectures, 2007.

[22] R. A. Chowdhury, and V. Ramachandran, “Cache-Oblivious Dynamic Programming”, Symposium on Discrete Algorithms, p. 591-600, 2006.

[23] R. A. Chowdhury, and V. Ramachandran, “Cache-Efficient Dynamic Programming Algorithms for Multicores”, Proceedings of the 20th ACM Symposium on Parallel Algorithm and Architectures, p. 207-216, 2008.

[24] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, and T. von Eicken, “Log-P: Towards a Realistic Model of Parallel Computation”, Proc. of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp 1-12, 1993.

[25] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, “Stencil computation optimization and auto-tuning on state-of-the-art multicore Architectures”, In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, Texas, November 15 - 21, 2008.

[26] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick, “Optimization and performance modeling of stencil computations on modern microprocessors”, LBNL-63192, 2009.

[27] E. D. Demaine, “Cache-Oblivious Algorithms and Data Structures”, In Lecture Notes from the EEF Summer School on Massive Data Sets, LNCS, Springer, 2002.

[28] H. Dursun, K.Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta, “In-Core Optimization of High-order Stencil Computations”, In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), p. 533-538, 2009.

[29] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-Oblivious Algorithms”, In Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS), p. 285-297, 1999.

[30] M. Frigo, “Portable High-Performance Programs”, PhD Dissertation, MIT, 1999.

[31] M. Frigo, and V. Strumpen, “Cache Oblivious Stencil Computations”, Proceedings of the 19th ACM International Conference on Supercomputing, 2005.

[32] M. Frigo and V. Strumpen, “The cache complexity of multithreaded cache oblivious algorithms”, Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures, 2006.

[33] K. Goto, and van de Geijn, “Anatomy of a high-performance matrix multiplication”, ACM Transactions on Mathematical Software, 2008.

[34] A. Grama, A. Gupta, G. Karypis, and V. Kumar, “An Introduction to Parallel Computing: Design and Analysis of Algorithms”, 2nd ed., Addison-Wesley, Reading, MA, 2003.

[35] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture", IEEE Micro, p. 10-24, 2006.

[36] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency", in proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04), IEEE Computer Society, 2004

[37] T. Haung and C. Yang, "Further Results for Improving Loop interchange in Non-adjacent and Imperfectly Nested Loops", High-Level parallel programming models and supportive environments, 1998.

[38] A. Heinecke, and M. Bader, "Parallel matrix multiplication based on space-filling curves on shared memory platforms", In Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem? (Ischia, Italy, May 05 - 07, 2008). MAW '08. ACM, New York, NY, 385-392. DOI= http://doi.acm.org/10.1145/1366219.1366223.

[39] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 2nd edition, 1996.

[40] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game", In Proceedings of 13th Annual ACM Symposium on Theory of Computing, p. 326-333, 1981.

[41] E. Im, "Optimizing the Performance of Sparse Matrix-Vector Multiplication", Ph.D. Dissertation, U.C. Berkeley, 2000.

[42] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor", IBM Journal of Research and Development, p. 589-604, 2005

[43] M. Kallahalla, and P. J. Varman, "Optimal prefetching and caching for parallel I/O systems", In Proceedings of the 9th Annual European Symposium on Algorithms (ESA), LNCS, v. 2161, Springer-Verlag, p. 62-73, 2001.

[44] T. Kimbrel, and A. R. Karlin, "Near-optimal parallel prefetching and caching", SIAM Journal on Computing 29, p. 1051-1082, 200.

[45] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor Communication Network: Built for Speed", IEEE Micro, v.26, p.10-23, 2006

[46] D. Krolak, "Just Like Being There: Papers from the Fall Processor Forum 2005: Unleashing the Cell Broadband Engine Processor-The Element Interconnect Bus", http://www.ibm.com/developerworks/power/library/pa-fpfeib/, 2005.

[47] J. Kurzak, W. Alvaro, and J. Dongarra, “Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor”, Parallel Computing 35 (2009) 138-150, 2009.

[48] M. S. Lam, E. E. Rothberg, and M. E. Wolf, “The cache performance and optimizations of blocked algorithms”, Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, p.63-74, 1991.

[49] W. Liu, J. Tuck, C. Wonsun, A. Karin, S. J. Renau, J. Torrellas, “POSH: A TLS Compiler that Exploits Program Structure”, Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, 2006.

[50] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, “Models of Parallel Computation: A Survey and Synthesis”, Proceedings of the Twenty-Eight Hawaii International Conference on System Sciences, p. 61-70, 1995.

[51] R. K. Malladi, “Using Intel VTune Performance Analyzer Events/Ratios & Optimizing Applications”, Intel White Paper, 2004.

[52] K. S. Mckinley, S. Carr, and C. Tseng, “Improving Data Locality with Loop Transformations”, ACM Transactions on Programming Languages and Systems TOPLAS, v. 18, p. 424-453, July 1996.

[53] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," pact, p.261-270, 2009 18th International Conference on Parallel Architectures and Compilation Techniques, 2009.

[54] A. Munshi, "OpenCL," http://s08.idav.ucdavis.edu/munshi-opencl.pdf, 2008

[55] G. J. Narlikar, “Scheduling threads for low space requirement and good locality”, Theory of Computing Systems, 35(2), Springer, 2002.

