THÈSE

Présentée à

L'UNIVERSITÉ DE LILLE 1 SCIENCES ET TECHNOLOGIES

Pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Fan YE

NOUVEAUX ALGORITHMES NUMÉRIQUES POUR L'UTILISATION EFFICACE DES ARCHITECTURES MULTI-CŒURS ET HÉTÉROGÈNES

Thèse soutenue le 16 Décembre 2015, devant la Commission d'Examen :

Rapporteurs :	Michael A. Heroux	Directeur de Recherche Sandia National Laboratories, U.S.A.
	Michel Daydé	Professeur à l'ENSEEIHT
Directeur :	Serge Petiton	Professeur à l'Université de Lille 1 Sciences et Technologies
Co-directeur :	Christophe Calvin	Docteur en Mathématiques Appliquées HDR en Informatique, CEA Saclay
Examinateurs :	Jean-Luc Lamotte	Professeur à l'Université Pierre et Marie Curie
	Philippe Thierry	Docteur en Géophysique Intel Corporation France
Invité :	Guillaume Colin de Verdière	Expert International CEA Bruyère-le-Châtel

Acknowledgements

First of all, my sincere appreciation goes to my two advisors, Professor Serge G. Petiton and Doctor Christophe Calvin. Professor Petiton has rich experience and he has enlightened me in many ways. I am grateful to him for his support, his attention to detail, and many insightful discussions. Doctor Calvin's enthusiasm and agility have been a real inspiration to me. I still remember the day when I first stepped into his office, and tried to answer his questions. I am lucky to have made a good first impression on him. We work closely during the entire journey of my graduate study. His pragmatic approach to problem solving has been a great help for me. My heartfelt thanks also go to Doctor Philippe Thierry, a senior engineer at Intel. He has provided me with the prototyping cards for Intel's manycore coprocessor. He also spent many hours with me fixing the delicate machines, discussing technical issues, and envisioning the future of HPC. Conversations with him gave me insights into how to understand a manycore processor at the architectural level.

The opportunity and experience to do graduate research at the French Alternative Energies and Atomic Energy Commission (CEA) has been a real pleasure. CEA is not only providing me with a financial support, but also with a comfortable research environment, high-quality logistics service, and an inclusive culture. In here, I have met many colleagues who made my research life even more exciting. I would first like to thank my colleagues from my lab, the Maison de la Simulation. As the administrator of the working group "GRACE", I would like to express my appreciation to all group members, especially France Boillod-Cerneux, Florian Dang, Alexandre Fender, Sébastien Cayrols, and others. I would also like to thank the lab staff members, Pierre Kestener, Julien Derouillat, and others for their help when I ran into problems with my machines. My fellow graduate students at the CEA have been a fantastic help during my research. Particularly, I would like to thank my Chinese compatriots Langshi Chen, Cihui Pan, Yunsong Wang, Feng Xing, Zifan Liu and many others. With them I never feel homesick. I was teaching assistant at the University of Paris-Sud, the École Centrale de Paris, the University of Versailles. The instructos and colleagues from these places also deserve thanks, in particular Jean-Claude Martin (in the IUT of the University of Paris-Sud), Nahid Emad (professor at University of Versailles).

Last but not least, I would like to extend my deepest gratitude to my family, in particular my father and my mother. I am also grateful to my girlfriend. They have been and continue a great support to me for all these years. I can never express how grateful I am for their selfless love and unwavering support. I will always be indebted to my family. There are many names I do not have space to put here, but they will always have a place in my heart. This work could not have been performed without the their support.

Abstract

High performance computing (HPC) plays a fundamental role in tackling the most intractable problems across a wide range of disciplines and pushing the frontiers of science. It now becomes a third way to explore the unknown world besides theory and experiments.

Many-core architecture and heterogeneous system is the main trend for future supercomputers. The programming methods, computing kernels, and parallel algorithms need to be thoroughly investigated on the basis of such parallel infrastructure.

This study is driven by the real computational needs coming from scientific and industrial applications. For example in different fields of reactor physics such as neutronics or thermo-hydraulics, the eigenvalue problem and resolution of linear system are the key challenges that consume substantial computing resources. In this context, our objective is to design and improve the parallel computing techniques, including proposing efficient linear algebraic kernels and parallel numerical methods.

In a shared-memory environment such as the Intel Many Integrated Core (MIC) system, the parallelization of an algorithm is achieved in terms of fine-grained task parallelism and data parallelism. Both dimensions are essential to obtaining good performance in many-core architecture. For scheduling the tasks, two main policies, the work-sharing and work-stealing was studied. For the purpose of generality and reusability, we use common parallel programming interfaces, such as OpenMP, Cilk/Cilk+, and TBB to explore the issue of multithreading. For vectorizing the task, the available tools include Cilk+ array notation, SIMD pragmas, and intrinsic functions. We evaluated these techniques and propose an efficient dense matrix-vector multiplication kernel. In order to tackle a more complicated situation, we propose to use hybrid MPI/OpenMP model for implementing sparse matrix-vector multiplication. We also designed a performance model for characterizing performance issues on MIC and guiding the optimization. These computing kernels represent the most time consuming part in a classical eigen-solver. As for solving the linear system, we derived a scalable parallel solver from the Monte Carlo method. Such method exhibits inherently abundant parallelism, which is a good fit for many-core architecture. To address some of the fundamental bottlenecks of this solver, we propose a task-based execution model that completely fixes the problems.

Table of contents

List of figures **xi**

List of tables **xiii**

Nomenclature **xv**

1 Introduction **1**
1.1 Motivations . . . 2
1.1.1 HPC in Nuclear Engineering . . . 2
1.1.2 Hardware Evolution . . . 3
1.2 Outline of the Study . . . 4
1.2.1 Scientific Backgrounds . . . 4
1.2.2 Problem Definitions . . . 5
1.2.3 Related Work . . . 6
1.2.4 Main Contribution of Thesis . . . 11
1.2.5 Organization of Dissertation . . . 12

2 State-of-the-art in Parallel Computing **15**
2.1 Task-Centric vs Data-Centric . . . 15
2.1.1 Task Parallelism . . . 15
2.1.2 Granularity . . . 17
2.1.3 Data Parallelism . . . 18
2.2 Runtime . . . 20
2.2.1 CPU vs GPU . . . 20
2.2.2 CPU task scheduling . . . 21
2.3 Shared Memory Parallel Programming APIs . . . 22
2.3.1 OpenMP . . . 22
2.3.2 Cilk/Cilk Plus . . . 22
2.3.3 TBB . . . 23

2.3.4 QUARK . . . 23
2.4 Performance Metrics . . . 24

3 Architectural Considerations for Parallel Program Design 25
3.1 Abstraction of Computer Architecture . . . 25
3.2 Memory Hierarchies . . . 26
3.2.1 Memory . . . 27
3.2.2 Cache . . . 30
3.2.3 Register . . . 31
3.3 Memory Access In Parallel . . . 32
3.3.1 Message Passing Interface . . . 32
3.3.2 UMA vs NUMA . . . 32
3.4 Heterogeneous Architecture . . . 33
3.4.1 Many Integrated Cores Architecture . . . 33
3.4.2 Experimental Platform . . . 39
3.5 High Performance Programming Tactics . . . 40

4 Efficient Linear Algebraic Operations to Accelerate Iterative Methods 43
4.1 Numerical Context for Dense Matrix-Vector Multiplication . . . 43
4.2 Dense Matrix-Vector Product Kernel . . . 45
4.2.1 Pure Multithreaded Solution . . . 45
4.2.2 Combined Virtue of Multithreading and Vectorization . . . 49
4.3 Numerical Context for Sparse Matrix-Vector Multiplication . . . 54
4.4 Sparse Matrix-Vector Multiplication . . . 55
4.4.1 Vectorized Kernel . . . 58
4.4.2 Hierarchical Exploitation of Hardware Resources . . . 59
4.4.3 Matrix Suite . . . 60
4.4.4 OpenMP and MKL Performances . . . 60
4.4.5 Hybrid MPI/OpenMP Performances . . . 61
4.4.6 Flat MPI Performances . . . 64
4.4.7 Cross-Platform SpMV Performances . . . 65
4.4.8 Performance Analysis . . . 65
4.4.9 Performance Modeling . . . 68
4.5 Towards next generation of MIC architecture . . . 73
4.6 Conclusion . . . 74

5 Algorithmic Improvement of Monte-Carlo Linear Solver 77
5.1 Numerical Context . . . 77
5.2 Mathematical Prerequisite . . . 78
5.3 Parallel Algorithm and Implementation . . . 81
5.3.1 Parallel Algorithm Using CSR Sparse Format . . . 81
5.3.2 Considerations for multi-core and many-core architecture . . . 86
5.3.3 Experiments on Convergence . . . 88
5.3.4 Maximizing Performance on multi-core/many-core Processors . . . 90
5.3.5 Scalability in Performance . . . 92
5.4 Performance Bottlenecks of the Conventional Random Walk . . . 94
5.4.1 Random Number Generation . . . 94
5.4.2 Selection of New State . . . 95
5.4.3 Lack of Vectorization . . . 95
5.4.4 Variability of Numerical Results . . . 96
5.5 New Task-Based Reformulation . . . 96
5.5.1 Implementation Details . . . 96
5.5.2 Proof of Equivalence . . . 101
5.5.3 Optimization . . . 101
5.5.4 Runtime Choice . . . 103
5.5.5 Performance Analysis . . . 103
5.6 Toward a Smart-Tuned Linear Solver . . . 107

6 Conclusion and Perspective 109
6.1 Synthesis . . . 110
6.2 Future Research . . . 113

Bibliography 117

List of figures

2.1 Flynn's taxonomy 19

3.1 Von Neumann architecture 25
3.2 SRAM cell 26
3.3 DRAM bank 28
3.4 DRAM bursting 28
3.5 Avoiding bank conflicts 29
3.6 Process memory layout 29
3.7 KNC core 35
3.8 Vector pipeline 36
3.9 KNC interconnect 37

4.1 Pure multithreaded dense matrix-vector product kernel executed in CPU using OpenMP, Cilk+, and TBB 47
4.2 Pure multithreaded dense matrix-vector product kernel executed in MIC using OpenMP 47
4.3 Pure multithreaded dense matrix-vector product kernel executed in MIC using Cilk+ 48
4.4 Pure multithreaded dense matrix-vector product kernel executed in MIC using TBB 48
4.5 Multithreaded and vectorized dense matrix-vector product kernel executed in various architectures 51
4.6 Explicitly restarted arnoldi method 55
4.7 Compressed sparse row (CSR) format 57
4.8 Performances of SpMV kernel using OpenMP 62
4.9 Performances of SpMV kernel using MKL 63
4.10 Gain of the hybrid MPI/OpenMP SpMV kernel over the OpenMP one . . . 64
4.11 Cross-platform performances of different SpMV kernels 66

4.12 Relationship between two indicators and performance 70
4.13 Comparison between the real and estimated local SpMV performances . . . 71
4.14 Comparison between the real and estimated global SpMV performances . . 72
4.15 The knights Landing Overview 73

5.1 Parallel Monte Carlo linear solver 87
5.2 Residual for different executions of parallel Monte Carlo solver 89
5.3 Strong and weak scaling of parallel Monte Carlo linear solver using up to 4 KNC coprocessors 93
5.4 Strong scaling of parallel Monte Carlo linear solver on lab cluster "Poincaré" using up to 90 computing nodes. 93
5.5 New task-based execution model for Monte Carlo linear solver 98

6.1 The task scheduling of the Cholesky-based matrix inversion with synchronizations 115
6.2 The task scheduling of the Cholesky-based matrix inversion without synchronizations 115

List of tables

2.1 Explanation for Flynn's taxonomy 19

3.1 KNC VPU pipelines 36
3.2 Commonly used experimental platforms in this study 40

4.1 Explicit Vectorization Methods 50
4.2 OpenMP loop scheduling policies 52
4.3 Sparse matrices for testing different SpMV kernels 60

5.1 Thread affinity types 91
5.2 Speedups of parallel Monte Carlo linear solver using hybrid MPI/OpenMP model 92
5.3 Sparse matrices for testing task-based execution model of Monte Carlo linear solver 104
5.4 Experimental results of both the original parallel implementation and task-based model of Monte Carlo linear solver 105

Nomenclature

Acronyms / Abbreviations

ALU Arithmetic Logic Unit

API Application Programming Interface

AVX Advanced Vector Extensions

BLAS Basic Linear Algebra Subprograms

CAF CoArray Fortran

CAQR Communication-Avoiding QR

CEA Commissariat à l'Énergie Atomique et aux Énergies Alternatives, i.e. the French Alternative Energies and Atomic Energy Commission

CFD Computational Fluid Dynamics

CG Conjugate Gradient

CMOS Complementary Metal-Oxide-Semiconductor

CPU Central Processing Unit, practically synonymous with a single **core** in the context of **mutlicore processors**

CSC Compressed Sparse Column

CUDA Compute Unified Device Architecture, a parallel computing platform and application programming interface model created by NVIDIA

DAG Directed Acyclic Graph

DDR Double Data Rate

DRAM Dynamic Random-Access Memory

ECC Error-Correcting Code

EMU Extended Math Unit

ERAM Explicitly Restarted Arnoldi Method

FLOPS FLoating-point Operations Per Second

FMA Fused Multiply-Add

GDDR Graphics Double Data Rate

GMRES Generalized Minimum RESidual

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

HPC High Performance Computing

ILP Instruction-Level Parallelism

IMCI Initial Many Core Instructions

ISA Instruction Set Architecture

KNC Knights Corner

KNL Knights Landing

LIFO Last In First Out

LRU Least Recently Used

MCDRAM Multi-Channel Dynamic Random Access Memory

MC Memory Controller

MD Molecular Dynamics

MERAM Multiple Explicitly Restarted Arnoldi Method

MIC Many Integrated Core

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MMU Memory Management Unit

MMX MultiMedia eXtensions

MOSFET Metal Oxide Semiconductor Field-Effect Transistor

MOS Metal Oxide Semiconductor

NUMA Non-Uniform Memory Access

PCIe PCI Express, a high-speed serial computer expansion bus standard designed to replace the older PCI, PCI-X and AGP bus standards.

PCI Peripheral Component Interconnect

PDE Partial Differential Equation

PGAS Partioned Global Address Space

QR It often refers to the QR factorization that decomposes a matrix into an orthogonal matrix Q and an upper triangular matrix R. It can also refer to the QR method.

RAM Random-Access Memory

RS Ring Stop

SCIF Symmetric Communication Interface

SDRAM Synchronous Dynamic Random-Access Memory

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SISD Single Instruction Single Data

SMP Symmetric Multi-Processors

SMT Simultaneous MultiThreading

SNB Sandy Bridge

SRAM Static Random-Access Memory

SSE Streaming SIMD Extensions

TD Tag Directory

TLB Translation Lookaside Buffer

TMP Template MetaProgramming

TSQR Tall Skinny QR

TTS Time-To-Solution

UMA Uniform Memory Access

UPC Unified Parallel C

VLIW Very Long Instruction Word

VPU Vector Processing Unit

Chapter 1

Introduction

High Performance Computing, or HPC, is the term for describing the practice of aggregating computing power and parallel processing techniques for solving complex computational problems. It originally pertains only to supercomputers for scientific research. Now it encompasses a collection of powerful hardware systems, software tools, programming languages, and parallel programming paradigms. It shares a number of common features with parallel computing, grid computing, cluster computing, distributed computing, or even cloud computing, though they certainly have different emphases.