[56] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, “Augmenting Loop Tiling with Data Alignment for Improved Cache Performance”, IEEE Transactions on computers, vol. 48, 1999.

[57] M. Rafique, A. Butt, and D. Nikolopoulos, “DMA-based prefetching for I/O-intensive workloads on the Cell architecture”, In Proceedings of the 2008 conference on Computing frontiers, p. 23-32, 2008.

[58] G. Rivera, and C. Tseng, “Tilling Optimizations for 3D Scientific Computations”, In Proceedings of the ACM/IEEE Conference on Supercomputing, 2000.

[59] J. C. Sancho and D. J. Kerbyson, “Analysis of double buffering on two different multicore architectures: Quad-core opteron and the Cell-BE”, IEEE/ACM Int. Parallel and Distributed Processing Symposium (IPDPS), p.1-12, 2008.

[60] J. E. Savage, "Extending the Hong-Kung Model to Memory Hierarchies", In D.-Z. Du and M. Li, editors, Computing and Combinatorics, volume 959 of Lecture Notes in Computer Science, p.270-281, Springer Verlag, 1995.

[61] J. E. Savage, "Models of Computation: Exploring the Power of Computing", Addison-Wesley Longman Publishing Co., 1997.

[62] J. E. Savage and M. Zubair, "A Unified Model for Multicore Architecture", In Proceedings of 1st International Forum on Next-Generations Multicore/Manycore Technologies (IFMT), Nov. 2008.

[63] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Saha, D. Sheahan, L. Spracklen, and A. Wynn, "Ultrasparc t2: A highly-threaded, power-efficient, sparc soc", IEEE Asian Solid-State Circuirts Conference, Nov. 2007.

[64] M. F. Su, I. El-Dady, D. A. Bader, and S. Lin, "A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization", In proceedings of the 2004 International Conference on Parallel Processing (ICPP'04), 2004.

[65] J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly, "Improving the performance of Morton layout by array alignment and loop unrolling: reducing the price of naivety", Proceedings of 16th International Workshop on Languages and Compilers for Parallel Computing, v. 2958, p. 241-257, Springer-Verlag, 2003.

[66] A. Tiskin, "The bulk-synchronous parallel random access machine", Theoretical Computer Science, 196, 1-2, p. 109-130, Elsevier, 1998.

[67] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a Holistic Approach to Auto-Parallelization: integrating profile-driven parallelism detection and machine-learning based mapping", In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (Dublin, Ireland, June 15 - 21, 2009). PLDI '09. ACM, New York, NY, 177-187. DOI= http://doi.acm.org/10.1145/1542476.1542496.

[68] L.G. Valiant, "A Bridging Model for Parallel Computation", Communications of the ACM, 33(8), p. 103-111, August 1990. DOI= http://doi.acm.org/10.1145/79173.79181.

[69] L. G. Valiant, "A Bridging Model for Multicore Computing", In Proc. 16th European Symposium on Algorithms, p.13-28, 2008.

[70] P. van Emde Boas, "Preserving Order in a Forest in Less Than Logarithmic Time and Space", Information Processing Letters, 6(3), p. 80-82, 1977.

[71] V. Varadarajan and R. Mittra, "Finite-Difference Time-Domain (FDTD) Analysis Using Distributed Computing", IEEE Microwave and Guided Wave Letters, Vol. 4, May 1994.

[72] J. S. Vitter, and E.A.M. Shriver, "Algorithms for parallel memory I: Two level memories", Algorithmica, v. 12, n. 2-3, p. 110-147, 1994.

[73] J. S. Vitter, and E.A.M. Shriver, "Algorithms for parallel memory II: Hierarchical multilevel memories", Algorithmica, v. 12, n. 2-3, p. 148-169, 1994.

[74] D. Wentzlaff, and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores", *SIGOPS Oper. Syst. Rev.* 43, 2 (Apr. 2009), 76-85. DOI= http://doi.acm.org/10.1145/1531793.1531805.

[75] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "A Unified Operating System for Clouds and Manycore:fos", Computer Science and Artificial Intelligence Laboratory Technical Report, MIT-CSAIL-TR-2009-059, 2009.

[76] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms", Proceedings of the 2007 ACM/IEE conference on Supercomputing, 2007.

[77] M. E. Wolf and M. Lam, "A data locality optimizing algorithm", In Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada, June 1991.

[78] M. Xu and P. Thulasiraman, "Parallel Algorithm Design and Performance Evaluation of FDTD on 3 Different Architectures: Cluster, Homogeneous Multicore and Cell/B.E.", Proceedings of 10th IEEE International Conference on High Performance Computing and Communications (HPCC), p. 174-181, 2008.

[79] K. S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media", IEEE Transaction on Antennas and Propagation, v. AP-14, p. 203-307, May 1966.

[80] L. Youseff, and R. Wolski, "Vshmem: Shared-Memory OS-Support for Multicore-based HPC systems", Technical report, UC-Santa Barbara, 2009, http://www.cs.ucsb.edu/research/tech_reports/reports/2009-15.pdf.

[81] M. Vouk, "Cloud computing Issues, research and implementations", In 30th International Conference on Information Technology Interfaces (ITI 2008), p. 31-40, 2008.

[82] First the tick, now the tock: Next generation Inter microarchitecture (Nehalem), Intel White paper, 2008.