The HPC is born within the academic community, which forms the basis of scientific computing, a.k.a. computational science. Numerical simulation, as a major component of computational science, is at the heart of numerous dramatic advances in fields as diverse as climate modeling, drug discovery, energy research, data analysis, and so on. In order to conduct high-accuracy quantitative study over complex phenomenon, large simulation needs to manipulate a great amount of data that is far beyond the capacity of any single computer. Such examples can be found in PDE (Partial Differential Equation) system in CFD (Computational Fluid Dynamics) or in MD (Molecular Dynamics) simulations. The answer to this predicament is parallel and distributed computing. It allows the data to be stored distributedly and the computers to work concurrently.

Different from the traditional forms of science and engineering composed of theory and lab experiment, scientific computing performs quantitative analysis based on domain specific models. The development of models as well as the HPC methodologies enables more complex analysis on artificial or natural systems. It begins to play a fundamental role in tackling the most intractable problems across a wide range of disciplines and pushing the frontiers of science.

1.1 Motivations

The research tools have been revolutionized since the last decades. Driven by the development of HPC, the numerical simulation becomes an indispensable instrument for scientific progress besides theory and experiments. The more HPC proves its effectiveness in diverse disciplines, the more reliant on HPC the scientific world is, and certainly being in demand of even more computing powers and memory capacities. This is almost the primary impetus behind the rapid development of HPC. Next, we will incorporate the background of this study into a more solid discussion on HPC in the subsection 1.1.1. Moreover, the hardware evolution also suggest the prosperity of HPC. We will include that part in the subsection 1.1.2.

1.1.1 HPC in Nuclear Engineering

This study is undertaken in and funded by the French Alternative Energies and Atomic Energy Commission (CEA). As the owner [1] and the operator of the largest public supercomputer (CURIE) in France, CEA is behaving proactively to embrace the forthcoming exascale computing era. According to the latest ranking of world's best supercomputers [125] released by June 2015, the Tianhe-2 supercomputer tops the list by touching a third of 100 petaFLOPS (33.8627 petaFLOPS, FLOPS is an acronym for floating-point operations per second). It is believed that the 100 petaflops system will make its debut very soon [22, 126, 129]. The exascale computing era is getting very closer. These recent achievements are built upon the architectures that are computationally more aggresive. The emergence of throughput-oriented accelerators makes up the deficiency of traditional latency-oriented processors in digesting large parallel workloads. They team up in a heterogeneous supercomputer to offer very promising performance.

The reactor physics has been evolving as a diverse and plural discipline with several important streams, such as thermodynamics, thermal hydraulics, neutronics, etc., which all contribute to the modeling of the nuclear reactor core. Different equations (Navier-Stokes, Boltzmann, Bateman, etc.) need to be solved, with the help of iterative solvers, so as to compute the state of the reactor core. These solvers, as well as the fine-grained modeling of the entire reactor core, or the rapid assessment of different solutions deriving from a vast parameter set, will require substantial computing power. The HPC technology is therefore pertinent in these applications, including the simulation, the optimization and the research of

[1] Along with other supercomputers (Jade at CINES, Titane at CCRT, Ada and Turing at IDRIS, OCCIGEN at CINES), the CURIE is owned by GENCI (Grand Équipement National de Calcul Intensif), which is a civil company held by French government, CEA, CNRS, and INRIA.

reactor physics. To further demonstrate this, I list here some actual needs for HPC arisen in the field of nuclear engineering.

parameterized computation As a basic optimization technique, this is a good occasion to practice HPC in a way that distinct parameters can be taken into account in order to reduce the time-to-market. Some optimization methods, such as neural network, are only practical upon the presence of HPC.

high-resolution physics A more realistic physics can be achieved by using systematically real physical model instead of simplified model or tabulated values. The refinement of models creates new demand for CPU power and memory capacity, which necessitates the use of HPC.

the real-time simulation This exists even before the birth of HPC. But HPC helps to build more realistic simulator which takes fewer approximations.

1.1.2 Hardware Evolution

The hardware evolution also stimulates the emergence and development of parallel computing, or HPC. It is shown in [62] that during 16 years since 1986 the performance of microprocessors enhanced on average 50% each year. However from 2002, this growth rate has dropped to 20%. The Moore's law says that the number of transistors in a dense integrated circuit has doubled approximately every 18 months [92, 20]. Higher transistor count means smaller transistor size. And smaller size allows to achieve faster clock speed. Let us exemplify it with MOSFETs (Metal Oxide Semiconductor Field-Effect Transistor). Generally, it is the threshold voltage, inter alia [2], that dominates the size of transistor. The threshold voltage is a function of the capacitance of the oxide layer in MOSFET. Lower capacitance leads to more quickly switching of transistor (0 to 1 or 1 to 0). Since the capacitance is directly related to the dimension of the oxide layer, smaller sized transistor may achieve higher frequency, which ensures a better overall speed of the integrated circuit.

However, the power consumption goes up much faster than the frequency. The power consumption is proportional to the frequency and the square of voltage with the latter being proportional itself to the frequency. So there is a cubic relationship between the frequency and the power consumption. Most of this power is dissipated as heat. And according to [60], the air-cooled integrated circuits are reaching their limits of dissipating heat, which is also known as the "power wall". This physical constraint prevents the frequency from scaling

[2] Other factors include channel (N/P) dimensions, dopant materials, MOS process, etc.

up. But the Moore's law still applies [3]. It means that we can keep increasing transistor density in the integrated circuits. Yet a growth in density but no further frequency increase, there seems to be only one path to exploit the additional transistors, through parallelism. Rather than longer-pipelined, shorter latency and more complex monolithic core, multiple relatively simple but complete cores on a single chip could be a wise even sole option given the imperative of continual computing power growth.

1.2 Outline of the Study

In this section, I will first introduce the scientific background of this dissertation, as well as the industrial needs that drives the study. Secondly, I will define the problems that need to be addressed in this dissertation, especially the challenges and opportunities for dense and sparse linear algebra on manycore systems. I will present our approaches for solving these problems, along with a recall of the related works from the published literature. By doing so, I expect to establish the major connections to previous work and help the reader to see the novelty and distinctive insights in this study. Furthermore, I will summarize in a separate subsection the contributions of this dissertation makes to the community so that they could be of direct help to those who work on the similar fields. Finally, I will provide the organization of this dissertation as the concluding subsection.

1.2.1 Scientific Backgrounds

As the recent advances in HPC computing architecture come with higher requirement for programming instruments to keep scaling the performance, we seek to put more effort on node-level parallel techniques and algorithmic design. Part of the purpose is to upgrade the exisiting voluminous application. A quick solution lies in profiling the execution of the program, detecting the hotspots, and responding accordingly to the performance issues. However, such practice is no more than a stress reaction that essentially just manually recompile a part of code. It provides no insights into the heart of problem and certainly pushes no further the scientific computing. Only placing hope in the hardware advancement will not bring us into a better place. Because the hardware has no magic power to comprehend the innate character of the application or algorithm. Therefore, the intellect should be invested in understanding the logic behind the hardware evolution as well as the interaction between the algorithms and computing systems. A conceptually well-designed algorithm usually

[3] At least before the quantum tunneling becomes the real limitation when the size of transistor is getting suffciently small.

has nice asymptotic complexity. But it is not necessarily equivalent to a good choice for implementation. Let us quote an example from the Monte Carlo simulation, in which the critical process is to perform a search for a randomly sampled key. The search is based on the precalculated values. It tries to identify the interval the key corresponds to, or simply the mapped value from many key-value pairs. At first glimpse, the binary search seems to be the ideal answer for the interval looking while the hash table for the key-value pairs. However, the best solutions proposed for the real cases are adapted or altered thoroughly [21, 80, 144]. The gain of performance comes from the incorporation of the hardware knowledge and the understanding of the application.

In this thesis, I am dedicated to develop and improve the parallel computing paradigms to respond to the actual computational needs from the nuclear engineering. Our work serves eventually the high-resolution physics and real-time simulation. But it does not target a specific application but rather to understand how to exploit effciently the new-coming architecture: manycore. The way I address this problem is to focus on the specific numerical kernels which are representative of final applications. Moreover, since the objective is to reuse the results of this work in the applications, I have to take into account some constraints, such as programming languages or data structures. As stated precedingly, there is no stop for the forthcoming exascale computing era. The scalability and efficient use of computing power will become two major problems to be addressed. In this work I mainly put my attention on the second point, hoping to be able to exploit efficiently the limited computing resources that are indicative of actual and future hardware architectures of supercomputers.

1.2.2 Problem Definitions

In neutronics, one key process in simulation is to determine the neutron flux in the reactor core. The neutron flux is expressed as the solution to the neutron transport equation, a.k.a the Boltzmann equation. Using the concept of virtual critical reactor associated to the real reactor in a determined moment, the temporal Boltzmann equation of neutron transport can be cast into the form of eigenvalue problem, where the eigenvalue provides a measure of whether the system is critical or subcritical and by how much [81, 86, 119]. The second case comes from the thermal hydraulics problem handled by 3D CFD code TrioU [25]. In progress of solving Navier-Stokes equation, a linear system will be generated at each time step, which is the most computationally costly problem to deal with. As a result, I mainly focus on the design and implementation of parallel numerical methods to solve linear systems and eigenvalue problems. Following the approach I discussed above, I start with the significant computing kernels such as the matrix-vector product, given their roles in these numerical algorithms. Then I explore the methods that potentially expose more parallelism. The key is

to find where lie the fundamental constraints of these methods. I restructure the algorithms so that they become aware of the underlying parallel architecture. It allows the algorithms to make a better use of the hardware. In the end, I merge the new methods with the commonly used ones in order to find a balance between the speed and numerical properties. With these core kernels implemented and replaced, the global application is prepared for a much better computing environment. The whole application behind this research should be kept consistent. So certain conditions need to be respected for the portability of performance. First of all, it is required to use a standard storage format for the sparse matrices. This consistency allows to decouple the performance from any particular architecture. It also facilitates the prototyping with other frameworks, most of which have implemented these formats. Secondly, it is advisable to use the broadly available programming languages. Again, this strategy gives us some distance from the low-level development which may involve excessive architectural details, frequent updates and maintenances. Such distance is also necessary for me to concentrate on the hardware and algorithms. Here the "hardware" refers specifically to manycore architectures, exemplified by Intel MIC (Many Integrated Cores) and GPGPU (General-Purpose computing on Graphics Processing Units). In this study, I will be focusing on a variety of parallel programming paradigms and studying how they best interact with the multicore and manycore systems especially in a heterogeneous environment.

1.2.3 Related Work

As explained above, I take two approaches in this research to design high performance linear algebra. The first approach considers the classical iterative methods for solving both the eigenvalue and linear system problems. For example, for solving the eigenvalue problem, there are power method [115] (see Algorithm 1), or ERAM [115] ("Explicitly Restarted Arnoldi Method") method, etc. For solving linear system, there are conjugate gradient method [114] (see Algorithm 12), or GMRES [114] ("Generalized Minimal Residual Method"), etc. These methods are time-tested. And they have good numerical properties in convergence, stability or robustness. The iterative methods repeat the same process until the solution converges. The same set of algebraic operations are executed periodically. To effectively shorten the execution time is to reduce the most time consuming part by parallelizing it. In the above mentioned classical iterative methods, the common challenge is the matrix-vector multiplication. So at the end of the day, the key of this approach is to figure out the efficiently parallelized matrix-vector multiplication for both dense and sparse linear algebra.

Matrix-vector multiplication is a memory bandwidth bound problem. The arithmetic intensity [141], namely the flop-to-byte ratio, is $\frac{1}{4}$ for double precision floating-point numbers.

A double precision number has 8 bytes. For each such number loaded, there are two floating-point operations to be performed: a multiplication plus an addition, which makes the arithmetic intensity equal to 2/8, or a quarter. Modern processors usually performs much better in terms of peak FLOPS performance than on-chip memory bandwidth. The aggregate node level bandwidth for Sandy Bridge [116] in fully subscribed mode is 60.8 GB/s [116] while the corresponding peak FLOPS performance is 332.8 GFlop/s [116].

Compared to the sparse linear algebra, the dense case is more friendly to memory. As detailed in the section 3.2, the memory content is never delivered in bits, bytes or words for efficiency reasons. Every memory load packs the whole burst section in the address space (see section 3.2) and send them all to the processor via cache. In a dense matrix-vector multiplication, both the matrix and vector elements are continuous in memory. When a matrix element is loaded for computation, its adjacent elements are also loaded along with it, and they will take part in the computation shortly, if not considering the vectorization. In other words, the memory bandwidth is never wasted. Every bit of the memory request will be put into use. However, in a manycore system things are more complicated. Traditional processors are designed to be latency-oriented device. They need to handle the mutli-tasking at the operating system (OS) level. They usually have large caches to convert long latency memory access to short latency cache accesses. They have been designed to be good at sophisticated control, including the support for branch prediction for reduced branch latency, data forwarding for reduced data latency, etc. And their single arithmetic logic unit occupies relatively large on-chip area to enable reduced operation latency. Whereas the manycore system aims purely at HPC tasks, which are often voluminous in terms of computation, but simple in control logic. So the manycore system is designed to be throughput-oriented. They would have small caches to boost memory throughput, simple control without support for branch prediction or data forwarding, long latency but many energy efficient ALUs for high throughput. A massive number of threads is required to tolerate latencies.

The mainstream manycore systems include GPGPU and MIC. For dense linear algebra, blocking (see section 3.5) is a commonly used optimization technique to make the memory accesses more compact and coalesced. In Nath et al.'s paper [93], the authors propose both the storage oblivious and storage aware algorithms to discuss different ways of organizing data in memory by taking advantage of matrix symmetry. Some techniques such as padding, pointer redirecting, or autotuning are also used in order to optimize the memory access. As the matrix-vector multiplication is a memory bandwidth bound problem, the memory usage efficiency is extremely important in its implementation. Memory contains not only the data for computation, but also the instructions that perform the computation. In Volkov et al.'s paper [134], the authors discuss the way to attain peak instruction throughput and

the best use of GPU memory. Other works [94, 83, 109, 128] express similar ideas in reorganizing the data in memory or adjusting the order of operations so that the instructions are executed in a high-throughput manner with few memory stalls. The GPU functioning mechanism reminds us of early vector machines like Cray-1 [113]. The operands are loaded and executed in terms of vectors. They both are data-parallel device. The ability of processing an array of data is the guarantee of high throughput. MIC, as also a mainstream manycore system, achieves high throughput by wide SIMD (single instruction multiple data, see subsection 2.1.3) engines as well. This is a similarity between the GPU and MIC. However, their difference is also remarkable. GPU has its own hardware scheduler. There is this scoreboard to keep track of a great number of SIMD instructions. If the operands are ready, the instruction gets executed by a warp of threads [60]. The current generation of MIC has 61 cores, which are all simple but complete x86 cores, each featuring a 512-bit vector processing unit. The programmer has to specify the scheduling in order for the instructions, including the SIMD extension of x86 instructions, to be properly distributed over these 61 cores. This demands the programming of MIC to consider one more issue than GPU, that is the scheduling of instructions. Some parallel programming languages, such as OpenMP [36], Cilk Plus [15, 51], or TBB [111] (see chapter 2), provides high-level runtime interface to accomplish the scheduling. With these three frameworks, I am able to express all the fine-grained task parallelism. So I will use them as the principal programming tools in this dissertation. Cramer et al. compares in [35] the OpenMP overheads in MIC with that in SMP machines. They also compare the MIC with Xeon processors in on-chip bandwidth and performance of real-world kernels. The results suggest that OpenMP is promising in MIC. It is also noted that the effort for porting scientific applications to CUDA or OpenCL can be much higher compared to directive-based programming models like OpenMP [138]. Early experiences on Intel Xeon Phi coprocessors revealed that porting scientific codes can be relatively straightforward [75, 59], which makes this architecture with its high compute capabilities very promising for many HPC applications. Many OpenMP applications prefer large shared memory systems. To make use of these NUMA machines, data and thread affinity has to be considered in order to obtain the best performance [127]. Taking into account these tuning advices, applications can scale to large core counts using OpenMP, like do TrajSearch [61] and Shemat-Suite [120]. Eisenlohr et al. [45] introduce the test of hardware utilization and runtime efficiency of OpenMP and Cilk Plus on MIC. The results show similar performance for OpenMP and Cilk Plus and demonstrate a minimal impact for application-level restructuring on performance. Vladimirov [131] sheds some insights on cache traffic optimization on MIC for parallel in-place sqaure matrix transposition with Cilk Plus and OpenMP. Reinders provides in his book [111] the in-depth overview of TBB. He

also comprehensively discusses TBB's task-based programming within concrete examples such as Fibonacci numbers. Saule et al. [117] present scalability results of a parallel graph coloring algorithm, several variations of a breadth-first search algorithm and a benchmark for irregular computations using OpenMP, Cilk Plus, and TBB on an early prototype MIC board. The results are positive showing the advantage of MIC for hiding latencies in irregular applications to achieve almost perfect speedup.

When we talk about the scheduling, there has to be a grainsize. The minimum grainsize is a single instruction. The way the instructions are grouped as a scheduling unit has a great impact on the performance. The fine-grained task often refers to a small piece of code. These fine-grained tasks have two relationships. The first one is when they are independent, or commutative, meaning they can be executed in whatever order. This type can be found in the OpenMP work-sharing model. Chapman et al. [31] explain the details of sharing work among threads. In this model, each chunk dispatched to some thread can be considered as a task. The second is when these tasks form a tree structure. That is to say in terms of data dependency, each task may have at most one parent. Imagine the tasks as the vertices, and the relations between tasks as the edges, then the ensemble of tasks form a tree. In these two cases, the scheduling overhead is the only cost to the parallelization. Blumofe et al. [16] study the problem of effciently scheduling fully strict multithreaded computations on parallel computers. The scheduling method they use is "work-stealing", which is also the basic scheduling mechanism of both Cilk Plus and TBB. In Duran et al.'s paper [44], the authors evaluate different scheduling strategies for OpenMP task runtime, including work-first scheduler, breadth-first scheduler, and cutting off scheduler. Although the results are obtained for OpenMP, the key ideas are applicable in Cilk Plus and TBB. The tasks can also have a graph structure, i.e. each task may have more than one parent. In this case, the ensemble of tasks form a directed acyclic graph (DAG). The task dependencies can only be inferred at runtime. In addition to the scheduling overhead, there is now a cost of inference. For this organization to be profitable, the tasks need to have larger grainsize. For this reason, I do not categorize this into fine-grained task. Kwok et al. [79] present static scheduling algorithms for allocating directed task graphs to multiprocessors.

When it comes to the sparse linear algebra, the memory operations are more challenging than the dense case. Since the matrix is sparse, there could be different storage formats to represent the matrix. But whatever format is used, the irregular and uncontinuous memory access is inevitable. For example, if the nonzero elements are compressed for a compact storage, then the access to the vector becomes sparse. Irregular and uncontinuous memory pattern means the waste of memory bandwidth, which undermines the performance of such memory bandwidth bound kernel. It has been shown that memory bandwidth bound kernels like sparse

matrix vector multiplication can achieve high performance on throughput-oriented processors like GPGPUs [12] (depending on the matrix storage format), but only little knowledge is present of what the performance is on Intel's manycore processor generation. Nevertheless, the sparse matrix-vector multiplication (SpMV) has been constantly investigated over decades on various architectures [76, 140, 3, 147, 23, 106, 13, 38]. Among these studies, blocking is always exploited as an important technique to optimize SpMV. In general, there are two different motivations for blocking. The first is to improve the spatial and temporal locality of SpMV kernel. In this case, data reuse is exploited at various memory levels, including register [89, 69], cache [69, 97], and TLB [97]. The second is to eliminate integer index overhead in order to reduce the bandwidth requirements [136, 106]. Besides blocking, matrix reordering [101], value and index compression [139, 76], and exploiting symmetry [23] have also been proposed. Improving the SpMV on GPUs has attracted lots of attention due to the increasing popularity of GPUs. Numerous matrix formats have been proposed, among which ELLPACK and its variant have been proven to be most successful. I refer to these survey papers [13, 91, 130, 77, 33, 90]. Similar to a SMP platform, the MIC system may benefit from hybrid MPI/OpenMP programming model when it shows negative NUMA effect. The prior work on hybrid MPI/OpenMP programs tends to conclude negatively for hybrid MPI/OpenMP model [30, 34]. However, the platforms used in those experiments are all clusters which are comprised of more than one node. The network bandwidth and the communication performance are identified as the main bottleneck of a such model.

Taking a different perspective, the second approach starts from the manycore hardware. Classical iterative methods may have good numerical properties, but they are not necessarily good candidates for manycore architecture. Even the computing kernels are well parallelized, the synchronization imposed by the transition of iterations could drag down the performance. So my idea is to design a numerical algorithm which understands more or less the functioning mechanism of the manycore system. Such algorithm is supposed to provide efficient parallelism that is easy to be implemented in manycore system. Demmel et al. [40, 6, 10] propose a new way to do the QR factorization for tall and skinny matrices. These matrices can be split into small pieces eligible for parallel factorization. Then the processed matrix blocks will update the coefficients in a divide-and-conquer manner. The conventional QR factorization processes the matrix from left to right. The dependency at each step limits its potential for parallelization. The TSQR (tall and skinny QR) and CAQR (communication-avoiding QR) expose the parallelism of QR factorization to the hardware, which makes it much easier to obtain good performance. Following the same logic, it is not hard to find Monte Carlo method as a good starting point for manycore system. Monte Carlo method was first used in linear algebra in the work of Forsythe et al. [49]. Srinivasan et al. [123] described how

Monte Carlo method can be used to perform stochastic matrix-vector multiplication and solve the linear system. Rosca [112] also discusses some important numerical properties of Monte Carlo linear solver. However, the Monte Carlo linear solver is rarely implemented in parallel machines. Jakovits et al. [72] demonstrates the parallel potential of Monte Carlo linear solver by implementing it with MapReduce.

1.2.4 Main Contribution of Thesis

This thesis considers the linear algebra problems within the context of manycore architecture. The manycore architecture is a recent progress in high performance computing domain. The manycore processors may unleash tremendous computing power in some circumstances. Their presence is the prerequisite for many other scientific or engineering advancement. However, there is no automatic performance gain. Parallel algorithms must be designed and implemented with proper programming interfaces based on the good understanding of the functioning mechanism of the underlying architecture. From a good algorithm to a good performance there is a long way to go. The first contribution of this thesis is the synthesis of four abstraction layers for an algorithm to efficiently run on a manycore system. The execution model of MIC is M-(SIMD/SISD), meaning "multiple SIMD/SISD (single instruction single data, see subsection 2.1.3)". The first "M" refers to the number of cores. A manycore machine such as MIC depends on threads to issue vector (SIMD) or scalar (SISD) instructions to cores. The instructions should be organized in terms of tasks, not threads (see section 2.2). High throughput can be achieved if the tasks are vectorized. So in terms of programming model, two dimensions should be taken into consideration: the data parallelism, and the fine-grained task parallelism. Therefore the 4 abstraction layers are,

- Algorithm
- (Vectorized) Tasks
- Threads
- Physical cores

Programming a shared-memory manycore system is to figure out the proper mappings between any two adjacent layers. For an algorithm to be easily translated into tasks, it is possible to restructure an exisiting algorithm, or to design a new parallel algorithm. For the (vectorized) tasks to be dispatched to threads in a balanced way, there are two strategies for fine-grained task scheduling: work-sharing, work-stealing. For the threads to be mapped to physical cores, the data locality and threads affinity should be considered. Once these

problems have their answers, the manycore machine is ready to contribute its computing power.

Porting an exisiting code to MIC does not has the same complexity as obtaining good performance in MIC. The second contribution of this thesis is providing instructions for optimizing code in MIC with different memory access pattern. In dense linear algebra, I obtained very promising performance using OpenMP and SIMD pragma. In sparse linear algebra, a hybrid MPI/OpenMP model can help improve the data locality and alleviate scheduling overhead. I also proposed a performance model for SpMV kernel for a better understanding of performance on MIC. This model can be instructive for other types of computing kernels.

The third contribution of this thesis is providing an task-based approach for implementing Monte Carlo method in a completely different way. Monte Carlo method is widely used in many domains. Repeated sampling is a key step in this method. No one ever imgained a Monte Carlo method without random sampling. I proposed a reform to the Monte Carlo method in order to bypass the random sampling. Although the technique is implemented only for the Monte Carlo linear solver. But it provides a reference for other Monte Carlo methods.

The last contribution of this thesis is having implemented in MIC the efficient numerical methods for solving eigenvalue problem and linear system. I proposed two approaches for implementing these methods. The first approach is to develop efficient computing kernels for classical iterative methods. The second approach is to design high parallel numerical algorithm for solving linear system. One of the ongoing work, as the continuation of this thesis, is further develop the Monte Carlo linear solver as a preconditioner for conjugate gradient method.

1.2.5 Organization of Dissertation

This is how the chapters of this dissertation are organized. As we take a bottom-up approach to consider the parallel solutions to concerned problems, most of the experiments are carried out in shared-memory intra-node level. The chapter 2 introduces the state-of-the-art in parallel computing that places a focus on the manner of a workload being resolved into parallelizable jobs, and how this is done by using shared-memory programming methods. The chapter 3 then makes an effort on a thorough but succinct discussion that takes side of hardware. The emphasis is put on the memory subsystem and heterogenenous architecture. The former is the key to performance. And the latter is the question being asked in this study that we hope to be able to answer to. Based on the understanding that we described in chapter 3, we propose in chapter 4 the parallel technique and programming model that would achieve better performance than high-standard manufacturer-provided libraries. The solution

we proposed not only leads to better implementation for widely used computing kernels, but also makes focal advice for similar applications. The chapter 5 takes a step further by exploring algorithmic improvement. We start from the method that inherently exposes abundant parallelism (Monte Carlo method), which is expected to better adapt multi-core or many-core architecture. Having identified the main bottlenecks of this method, we restructure the algorithm and propose a better execution model for it. The chapter 6 synthesizes different parts and concludes.

Chapter 2

State-of-the-art in Parallel Computing

2.1 Task-Centric vs Data-Centric

The discussion of parallel computing is only meaningful when involving the underlying parallel architectures. But a simple addition of CPUs will not automatically speed up the serial programs. The awareness of the available processing units, or the ability to spawn and manage additional "workers" to dispatch a portion of work to, is the watershed between the serial and the parallel programs. Here the term "worker" means a sequence of instructions that can be managed independently by a scheduler from the operating system. More specifically, it can refer to a thread or a process [1]. Based on how the work to be done is partitioned among the workers, there are two major categories of parallelization: **task parallelism** and **data parallelism**. Commonly the workers will interact with others during execution. But there is a special case where no communication needs to be performed across the workers. Such case is called "embarrasingly parallel". Derogatory as this term may suggest, it is nevertheless the ideal case in parallel computing, especially for the "communication-avoiding" approach of the algorithmic design.

2.1.1 Task Parallelism

In a task-centric approach, the workload is decomposed into separate tasks. One can think of the original problem being broken down into smaller subproblems. Each subproblem can be solved by a task. In essence, a task is a series of instructions operating on its own input data. Different tasks may have dependencies between them. The establishment of

[1] The processes and threades are both independent sequences of execution. What makes them different is that threads (of the same process) run in a shared-memory space, while process run in separate memory spaces. A process has at least one thread, but it can also consist of multiple threads.

this bond is mainly due to the data availability, when the input data required by a task are the intermediate results produced by other tasks, or the paternity could also be the reason, meaning a task is explicitly spawned by another task. In programming, the relationship between tasks can be expressed variously. The dataflow model is one way of doing it. In this approach the programmer specifies the tasks, their data usage, and the order between them. Knowing the information on the data usage and their relative order, it is possible to deduce the data dependencies which forms an implicit directed acyclic graph (DAG) connecting the tasks. In this DAG, the nodes represent the tasks and the edges denote the dependencies. The graph representation provides a basis for scheduling and allows the tasks to execute asynchronously in parallel. To name a few prominent examples, PaRSEC [2] [142, 29], which is designed for distributed many-core heterogeneous architectures, QUARK [3] [42], which is designed for shared-memory environment, SMPSs [4] [108], which is designed for shared memory multi-core or SMP (Symmetric Multiprocessor system), and StarPU [5] [8], which is designed for hybrid architectures and heterogeneous scheduling, fall into this category.

An alternative perspective is to explicitly define the spawn of child tasks within the scope of parent task. This method depicts directly the task graph without inference. But in contrast to the dataflow model where the execution order follows the dataflow from top to down in a DAG, this one, considering the possible dependence of parent task on child task, prefers rather a depth-first execution. These discussions are built on a global view across all workers. In the later section (2.2) we will stand on the position of a worker to shed more details on scheduling strategies.

In this thesis we mainly consider the fine-grained task parallelism on multi-core or many-core shared memory environment. Fine-grained parallelism means individual tasks are relatively small in terms of code size and execution time. A typical fine-grained task takes 10,000 to 100,000 instructions to execute, which means thousands of mathematic calculations. Two scheduling policies will be evaluated: the work-sharing and work-stealing. Unlike dataflow-based scheduling implemented in PaRSEC, QUARK, SMPSs or StarPU, the work-sharing and work-stealing do not based on the runtime dependency inference. As a result, they introduce lower runtime overhead. More details will be found in section 2.2.

[2] Parallel Runtime Scheduling and Execution Controller

[3] QUeuing And Runtime for Kernels

[4] SMP Superscalar framework

[5] "*PU" may stand for CPU or GPU. The name implies that this runtime system is designed for heterogeneous multicore architectures.

Instruction-Level Parallelism (ILP)

Although the ILP-related domains (microarchitecture, compiler) are beyond the scope of this thesis, having a good understanding of them is crucial to the study of parallel computing.

If we consider an extreme case where each task is as simple as an instruction, the relevant dataset is also reduced to the operands required by the instruction, then we are actually facing the intruction-level parallelism. Obviously, the parallelism of this kind is too burdensome for humans to identify, we can not apply the same rules that we discussed before but to resort to the compiler or hardware. The modern computer architecture has virtually pipelined and superscalar structure. The pipelining technique splits each instruction into a sequence of stages so that multiple instructions may execute at the same time as long as they are not in the same stage. This keeps every portion of the processor busy with some instruction and increase the overall instruction throughput. A superscalar CPU moves one step forward with more than one pipeline or at least replicated functional units which allows the multiple issue of instructions. Many processors also support out-of-order execution [6]. However, all these 3 techniques (pipelining, superscalar, out-of-order execution) come at the cost of increased hardware complexity. There is another approach which limits the scheduling hardware while enabling the simultaneous execution of instructions: very long instruction word (VLIW). It relies on the compiler to determine which instructions to be executed in parallel and to resolve the conflicts. Due to this fact, the compiler becomes much more complex [32].

2.1.2 Granularity

If the ILP is viewed as task parallelism with minimal size of task, the opposite edge case would be a program with one single task, such as a serial main function, which implies the absence of parallelism. Therefore we must introduce the notion of granularity when talking about task parallelism. It describes the amount of work or task size that a processing element can execute before having to communicate or synchronize with other processing elements. A bad choice of grain size really hurts the parallel performance. With coarse-grained tasks, there may not be enough of them to be executed in parallel. Or some processing elements may get into idleness due to irregular running times with no further tasks to run, causing load unbalance. A large number of fine-grained tasks, in contrast, will cause unnecessary parallel overhead. Notwithstanding the inexact nature of setting the best grain size, we will discuss a more exercisable approach of determining the task size based on the manycore architecture.

[6] Out-of-order execution, or dynamic execution, refers to the paradigm that the processor executes instructions in an order determined by the availability of input data, rather than by their original order.

2.1.3 Data Parallelism

Instead of focusing on the works that can be parallelized, data parallelism cares about the data that can be processed identically. The data parallel programming model [104] is often associated with SIMD (Single Instruction Multiple Data) machines [105]. According to Flynn's taxonomy [48] illustrated in Figure 2.1, the computer architectures can be classified by their control structure along the metrics. The Flynn's taxonomy is explained in detail in Table 2.1. The instruction dimension specifies the number of instructions that may be executed at once. While the data dimension indicates the number of data streams that may be operated on at once. A classical von Neumann system is a SISD (Single Instruction Single Data) system. Conceptually compared with the SISD system, the SIMD sytem has a single control unit and duplicated arithmetic logic units (ALU) so it can broadcast the instruction from the control unit to the ALUs, and operate on multiple data streams by applying the same instruction to multiple data items. Typical examples of SIMD system include vector processor or graphics processing unit (GPU). They all provide the functionality of processing an array of data elements within a single instruction. The model for the NVIDIA GPU is "Single Instruction, Multiple Threads", or SIMT. Because of some differences in architectural design [7], SIMT offers more flexibility and expressivity in programming. Nevertheless, SIMT itself is not designed for task parallelism. Because the task parallelism requires the threads being capable of executing different code on different data set. The "multiple threads" of NVIDIA GPU within a "warp" [8] always execute a "single instruction".

Most modern CPU implements SIMD units. This reminds us of early vector processors, such as Cray-1 [113] back in 1970s. Like the vector processors, SIMD units are fully pipelined. However, they are still different in some ways. Vector architectures offer an elegant instruction set which is intended to be the target of a vectorizing compiler. The "SIMD" in CPUs borrows the SIMD name to signify basically simultaneous parallel data operations. As an extension, it is found in most today's instruction set architectures (ISA) that support multimedia applications [9]. Multimedia SIMD extensions fix the number of data operands in the opcode. In contrast, vector architectures have a vector length register that specifies the number of operands for the current operation. Multimedia SIMD usually does not offer the more sophisticated addressing modes of vector architectures, namely strided

[7] Both SIMD and SIMT share the same fetch/decode hardware and replicate the execution units. But compared with SIMD, SIMT has multiple register sets, multiple addresses, and may have multiple flow paths that SIMD does not have.

[8] "Warp" is the basic unit for the SIMT model. A single instruction is issued each time for the threads within the same warp.

[9] For x86 architectures, the SIMD instruction extensions started with the MMX (Multimedia Extensions) in 1996, which were followed by several SSE (Streaming SIMD Extensions) versions in the next decade, and they continue to this day with AVX (Advanced Vector Extensions).

accesses and gather-scatter accesses. However, this is not true for the SIMD instruction set named "Initial Many Core Instructions (IMCI)" that Intel implements for its many integrated core architecture (see subsection 3.4.1). Finally, Multimedia SIMD usually does not offer the mask registers to support conditional execution of elements as in vector architectures. Again, this is not true in IMCI.

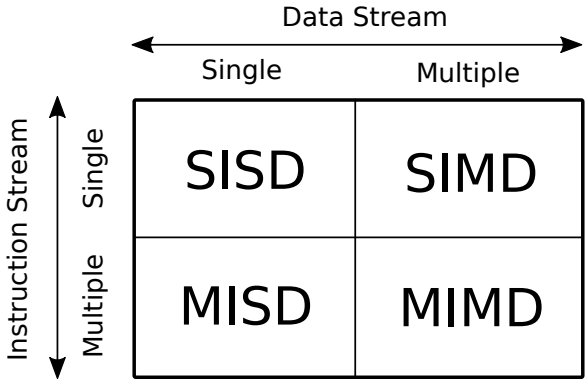


Figure 2.1 Flynn's parallel architecture taxonomy.

Table 2.1 Explanation for Flynn's taxonomy.

Type	Specific
SISD	Single instruction stream, single data stream. It is used in a sequential computer that exploits no parallelism in either the instruction or data streams.
SIMD	Single instruction stream, multiple data streams. It refers to the architecture that exploits multiple data streams against a single instruction stream. Vector processor, SIMD units or GPU fall into this category.
MISD	Multiple instruction streams, single data stream. Multiple instructions operate on a single data stream. Such architecture is uncommon and usually designed for fault tolerance.
MIMD	Multiple instruction streams, multiple data streams. Multiple autonomous processors simultaneously executing different instructions on different data. This is how we exploit parallelism in today's computer systems, including distributed systems, or multi-core superscalar processors.

2.2 Runtime

2.2.1 CPU vs GPU

All forms of parallelism need to be expressed by instructions. In CPU, the logical (instead of physical) executor of instructions at runtime is process or thread. A process has at least one thread, the main thread. Here the term "thread" implies the software thread, which is different from the hardware thread. The software threads are those that program creates. While the hardware threads are physical resources. We may draw an analogy between software/hardware thread and runner/track. The number of hardware threads is upper bounded by the processor's microarchitecture (the number of cores, the program visible registers, the processor control registers such as the program counter, the shared resources such as caches and TLBs [10], etc.). When the software threads outnumber the hardware threads, it incurs systematically additional overheads, such as expensive context switching (register state, cache state, memory thrashing, etc.) or aggravated lock convoy issues. Therefore we are not particularly interested in this case. In the following discussion, without specific indication, the number of software threads will respect the limit of the hardware threads. Since the most efficient number of threads is variant acrossing the hardwares, when the program does not target a specific processor, programming in terms of threads is not an ideal solution for generality reasons. It then leaves us no choice but to resort to task to express the parallelism. A task refers typically to a small routine that can be runned by a thread. Instead of assigning one task per thread (thread-based programming), we rely on some mechanism that maps a big number of tasks onto threads (task-based programming). This is advantageous because a task is lighter weight than a thread. It takes much less time starting and terminating a task than a thread. It also allows to express a higher degree of fine-grained parallelism and thus feed the processor in a more balanced way.

The GPU, however, follows a different logic, especially when it concerns the NVIDIA GPUs and CUDA [11] programming. As discussed before, GPU is a data-parallel device. It assumes the workloads with plentiful data parallelism, which means it goes for high throughput, high latency scenario in the first place. The hardware CPU invests to bring down the single-thread latency, such as the superscalar execution, register renaming, branch prediction, speculative execution/prefetching or others, does not fit in the GPU's philosophy. In fact, switch to another hardware thread in CPU happens only when all these other measures failed. It helps to fill the stalls but it also hurts single-threaded performance. Typically the

[10] TLB is short for translation lookaside buffer. It is a cache that memory management unit (MMU) uses to improve virtual address translation speed.

[11] CUDA stands for Compute Unified Device Architecture. It is a parallel programming platform and application programming interface model created by NVIDIA.

CPU has very few register sets because of the hardware complexity and latency requirement. GPU, on the other hand, has simpler hardware so it could afford a lot more registers, supporting instantaneous context switching of many threads. What's more, the GPU has implemented zero-overhead thread scheduling [103]. In other words, when a stall is met, such as a long latency memory request, the GPU won't wait idly, it just switches straightaway to the other ready threads. Unlike CPU, threading is the first stall-mitigating measure for GPU.

2.2.2 CPU task scheduling

Now let us revisit the task-based programming in CPU, where the program is constructed of tasks. We should need a scheduler that is in charge of issuing the tasks to threads. It has to respect the data dependencies between the tasks, according to how we inform it.

Work-sharing

The best case scenario is that tasks are independent, or loosely dependent, which means the sensitive data are under protection, it is free to schedule the tasks in arbitrary order. For example in an reduction of an array, the task is defined as the summation of a partial array to a local variable. As long as the update to the final result is atomic because it is shared by all threads, the execution order of the tasks is not a concern. We can also adjust the granularity of task by resizing the partial array. Under such assumption, the easiest way is to split the tasks statically into as many parts as there are available threads. No further scheduling needed, almost zero overhead. However, doing so may result in severe load imbalance, when some of the threads terminate more quickly than others. A more balancing solution is to maintain a central task pool for all threads. Every time a thread finishes its work, the scheduler assigns to it a new portion of the leftover tasks. This often leads to better workload distribution by introducing addtional synchronization costs.

Work-stealing

The tasks usually have relations between them. Similar to the famous "divide-and-conquer" algorithmic design approach, the tasks can be created recursively. The dependency between the parent task and the child task is clear. Each thread has its private task pool. When some thread empties its pool, it steals work from another random victim thread, so no centralized control is necessary. Different ways of traversing the task tree correspond to different task scheduling strategies. Breadth-first execution maximizes parallelism while depth-first execution has good use of memory and cache. A strategy is deemed good as

long as it creates enough parallelism to keep threads busy, and keeps memory consumption reasonable as well.

Dataflow

Instead of specifying the relation between tasks, we can count on the scheduler to figure out the tasks' dependencies. The tasks are submitted to the scheduler in a sequential order. The information the scheduler receives includes the content of the task as well as its data usage (whether a piece of data is used as input, output, or both). The scheduler will then construct a directed task graph according to the submission order of the tasks and their data usages.

2.3 Shared Memory Parallel Programming APIs

In this section we'll list some of the commonly used shared memory (see section 3.3) parallel programming APIs that based on the techniques described in 2.2.

2.3.1 OpenMP

OpenMP [36] is an extension to C/C++ and Fortran languages that defines a thread-based shared memory environment. It is one of the earliest attempt to easily parallelize a serial program. In C, a serial program can be transformed into parallel by simply inserting OpenMP `#pragma` preprocessor directives into source code. This advantage in programmability is called incremental parallelization. The compiler is responsable for interpreting the directives. The source code is still legal even if the compiler does not support OpenMP.

Besides the compiler directives, OpenMP also has a modest number of library calls and a runtime system that manages the parallel execution. The work-sharing scheduling explained in 2.2.2 is basically what OpenMP does after adding a directive above a `for` loop. OpenMP extends itself by introducing the task feature in its 3.0 version. More recently OpenMP adds explicit vector programming (simd pragmas) in its 4.0 version.

2.3.2 Cilk/Cilk Plus

Cilk [15, 51] extends C and C++ with constructs to express parallel loops and fork-join idiom. Using this language, the programmer is responsible for exposing the parallelism as well as identifying elements that can safely be executed in parallel. The rest of the work is left to the runtime environement, i.e. the scheduler, who divides the tasks between the processors. This design allows Cilk programs to run without actually specifying the number of the processors.

A basic parallel program can be written using two Cilk keywords: `spawn` and `sync`, (spelled `_Cilk_spawn`, `cilk_spawn`, `_Cilk_sync`, `cilk_sync` in Cilk Plus). The former creates a new child task, whereas the latter sets a synchronization point. The scheduling of task in Cilk is based on work-stealing described in 2.2.2. The Cilk also supports reducers and for loop parallelizer (use `cilk_for` to replace `for`). For vector features, Cilk Plus adds array notations that allow users to express high-level operations on entire arrays or sections of them (`x[0:n]` to express a vector of size n). These notations help the compiler to effectively vectorize the application.

Initiated within MIT [12], Cilk is now acquired by Intel. It is also supported in the latest GNU Compiler Collection from version 4.9.

2.3.3 TBB

Threading Building Blocks (TBB) [111] is a C++ template library which provides data structures and algorithms to exclude the use of low-level threads (POSIX threads, Windows threads or Boost threads). TBB benefits from the low-overhead polymorphism through compile-time optimization by using templates extensively. Thanks to the object-oriented features, TBB encapsulates a collection of algorithms (`parallel_for`, `parallel_reduce`, `parallel_scan`, `parallel_while`, `parallel_do`, `parallel_pipeline`, `parallel_sort`, atomic operations) and data structures (containers, scalable memory allocators, mutexes). The high-level algorithms are all built on task scheduling similar to Cilk (refer to 2.2.2).

2.3.4 QUARK

QUARK (QUeueing And Runtime for Kernels) [42] is a runtime environment for the dynmaic scheduling and execution of applications that consist of precedence-constrained kernels on multicore, multi-socket shared memory systems. Unlike OpenMP, Cilk/Cilk+, or TBB, QUARK is rather a middleware (or runtime) system than a programming language extension which defines a way to directly express parallelism in the code. Instead, QUARK users implements QUARK-style tasks using plain C language and submits them into the runtime system so that the data dependency DAG can be inferred during runtime.

QUARK implements the data flow model described in 2.2.2, where scheduling is based on data dependencies between tasks in a task graph. The data dependencies are inferred through a runtime analysis of data hazards implicit in data usage by the kernels. QUARK is used as the runtime engine in PLASMA (Parallel Linear Algebra Software for Multicore Architectures). Another famous similar academic project is StarPU [8] from INRIA Bordeaux.

[12] Massachusetts Institute of Technology.

2.4 Performance Metrics

There are several dimensions to evaluate a parallel program. Given the computing potentials of the system and the nature of the application, we shoud be able to sketch the optimal parallel scheme and assess the performance of the actual parallel program. Two indicators are often used to characterize the capability of the computing system: CPU computing power calculated in floating-point operations per second and memory bandwidth computed in transmitted bytes per second. The former is bounded by the frequency, length of vector units, fused operations, and the number of cores. The latter is bounded by the core memory interface, the memory controllers.

Let T_p denote the time of computation using p processors, then the span is defined as T_∞ which indicates the length of the critical path (longest series of operations respecting the data dependencies). Using this representation, the speedup is defined as: $S = T_1/T_p$. A scalable algorithm exhibits linear speedup. The parallelism is defined as: $P = T_1/T_\infty$, which represents the maximum possible speedup on any number of processors. Let W be the workload, $a \in [0,1]$ denotes the fraction of the workload that can not be parallelized, then

$$S = T_1/T_p = \frac{W}{(aW + \frac{(1-a)W}{p})} = \frac{1}{a + \frac{1}{p}(1-a)} \tag{2.1}$$

This is the famous Amdahl's law [5], which models the speedup between serial and parallel implementations of an algorithm. It assumes the problem size remains the same when parallelized, which corresponds to the case of ***strong scaling*** when the scalability of a parallel program is concerned. The scalability is the capability of a parallel program to use more computing resources. In practice, as more computing resources become available, they tend to get used on larger problems with bigger datasets. Compared with Amdahl's law, Gustafson's law [88] gives a more realistic evaluation of parallel program. The key assumption of Gustafson's law is that the total amount of work to be done in parallel varies linearly with the number of processors. In the base case with 1 processor, let W_s denotes the serial workload, W_p denotes the parallel workload. Then

$$S = T_1/T_p = \frac{W_s + pW_p}{W_s + W_p} \tag{2.2}$$

Let $\alpha = W_s/(W_s + W_p)$, we have $S = \alpha + p(1-\alpha) = p - \alpha(p-1)$. Its corresponding case in scalability analysis is ***weak scaling***, where the per processor problem size is fixed.

At this point, we have defined all the variables and parallel models based on which we will build the rest of this thesis.

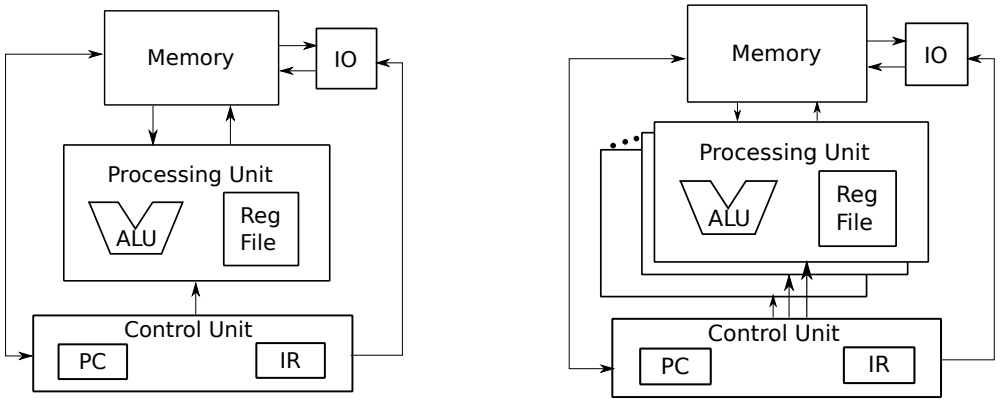
Chapter 3

Architectural Considerations for Parallel Program Design

Understanding the computer architecture is essential to writting the efficient code. This is especially true in the domain of high performance computing, even more true when there's a dedicated accelerator for enhancing the computing power.

3.1 Abstraction of Computer Architecture

Modern computer architectures can still be described as von Neumann architectures [135] on a very high level of abstraction. Based on von Neumann model, the Figure 3.1 shows a simplified diagram of how the hardware is typically organized to execute programs that are represented at the instruction set architecture level (ISA).



(a) von Neumann architecture.

(b) von Neumann counterpart for SIMD machine.

Figure 3.1 Von Neumann architecture.

The von Neumann model describes roughly main phases of execution: instruction fetch (program counter, memory), instruction decode (instruction register), memory fetch (memory, register files), execution (ALU and register files in processing unit), write-back (register files, memory). Though the von Neumann model is not particularly fastidious about the pipelined execution, hierarchies of memory, or separation of instructions and data, which is present in Havard architecture [56]. In a way, the contemporary CPU looks to the programmer just like a von Neumann machine. We can think of a thread as a virtualized or abstracted von Neumann processor. In this thesis, we will be focusing on two hardware features that empower the parallel computing: the multithreading and the vector processing (SIMD). The former can be pictured as multiple of Figure3.1a participating in computation. Whereas the latter can be pictured as Figure 3.1b: the duplicates of processing unit operate on an array of elements within a single instruction.

3.2 Memory Hierarchies

Some applications are CPU-bound, in which the speed of the processor is the key factor. In scientific computing, we often encounter the applications that are memory-bound or memory-bandwidth bound. There's a direct reason for that. Over the past decades devices used to design CPU have always 10 times faster than those used in memory [124]. RAM access time have only improved at roughly 10% per year since 1970 [74]. One reason for that is economics. We want the computer system to support most applications and thus slower and cheaper devices are used to design them compared to CPU. Secondly, the use of fast memory could be cumbersome. Fast memory such as SRAM or static RAM is typically made

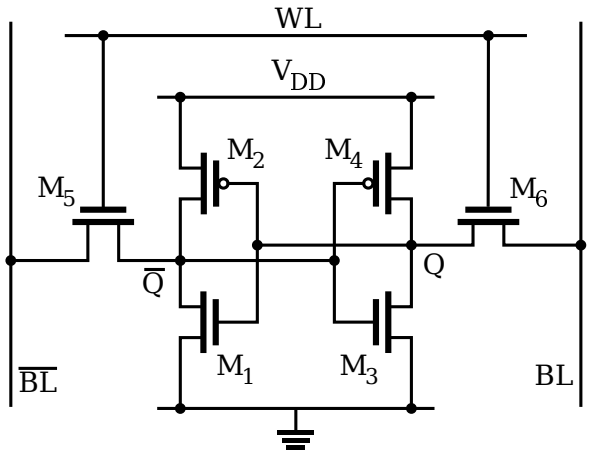


Figure 3.2 A six-transistor CMOS SRAM cell.

up of six-transistor CMOS cells, as illustrated in Figure 3.2. The cross-coupled inverters (M1,

M2, M3, M4) are used to store a single bit in an SRAM, which actively drive the levels (low or high) of bit lines (BL) in the cell. This configuration provides higher bandwidth compared to DRAM where each bit of data is stored in a charge storage cell consisting of one capacitor and transistor. Given the structure of the unit bit cell of memory, the SRAM is evitably less dense than DRAM and thus takes more space. The ever-larger speed mismatch between CPU and memory may lengthen the idle time of CPU while retrieving the data, which aggravates the performance bottleneck.

This is also the problem of von Neumann model. It assumes data are loaded immediately from memory to the processing unit. In reality, this is extremely inefficient due to the memory wall [143]. A typical load from memory usually takes beyond 100 cycles, while a processor may perform several operations per cycle. This huge speed mismatch urges us to refine the von Neumann model in order to bridge the gap. Under the assumption that the data are unlikely to be used only once, which is true in most of the programs, keep temporarily the recently accessed data somewhere faster than memory should be a good idea. In practice, there are at least 4 levels of media that keeps the data and the instructions: registers, cache (usually multiple levels), memory and disk. As mentioned before, the pitfall of the fast storage media is their volume. In other words, they trade the lightness of size for speed.

Before going to the details, we need two more notions to depict the speed of memory. When the processor issues a request for a memory item that is not available immediately, the initiatives of delivering will take time. The delay between the request and the arrival of what is requested is defined as latency. After the initial latency is overcome, the data will come at a certain rate dominated by the clock and the width of the bus. This rate is also called bandwidth. If let α represents the latency while β the inverse of the bandwidth (the time per byte), then the time a message of n bytes is delivered:

$$T(n) = \alpha + \beta n \tag{3.1}$$

3.2.1 Memory

The memory is made of DRAM and organized as DRAM banks. The Figure 3.3 shows the typical organization of a DRAM bank, where each bit is stored in a tiny capacitor, usually the byproduct of building a transistor, what we called the parasitic capacitance. It is illustrated in the upper right corner of the Figure 3.3. Reading from such a cell in the core array is very slow. The DDR3 or GDDR4 ("DDR" for double data rate, and "G" for Graphics) is a recent type of DRAM with a high bandwidth interface. Its core speed is only $\frac{1}{8}$ of interface speed. And it is likely to be worse in the future. That's why modern DRAM systems are designed to be always accessed in burst mode. For SDRAM (Synchronous DRAM) cores

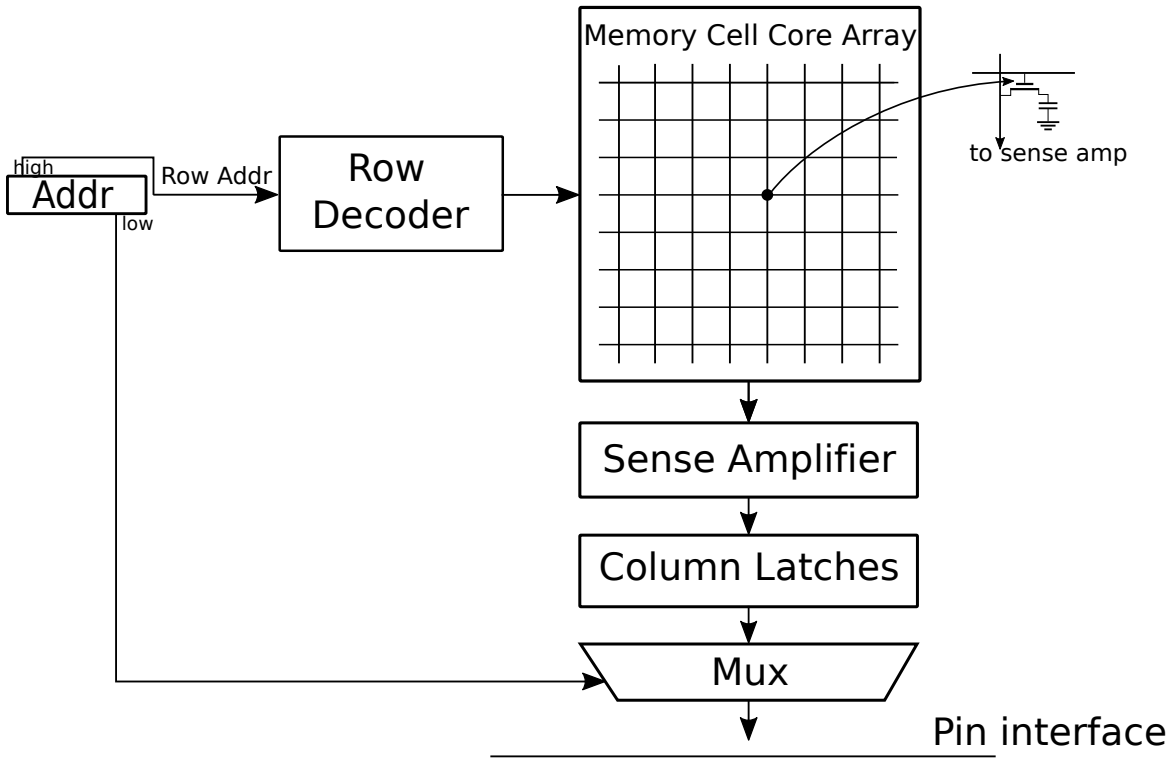


Figure 3.3 DRAM bank organization.

clocked at $\frac{1}{N}$ speed of the interface, load N times of interface width of DRAM bits from the same row at once to an internal buffer, then transfer in N steps at interface speed. The Figure 3.4 compares the timing of burst and non-burst modes of a DRAM bank. The bursting

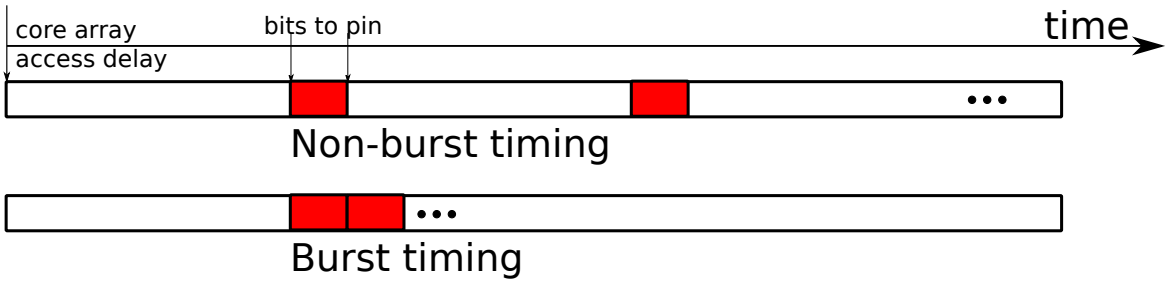


Figure 3.4 DRAM bursting vs DRAM non-bursting.

bridges the gap between the SDRAM core speed and the interface speed. Each address space is partitioned into burst sections, whenever a location is accessed, all other locations in the same section are also delivered to the processor within only one DRAM request. This fact also explains the use and length of a cache line that will be covered in the next subsection. A bank can only service one request at a time. Any other accesses to the same bank must wait for the previous access to complete, known also as bank-conflict. In contrast, memory access

to different banks can proceed in parallel. The Figure 3.5 shows how bank level parallelism can help to avoid bank conflicts. It also pictures vividly the relationship between the latency and the bandwidth.

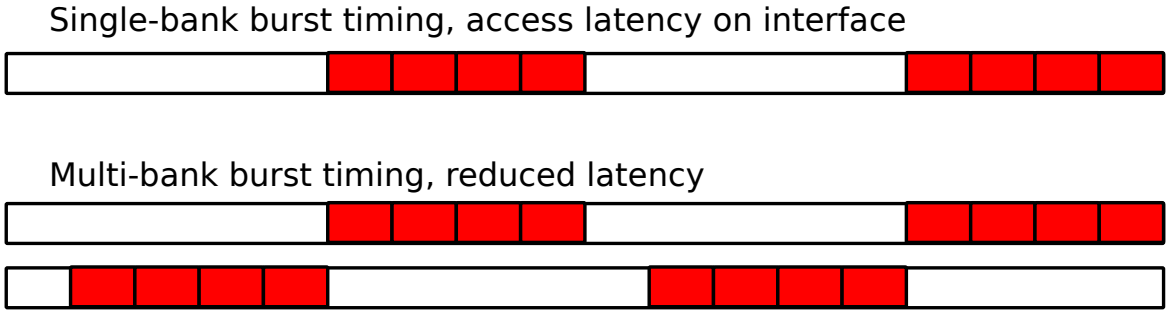


Figure 3.5 Avoiding bank conflicts: DRAM bank level parallelism.

In C/C++, memory exposes itself to a process as 5 different areas: code (text segment), initialized data (data segment), uinitialized data (bss segment), heap, and stack, as indicated in Figure 3.6. nd the bandwidth. Programmers have the control of the allocations for stack,

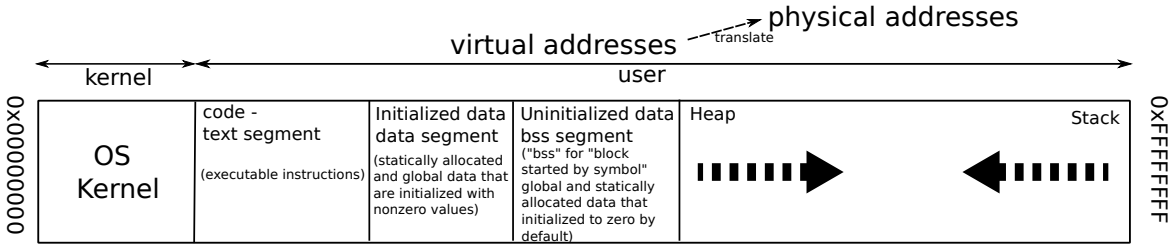


Figure 3.6 C/C++'s process memory layout on an x86.

heap and global values. The stack is the scratch space for a thread. The OS allocates the stack for a system-level thread when the thread is created. It is always reserved in a LIFO (Last In First Out) order for local (automatic) variables. The heap is where dynamic memory comes from. Unlike the stack, there is no enforced pattern to the allocation and deallocation of blocks from the heap. It is often used as the workspace for processing a large chunk of data, like a matrix. In a multi-threaded application, each thread will have its own stack. But all the threads will share the heap.

In fact, all the program layout in the address space is virtually the same. The virtual address (also called logical address) needs to be translated into the physical address at the runtime. This dynamic relocation implemented in the modern OS provides a protected zero-based address space, and allows the process to use larger memory than what is available in RAM chips (swap with the hard disk). The virtual address translation sometimes could be

penalizing to the performance. We'll see more details in the next subsection when it comes to the TLB cache (translation lookaside buffer).

3.2.2 Cache

Cache has lower latency and higher bandwidth than memory. It is typically consist of static random-access memory (SRAM), which is faster and more expensive than DRAM. The data from memory have to travel through the cache to wind up in registers. If no eviction is triggered, the data item will be stayed in cache, allowing a much more faster access the next time it is requested. However, distinct from a register load which is done explicitly by assembly instructions, the transport of data from memory to cache is implemented in the hardware level. In other words, the direct cache manipulation is out of the programmer's control. Yet a good cache usage is the key to the high performance. Therefore a precise understanding of cache is the premise for influencing the cache use in a good implicit way. If the data item does not reside in the cache when it is requested, a cache miss occurs [1]. In general, there are 3 types of cache misses.

Compulsory cache miss The very first time a data item is loaded from memory. The word "compulsory" implies the unavoidability of such case.

Capacity cache miss When the data item, once in cache, has been overwritten simply because the cache only has a limited size.

Conflict cache miss When two data items are mapped to the same cache slot, even as there may be empty slots. Such a conflict would force an unnecessary eviction.

Besides, in a multicore context, the invalidation cache miss may happen when a data item in cache has been invalidated due to the change made by a another core. The core having the invalid data has to reload the corresponding address.

In either compulsory, or capacity case of cache miss, the system may have to decide which data is going to be evicted. A general principle, called Least Recently Used (LRU), is usually applied as the cache replacement policy. As the name suggests, the least recently used data will be overwritten with the new item, or flushed. As for conflict cache miss, it usually occurs in direct-mapped caches, where each memory address maps to a unique address in cache by taking instantly the least significant bits of the memory address as the cache address. The conflict problem will be eliminated if any data item could go to any cache location. This makes a fully associative cache. However, such a cache is very costly in the sense of both

[1] The opposite case is called a cache hit.

price and speed (more complicated address calculations). As a trade-off between the cost and the conflict, the modern cache is often designed to be k-way associative, meaning a data item can go to any of k cache locations.

In fact, the data in cache are organized in terms of cache lines, for the reason explained in section 3.2 that a DRAM memory request brings back a whole burst section. This fact can not be ignored in an efficient program design. A cache line normally has the length of several words. In a typical instance it contains 8 double precision floating point numbers (64 bytes). When a compulsory cache miss is met, there is no way to mitigate the latency and the bandwidth just by the presence of cache. However, the access to the rest of the cache line is effectively free. What's more, the second time the same data item is referenced, it may still be in cache, accessible at greatly reduced latency and amplified bandwidth than memory. So the data should be exploited in cache long enough before they are done with the computation.

As mentioned before, a compulsory cache miss pays the price for latency and bandwidth. That said, it does not have to penalize the performance. Today's microprocessors can detect the memory access pattern, predict which memory locations will be needed, and issue prefetches to them. As programmers, we can also hint the compilers to add explicitly the fetch instruction in order to "software prefetch" the data during the computation. This philosophy of pipelining is almost everywhere in parallel computing world. Even without the prefetch, the "out-of-order execution" feature of modern processors allows them to switch the order of irrelevant instructions and perform other operations during a memory stall [2].

In addition to data and instruction cache, there is this special cache that accelerates the virtual memory addresses translation (see subsection 3.2.1). The virtual memory is managed in terms of fixed-length contiguous block, called memory page, which is analogy to a cache line. The operating system uses a page table to store the mapping between the virtual addresses and physical addresses. Same as a memory load, the address translation by lookup is slow. The solution is to prepare a cache for frequently used page table entries: translation lookaside buffer (TLB). A TLB miss will necessiate an access to an access to the page lookup table, which is costly. A feasible way to keep down the TLB misses is to allocate larger-sized memory pages, if the system supports.

3.2.3 Register

The registers store immediate input and output data for instructions. They are on-chip and internal to the processor. By reading the assembly code one can figure out which register the instruction is using for some certain operation. They do not have addresses but distinct names.

[2] the execution stalls when data being fetched from memory

Like cache, we want the data stayed in registers being reused as much as possible. However, since the register file (the ensemble of registers) has only a very small size, forcing too many quantities in register only go opposite. In C, the programmer may guide the compiler to declare a register variable: `register int i;`.

3.3 Memory Access In Parallel

The foregoing discussion of memory hierarchies somewhat begs the question of how the memory is accessed by the processor, especially in a multi-processor context. For a parallel machine, the choice has to be made between two alternative schemes: either to let each processor has its own memory with individual address space, or to share a unified address space, if the memory is accessible to all the processors. The former case is referred to as *distributed memory*, whereas the latter as *shared memory*.

3.3.1 Message Passing Interface

In fact, the physically unified memory can also be logically distributed, the difference lies only in the number of processes we launch the program with. As pointed out before (see section 3.2), each process has its own allocatable address space (the heap) shared by all its threads. Different processes can not see into each other's memories. But they are allowed to exchange information through explicit message passing. In parallel programming, this is always taken care by MPI (Message Passing Interface) [54, 55]. MPI is a portable message-passing system designed to function on a wide variety of parallel computers. MPI is a language-independent communications protocol which provides essential virtual topology, synchronization, and communication functionality (both point-to-point and collective) between a set of processes. MPI has displaced most other message-passing libraries, such as PVM (Parallel Virtual Machine) [53], because of its advantages of portability and speed. Another benefit of MPI model against explicit shared memory models (see section 2.3) is that works better for NUMA architectures (see subsection 3.3.2) because MPI encourages memory locality.

3.3.2 UMA vs NUMA

UMA stands for uniform memory access. It defines a shared memory environment where the memory access time is identical for whichever processor or memory location. Such system is called SMP (Symmetric Multi-Processors). Multicore processors may have UMA in cache level through a shared cache. The opposite of UMA is NUMA, non-uniform memory access,

where memory at various point in the address space of a processor have different performance characteristics. A typical scene for NUMA is accessing the memory directly attached to the other socket from the local one in a dual-socket CPU, such as Intel Xeon processors. The traversal of the QPI (Intel QuickPath Interconnect, a better version of the front-side bus connecting the CPU and the memory controller), in spite of the "quick" in the name, introduces addtional latency overhead. Modern processors have multiple memory ports, and the latency of access to memory varies depending even on the position of the core on the die relative to the controller. We will analyze a concrete example in section 3.4.1. As long as bringing memory nearer to processor cores is a critical concern for performance, NUMA will play an increasingly important role in it.

3.4 Heterogeneous Architecture

Heterogeneous architecture refers to the computing system consisting of more than one kind of processor. The idea here is to bring together the best of dissimilar processors, usually a general-purpose CPU being associated with another device that has specialized processing capabilities to handle particular tasks. The CPU is designed to be latency-oriented in order to run the operating system and host all kinds of applications. But for certain workload that necessiates a massively parallel solution, the setting of modern CPU does not make it the best candidate to provide one. The associated processor, on the other hand, is born to accelerate those tasks. So it is also known as the accelerator. The accelerators are often designed to be throughput-oriented to digest parallel workloads while attempting to maximize total throughput, at the expense of increased latency on individual tasks.

Strictly speaking, the heterogeneity in the context of computing refers to different ISA. But it also can be used to describe the speed of different microarchitectures of the same ISA, which intend to offer the very division of work that we talked before. In today's supercomputer's layout, the heterogeneous system has proved its effectiveness [125, 47], and can be categorized into these two types. CPU+GPU, different ISA, and CPU+MIC, both x86 ISA. In the subsection 3.4.1 we will elaborate on MIC, which is one of the major working environment of this study.

3.4.1 Many Integrated Cores Architecture

MIC is short for Many Integrated Cores Architecture. It is a coprocessor computer architecture developed by Intel incorporating several earlier efforts on GPGPU, many-core system and cloud computing chip [71]. MIC functions similarly as GPU. They both appear as

the PCI device interacting with the CPU. Although there is still one difference that makes MIC more flexible: the MIC runs a micro OS on itself, so it can be used as a standalone device initiating/terminating the program, communicating with other CPU or MIC, without intervention of CPU. The GPU, however, needs to get the data and computing kernel from CPU, and transmit back the result to CPU. During the computation, the GPU may be able to contact directly other GPUs, without passing by the CPU memory, thanks to the new direct link technologies [52, 98, 99].

Up to now, there are two generations of MIC products available in the market. The first generation, codenamed Knights Ferry, is more of a prototype than the second generation, codenamed Knights Corner (KNC). The next generation, codenamed Knights Landing (KNL), will be publicly available at the end of 2015, or beginning of 2016. The branding of this processor product family is called Intel Xeon Phi. As stated previously, the Xeon Phi coprocessor uses x86-based cores sharing the same ISA with the host processor[3].

The Figure 3.7 depicts the microarhitecture of a single KNC core [4] based on Intel Pentium design. The core has dual issue pipelines (U-pipe and V-pipe) so it can execute two instructions per cycle [5]. The core pipeline is relatively short (7 stages for integer instructions and 6 more stages for vector pipeline) running at a low frequency (around 1.1 GHz), which leads to a good power efficiency. The instructions are coming concurrently from 4 threads or processes [6]. However, the prefetch function (PF, see Figure 3.7) works in a round-robin order when instructions are ready in the prefetch buffer, therefore the hardware can not issue the instructions from the same context in back-to-back cycles. Things could be worse when PF and PPF (see Figure 3.7) are not properly synchronized (stalled by, say, a cache miss). Having more than one thread running in the core would amortize the potential performance loss and hide the vector pipeline and memory access latencies.

The vector processing unit (VPU) illustrated in Figure 3.7 is the engine for executing vector instructions. The VPU in KNC implements a 512-bit vector ISA called "Intel Initial Many Core Instructions" (IMCI) that can execute 16 single-precision floating point or 32-bit integer and 8 double-precision floating point or 64-bit integer vector instructions. It has 218 new instructions, including gather/scatter and mask instructions, compared to those implemented in the Xeon family of SIMD instruction sets (MMX, SSE, AVX). The VPU

[3]The KNL will be available in two forms: as a coprocessor or a host processor (CPU). But before KNL, all KNC are presented as coprocessors, which can not be functioning without the host CPU.

[4]A KNC coprocessor has 61 cores. The coprocessor micro OS is running on "OS proc 0" which maps to the the first thread of the highest numbered core. So always leave the highest numbered core free from application threads to prevent interference from OS threads.

[5]One on U-pipe and the other on V-pipe. But V-pipe executes only a subset of the instructions and is governed by instruction pairing rule.

[6]All architectural states are replicated 4 times.

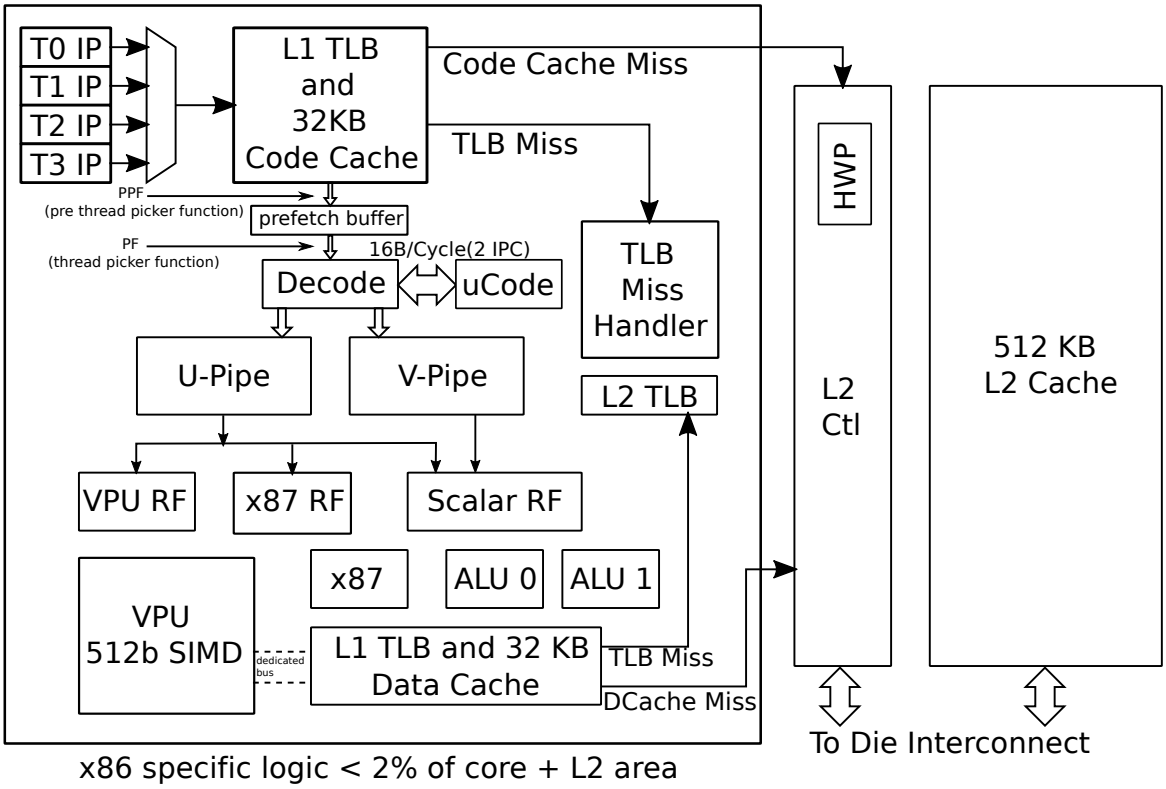


Figure 3.7 Microarchitecture of a single KNC coprocessor core. Each core has 4-way simultaneous multithreading (SMT), 2 pipelines, a vector processing unit, on-core L1 cache, and local L2 cache connected with the ring network (L2 cache is fully coherent between the cores).

receives its instructions from the core ALU and receives the data from the L1 cache by a dedicated 512-bit bus. The VPU is fully pipelined and can execute most instructions with 4-cycle latency and a single-cycle throughput. Each VPU underneath consists of 8 micro ALUs (UALU) each containing 2 single precision (SP) and 1 double precision (DP) ALU with independent pipelines. Each VPU instruction passes through all five pipelines to completion.

Table 3.1 Five vector pipelines for each KNC vector instruction to pass through.

Type	Specific
Double Precision (DP) pipeline	Execute float64 arithmetic, conversion from float64 to float32, and DP-compare instructions
Single Precision (SP) pipeline	Executes most of the instructions including 64-bit integer loads. This includes float32/int32 arithmetic and logical operations, shuffle/broadcast, loads including loadunpack, type conversions from float32/int32 pipelines, EMU (Extended Math Unit) transcendental instructions, int64 loads, int64/float64 logical, and other instructions
Mask pipeline	Executes mask instructions with one-cycle latencies
Store pipeline	Executes store instructions with one-cycle latencies
Scatter/Gather pipeline	Read/write sparse data in memory into or out of the packed vector registers

The specifications of these vector pipelines are listed in Table 3.1.

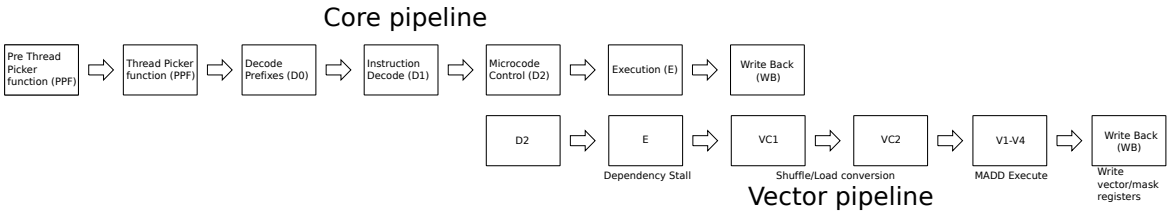


Figure 3.8 Vector pipeline stages relative to the core pipeline.

The vector pipeline is illustrated in Figure 3.8. The core pipeline is divided into 7 stages for integer instructions plus 6 extra stages (VC1, VC2, V1-V4) for vector pipeline. Once a vector instruction is decoded in stage D2 of the main pipeline, at E stage the VPU detects if there is any dependency stall. At the VC1/VC2 stage the VPU does the shuffle and load conversion as needed. At V1-V4 stages it does the 4-cycle multiply/add operations, followed by WB stage where it writes the vector/mask register contents back to cache as instructed. Most of the VPU instructions are issued from the core through the U-pipe. Some of the instructions can be issued from the V-pipe and can be paired to be executed at the same

time with instructions in the U-pipe VPU instructions. Each VPU has 128 entry 512-bit vector registers divided up among the threads, thus getting 32 entries per thread. These are hard-partitioned. There are eight 16-bit mask registers per thread which are part of the vector register file. They allow conditioned execution on the vector elements. The scatter/gather [7] vector memory instruction added in IMCI enables irregular memory access. What is more, the transcendental functions like exp, log, recip, sqrt are available in single-precision. Since the FMA (Fused Multiply-Add) is supported, the peak performance of KNC in terms of FLOPS is achieved when all 61 cores executing fused multiply-add VPU instruction on data within all SIMD lanes in full speed.

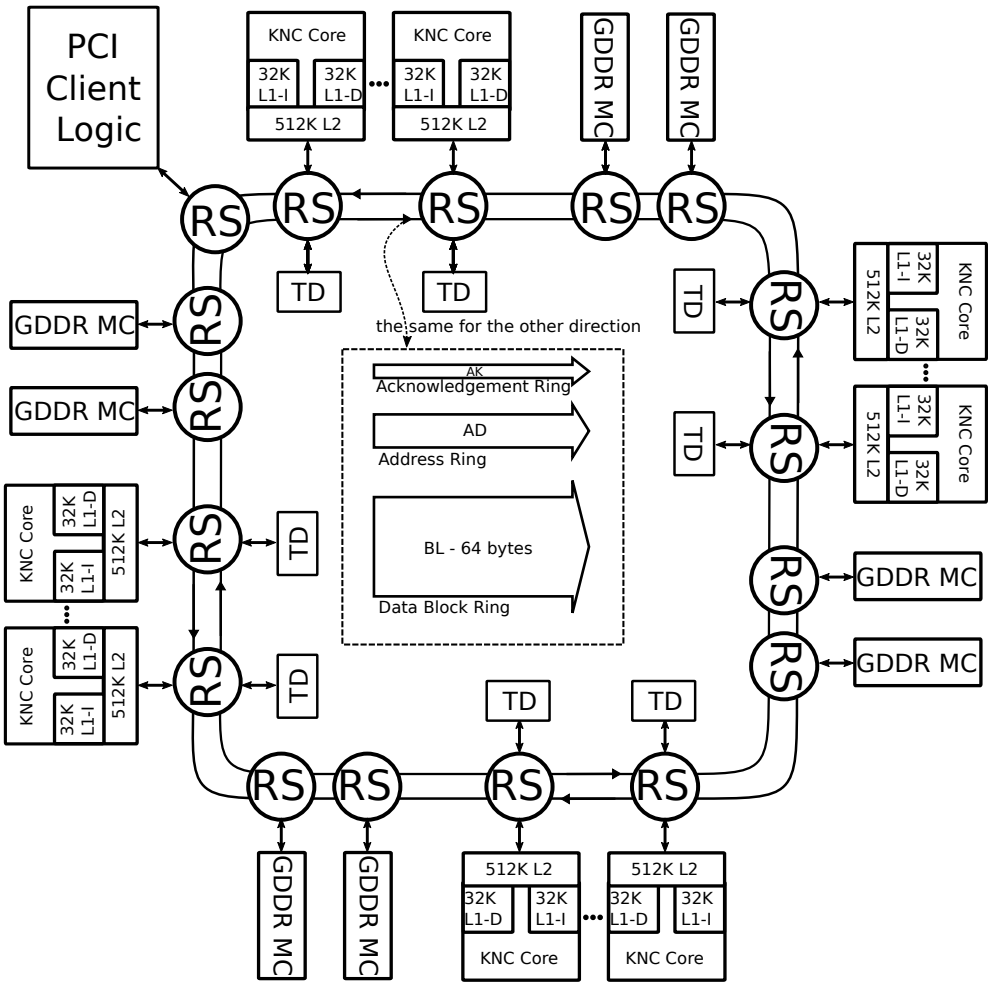


Figure 3.9 Microarchitecture of the entire KNC coprocessor. It consists of 61 KNC cores (see Figure 3.7), and KNC memory subsystem).

[7] The *gather* instruction fetches from abitrary memory locations indicated by an index vector, into a dense vector register. The *scatter* instruction does the opposite.

The Figure 3.9, along with a part of the Figure 3.7, illustrate how KNC memory system interacts with 61 cores. Each core has 8-way set associative 32 KB L1 instruction and 32 KB L1 data cache which supports an access latency of approximately 3 cycles. The L1 data TLB cache supports 3 page sizes (4 KB, 64 KB, and 2 MB). The L2 cache is the secondary cache which is inclusive of L1 cache. Its accessing latency is 11 cycles. The L2 cache local to a core is of size 512 KB, fully coherent between the cores using a set of tag directories (TD) linking the ring stop (RS). As a component of the ring interconnect, the RS manges the message passed to the ring, including the occupation and release of the ring. The bidirectional ring consists of 3 independent segments: the largest one data block ring of width of 64 bytes with high bandwidth, much smaller address ring to transmit read/write commands and memory addresses, and the smallest acknowledgement ring to send flow control and coherence messages. 8 GDDR5 memory controllers (MC) are symmetrically interleaved with groups of cores around the ring, each with 2 channels being capable of delivering 44 GB/s [8] of bandwidth. That makes a theoretical aggregate bandwidth of 352 GB/s. The core's memory interface are 32-bit wide with 2 channels sustaining a total bandwidth of 8.4 GB/s per core. The 61 of them can consume 512.4 GB/s (8.4 GB/s $\times$ 61), which is much higher than what memory controllers are capable of delivering [9]. This fact underlines again the importance of cache reuse, especially the reuse of L2 cache, which is kept consistent through the tag directories. There is an all-to-all mapping from TD to the MC. The memory addresses are evenly distributed across the MC and TD, providing a uniform access pattern. Each tag directory includes address, status, and the related L2 cache ID. Since the KNC is a sophisticated system with a large number of cores and coherent caches, L2 cache misses on KNC are more expensive and trickier to handle than the host processors. When an L2 cache miss occurs on a core, an address request will be sent on the AD ring to the tag directory that is related to the memory address for the cache miss [10]. According to whether the data of the requested address exist in another core's cache, the core generates another forwarding request and queries the corresponding core or the memory for the data. Once the data is fetched, it is forwarded to the core over the BL ring. This process costs one data block, two address requests, and 2 acknowledgement messages (by protocol) transmitted over the rings. The cost of data transfer is a function of the distance between the source and destination. It could be as long as hundreds of cycles.

Keeping these in mind we shall further imagine the application with irregular memory access pattern. Given the size of the L2 cache local to a core, it is very likely to trigger

[8] 5.5 G Transfers/s $\times$ 4 Bytes/Transfer $\times$ 2 channels

[9] Needless to mention the impact of ring network and overhead of ECC (Error-Correcting Code) memory, the actual ring bandwidth is only a fraction of the theoretical value.

[10] May not be the one linked to the core that triggered the cache miss event.

the L2 cache miss in this case. First of all, an L2 cache miss will definitely congest the ring bandwidth by a data forwarding request. Depending on whether the data are coming from the memory, or another cache, as well as the distance between the requesting core and the corresponding cache or memory controller, not to mention the address requests prior to the data transfer, the access latency may vary a lot regarding the core's location. This difference in the access latency practically cause a NUMA effect on KNC system. This NUMA effect would neutralize the computing power that MIC is supposed to provide and should be properly coped with. In chapter 4, we propose a method to eliminate this side effect of memory access and promote the memory locality.

The KNC can be functioning in three modes: "offload", "native", and "symmetric". The "offload" execution mode reminds us of GPU and CUDA programming. It implies the existence of the host processor.The application starts on host, the specified part of the code will be sent to the coprocessor through the SCIF (Symmetric Communication Interface), which is a low-level communication API across PCIe implemented for MIC. Compared with MPI, the SCIF is not easy to program with, especially for pipelining the computation and communication sort of things [58]. What's more, it provides inter-node communication only within a single host platform, and may be constrained by the evolving standard. Using "native" mode may also introduce power inefficiency and forfeits the processing capabilities of multicore host processors [82]. As KNL, the next generation of MIC, will be available in terms of host processor, the offload mode is really not a must way to go. The "native" mode is more compliant with the conventional CPU programming. As the coprocessor hosts a micro Linux OS in it, it can appear as an independent computing node. The program runs and terminates on the coprocessor as if the host processor were not even there. The "symmetric" mode is a natural extension of the "native" mode, where each computing device is defined in a heterogeneous cluster environment and treated equally. The exchange of information is realized through the standard communication mechanism such as MPI. In this thesis, we will mainly focus on the "native" and "symmetric" mode.

To sum up, a simplified way to view MIC from a programming standpoint is a x86-based cache-coherent SMP-on-a-chip with 61 cores and VPUs (Vector Processing Unit). But if we take a step further, the disturbance like NUMA effect begins to bother, which urges us to put more thoughts on it. More information about MIC is available in these references [73, 133, 137, 66, 19, 100, 57, 110, 121, 122].

3.4.2 Experimental Platform

Without particular indication, the experimental test platform for the most of the work in this study is two workstations. The first one has a dual-socket Intel Xeon E5-2670 Sandy Bridge

(SNB) host processor, and 4 cards of pre-production part of KNC codenamed "C0" installed with the latest Intel MIC Platform Software Stack (MPSS) [11]. The other one has a NVIDIA K20 GPU. The Table 3.2 lists the specific parameters of these computing devices.

Table 3.2 Commonly used experimental platforms in this thesis, including a Sandy Brdige workstation pairing with MIC coprocessors (KNC), and a GPU (K20) workstation. In chapter 5, we also utilize our lab cluster for a study of scalability. It is not listed here. For more details please see subsection 5.3.3.

Type	Specific
SNB	Dual-socket Intel Xeon E5-2670, 8×2 cores running at 2.6 GHz (Max Turbo Frequency up to 3.3 GHz) supporting up to 16×2 threads with hyperthreading. But the hyperthreading is turned off in practice for thread affinity reasons. The processor has 64 GB ECC supported DDR3 memory with a maximum bandwidth of 51.2 GB/s. There are 3 levels of cache: 32 KB×16 instruction L1 cache + 32 KB×16 data cache, 256 KB×16 unified L2 cache, and 20 MB per socket L3 cache
KNC	4 pre-production KNC prototype "C0" cards plugged in 4 PCIe slots of the workstation. Each KNC coprocessor has 61 cores, running at 1.2 GHz, sharing 16 GB GDDR5 memory with ECC enabled. The most recent update of MPSS is coming from version 3.1
GPU	NVIDIA Tesla K20 GPU (Kepler microarchitecture), 2496 CUDA cores, running at 0.71 GHz, 5 GB ECC-enabled GDDR5 memory [12] with a bandwidth up to 208 GB/s. It supports dynamic parallelism and HyperQ features

3.5 High Performance Programming Tactics

In general, the contemporary processor is better at executing instructions than loading data. So it is preferable to keep the data as close to the processor as possible. However, the closer to the processor, the more precious the space is. Only an intensive data reuse may justify the legitimacy of any data being close to the core by occupying the fast storage media (register, cache, scratchpad memory, "shared memory" in GPU terminology, etc.). Considering the memory hierarchies, the cache is usually the most important weighting factor for performance. However, unlike the shared memory in GPU, programmers do not have the direct control over the cache, the data reuse can merely be improved by promoting data locality. Two types of locality exist: temporal locality and spatial locality. As a majority of cache adopts LRU (see subsection 3.2.2) as their replacement policy, the temporal locality

[11] The MPSS is updated regularly.

underlines the length of the time slot a data element is being used. If the program somehow condenses the use of a data element into a short amount of time, then it has high temporal locality which also maximizes the cache efficiency. The spatial locality, on the other hand, starts from the fact that data are loaded from memory in terms of blocks (burst sections, see section 3.2.1), in lieu of single byte or word. The referenced memory location may suffer from high access latency, but the rest of the cache line is presented to the program nearly for free. An algorithm with high spatial locality knows how to fully exploit the contiguous memory locations while they are still hot in cache. The spatial locality applies also to the TLB cache. Same principle as before. Spatially adjacent elements are more likely to be on the same memory page, then a fast TLB address translation can be expected. Otherwise, having too many memory pages to reference will bring about TLB misses which could harm performance considerably. Changing to a larger page size can also be viewed as expanding the scope the "locality" refers to. More importantly in a multicore environment like MIC, the coherence of cache would have to take up the memory bandwidth when any core tries to modify the content that is kept in another core's cache. Since the memory bandwidth is much lower than the cache bandwidth, and prioritized to memory load, the congestion of memory bandwidth can only act adversely to performance. In fact, the notion of memory locality should always be serving the physically proximate zone, which is believed sharing the fast memory connection.

Common strategies to optimize the algorithm regarding the hardware are based on the preceding principles. Here we just name some techniques that are used in this study.

Pipelining Basically overlap the memory load and the computation, which help hide the memory latency. Concrete implementations involves loop unrolling, software/hardware prefetching, etc.

Cache blocking/Loop tiling Keep the size of working size containable within lower-level caches. Split the loop and rearrange the inner loop into blocks that fit into the size of cache. Also avoid stride whenever dealing with an array of elements

TLB blocking Prioritize the touch of elements within the memory page that has already been visited. Or allocate pages of larger size

Memory alignment The aligned address can be loaded into vector register within one instruction instead of two. Pad zeros at the tail if necessary

Streaming stores Allow to bypass the memory hierarchy by writing directly to the memory without polluting the cache, especially for non-temporal stores

Cache-oblivious algorithm In work-stealing runtime scenarios (see section 2.2), the algorithm typically take a recursive approach to divide the original work into smaller pieces and then "conquer" them later after them being processed. The individual divided task is supposed to be operated by a thread. If the size of task fits into the size of local cache, then it will have the same effect as cache blocking. Needless to effect explicit blocking, this algorithm does not take the size of cache (or the length of a cacheline) as an explicit parameter. To some extent, it is more portable and easier to tune. In most cases, the depth of recursion tree is the only parameter that matters.

The side effect of these optimizations include increased instruction volume, and error-prone code. All optimization decisions should be tuned with regard to individual cases. Besides, there exists no optimization technique that enables a program to run faster than the peak performance claimed by the manufacturer, actually not even close.

Chapter 4

Efficient Linear Algebraic Operations to Accelerate Iterative Methods

Krylov subspace methods [115, 114] have been extensively used for solving eigenvalue problems and linear systems. And they are not the only iterative methods that rely heavily on the matrix-vector multiplication. At each iteration, this very computing kernel tries to monopolize the computing resource and dominate the time spent on a particular solving process. As the most straightforward way to accelerate iterative methods is no other than parallelizing the consuming components, this chapter will focus on the optimization of the matrix-vector product kernel, for both dense and sparse matrices.

Note that all the experiments described in this chapter are performed particularly in 61-core KNC coprocessor (see section 3.4.1), but not exclusively. Other platforms (host CPU, GPU) are also used for comparison and validation purposes.

4.1 Numerical Context for Dense Matrix-Vector Multiplication

Two methods exist for simulating and modeling neutron transport and interactions in a nuclear system. Deterministic methods solve the Boltzmann transport equation in a numerically approximated manner everywhere throughout a modeled system. Monte Carlo methods model the nuclear system (almost) exactly and then solve the exact model statistically (also approximately) anywhere in the modeled system. Deterministic neutronics methods play a fundamental role in reactor core modeling and simulation. A first-principles treatment requires the solution of linearized Boltzmann transport equation. This task demands enormous computational resources because the problem has seven dimensions: three in space,

two in direction, and one each in energy and time. The Boltzmann transport equation can be considered as an eigenproblem [86]. The objective is to find the effective neutron multiplication factor, K, which is directly related to the dominant eigenvalue of the system. The power method [115] is the primary choice in such context to compute the dominant eigenvalue of the system. As a broadly used basic method, it has been implemented in the main deterministic neutronic code [1, 11].

Power method computes one [1] dominant eigenvalue λ along with the corresponding eigenvector v. The Algorithm 1 gives the principal steps for this iterative method. Let us consider A as a large non-hermitian matrix. Under mild assumptions, λ has the largest absolute value among A's eigenvalues when the power method is converged.

Algorithm 1 Power Method.

1: Choose a nonzero initial vector v_0
2: **for** i = start to end **do**
3: $v_{i-1} \leftarrow \frac{v_{i-1}}{\|v_{i-1}\|}$
4: $v_i \leftarrow Av_{i-1}$
5: $\lambda \leftarrow (v_i, v_{i-1})$
6: $r \leftarrow Av_{i-1} - \lambda v_{i-1}$
7: residual $\leftarrow \frac{\|r\|}{\lambda}$
8: **if** residual < tolerance **then**
9: Stop
10: **end if**
11: **end for**

The step 4 in Algorithm 1 iteratively generates a sequence of vectors $A^i v_0$ where v_0 is a nonzero initial vector. The repetitive matrix-vector multiplication is the most consuming part of the method. In terms of time complexity it is $2N^2 + O(N)$ [2] while the other operations are only $O(N)$. In fact, in our preliminary experiments with dense matrices, over 99% of the serial execution time is spent on the matrix-vector multiplication. This proportion shows us the importance of having an efficient kernel. The techniques (OpenMP, Cilk+, TBB for multithreading. Cilk+ C/C++ Array Notation, Intel simd pragma as explicit vectorization methods) discussed in chapter 2 will be put into use while implementing this kernel.

[1] There may be some eigenvalues that are clustered together with the dominant one.
[2] N is the dimension of the vector v.

4.2 Dense Matrix-Vector Product Kernel

At first glance, it is necessary to learn how much the computer is able to deliver. It is demonstrated in preprint paper [118] that the KNC bandwidth peaks at 183 GB/s for read when using 61 cores with 2 threads per core, and 160 GB/s for write also when all cores are used. The frequency of Intel Xeon Phi coprocessor is around 1 GHz. According to the subsection 3.4.1, the theoretical peak performance deduced for this machine in single precision, taking 1.09 GHz as its main frequency, would be 2.128 TFLOPS (1.09 GHz $\times$ 16 SP floats $\times$ 2 FMA ops $\times$ 61 cores) and halved in double precison which is 1.064 TFLOPS. However, for memory bandwidth bound problem like matrix-vector multiplication, the reality is nowhere close to those numbers. For every 8 bytes of double precison number loaded, there will be 2 floating-point operations (1 multiplication and 1 sum), which gives a flop-to-byte ratio of $1/4$. The sustainable peak performance then should be somewhere between 40 and 45 GFLOPS ($160 \times 1/4 = 40$ GFLOPS, $180 \times 1/4 = 45$ GFLOPS). The same arithmetic applies to Intel Xeon processors as well. The aggregate node level (dual-socket, i.e. 2 processors) bandwidth for Sandy Bridge in fully subscribed mode was 60.8 GB/s [116]. So the sustainable peak performance would be $60.8 \times 1/4 = 15.2$ GFLOPS. These limits will serve as guidance when optimizing the kernel.

4.2.1 Pure Multithreaded Solution

Let's first assume the matrices are stored in row-major order.

Algorithm 2 Serial multiplication of square matrix A ($n \times n$) and vector x.

```
1: for i = 1 to n do
2:     b_i = 0
3:     for j = 1 to n do
4:         b_i ← b_i + A_ij * x_j
5:     end for
6: end for
```

The Algorithm 2 shows an ordinary implementation of dense matrix-vector multiplication without using any optimization or parallel techniques. The two "`for`" loops indicates the work that needs to be repeated. If there is no data dependency, then they are perfect candidates for parallel execution. Fortunately, there exists only an addition race in inner loop, which can be overcome by a reduction. As mentioned in subsection 3.4.1, the KNC has two pipelines that support up to 2 instructions executed per cycle [3]. What if we merely throw scalar

[3] KNC only sustain a throughput rate of 1 vector instruction per cycle, also see subsection 3.4.1.

instructions at significant number of threads without touching the vector units? Is it enough to get close to the limits we calculated before?

The outer loop in Algorithm 2 iterates over the matrix rows. Each row corresponds to an independent task. Accordingly, the outer loops can be partitioned among the threads using:

OpenMP `#pragma omp parallel for`

Cilk+ `cilk_for` instead of normal `for` in C/C++

TBB Defines in C++ an `operator()` class that specifies the work to be done for a range. In our case, the inner loop should be defined in the `operator()`. The instance of this class will be passed into TBB's `parallel_for` interface, which is a template function defined for basic parallel algorithms like loop parallelization (along with `parallel_reduce` and `parallel_scan`)

Since only a pure multithreaded solution is intended here, we will leave the inner loop as it is. Or an alternative solution would be to perform a nested reduction operation hoping to exploit more parallelism. The reduction within different programming environments would be:

OpenMP `# pragma omp parallel for reduction(+:reducer)`

Cilk+ Declare a reducer of type "`cilk::reducer_opadd<ScalarType>`" for "ScalarType" numbers. Then add normally the values into reducer: `reducer+=value;`. The final reduced result is obtained through `reducer.get_value()`

TBB Declare in C++ a "reducer" class with `operator()` defining the work to be done for a range and a `join()` method specifying the reduction operation

We test both situations on KNC coprocessor, and also host processors [4] for comparison. The matrix used in all of the experiments, including those described in subsection 4.2.2, is a dense Lehmer matrix [96]. The Figure 4.1 shows the results obtained from all of three parallel techniques (OpenMP, Cilk+, and TBB) in host processors. The Figure 4.2, 4.3, 4.4 show the results obtained with each individual case.

From these graphs we know the nested reduction structure is not necessarily a good idea. It depends on factors like the number of threads, the nature of platform, etc. We can certainly work on identifying in each case the constraints and increasing the performance. But there seems to be no such need. In MIC, the performance is far below the expected

[4] Dual-socket Intel Xeon processors. See subsection 3.4.2

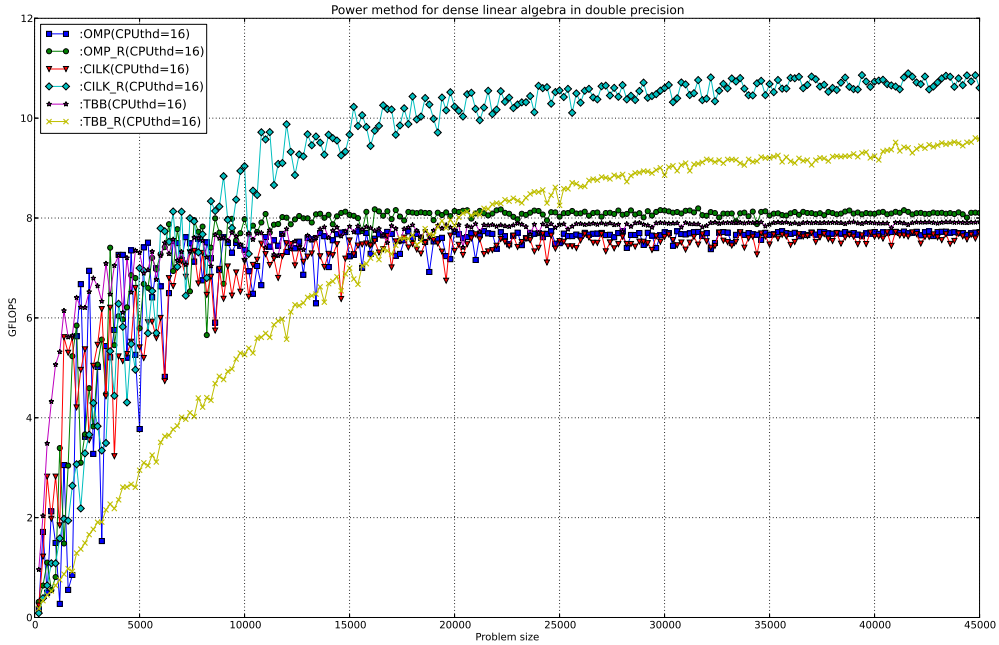


Figure 4.1 Pure multithreaded parallelization for dense matrix-vector product using respectively OpenMP, Cilk+, and TBB in dual-socket Intel Xeon E5-2670 processors with 16 threads (one per core). The "_R" suffix in the legend indicates the existence of a nested reduction structure for the inner loop of the dense matrix-vector product kernel.

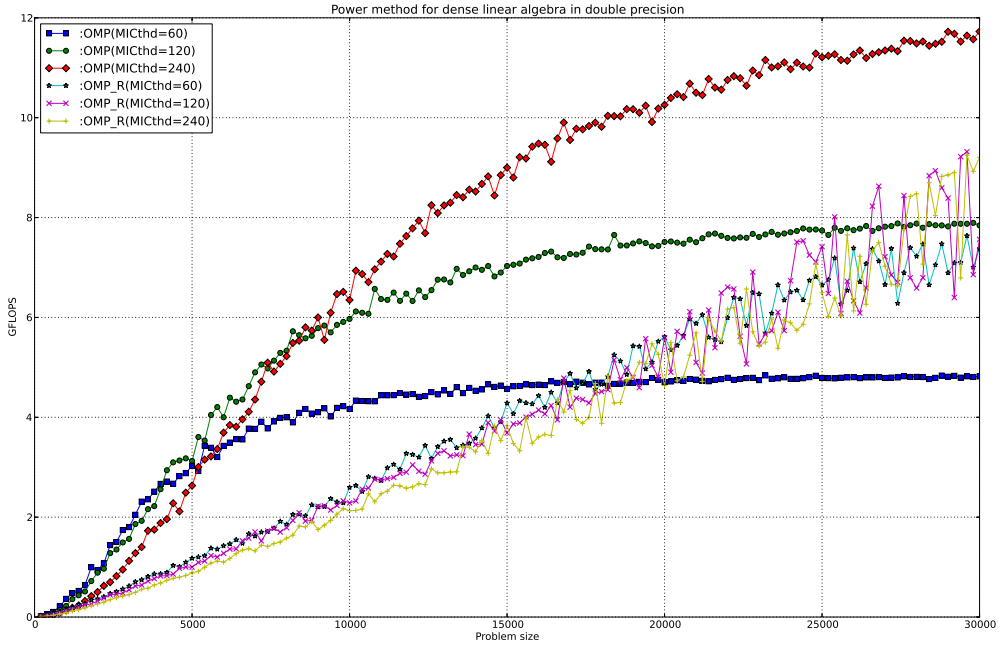


Figure 4.2 Pure multithreaded parallelization for dense matrix-vector product using OpenMP in KNC coprocessor with different number of threads (60: 1 thread/core, 120: 2 threads/core, 240: 4 threads/core). The "_R" suffix in the legend indicates the existence of a nested reduction structure for the inner loop of the dense matrix-vector product kernel.

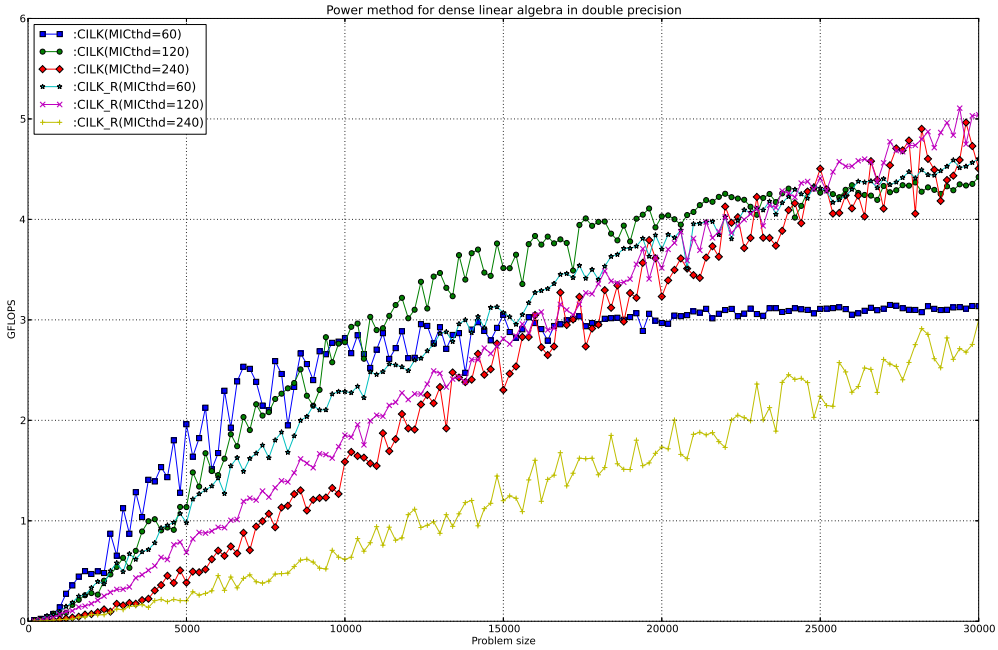


Figure 4.3 Pure multithreaded parallelization for dense matrix-vector product using Cilk+ in KNC coprocessor with different number of threads (60: 1 thread/core, 120: 2 threads/core, 240: 4 threads/core). The "_R" suffix in the legend indicates the existence of a nested reduction structure for the inner loop of the dense matrix-vector product kernel.

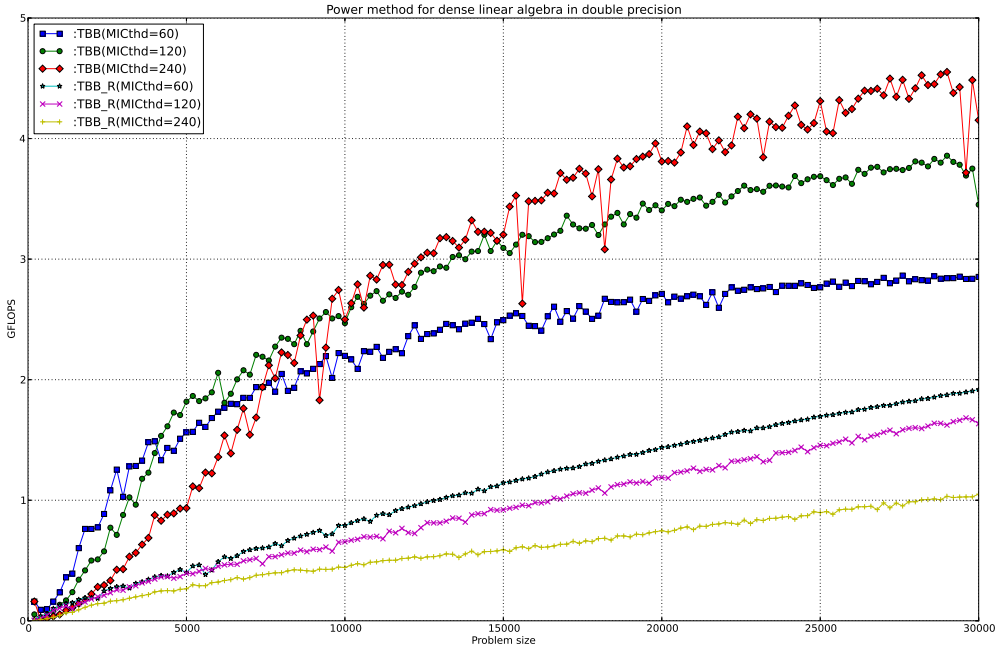


Figure 4.4 Pure multithreaded parallelization for dense matrix-vector product using TBB in KNC coprocessor with different number of threads (60: 1 thread/core, 120: 2 threads/core, 240: 4 threads/core). The "_R" suffix in the legend indicates the existence of a nested reduction structure for the inner loop of the dense matrix-vector product kernel.

height. Although in CPU we are close to what we want, we are still not so satisfied with the performance of small matrices. The point we are trying to make here is that the use of vector processing units is almost indispensable, especially for MIC [5], in order to achieve reasonable performance.

4.2.2 Combined Virtue of Multithreading and Vectorization

Based on what we have learnt from the last subsection, we must involve the vectorization in the parallel implementation of the dense matrix-vector product kernel. Since the vector processing unit is implemented within each core, the vectorization has to be a nested dimension under the multithreading. Therefore the problem needs to be partitioned into two related dimensions that are suited for these two forms of parallelization.

Basically there are two major ways to consider a matrix-vector multiplication [6]. Considering the matrix-vector multiplication as follows:

$$Ax = b \tag{4.1}$$

It can be computed as:

$$A_{i*} \cdot x = b_i, \text{ for } i = 0 : N \tag{4.2}$$

Where A_{i*} is a row vector [7] dot multiplied [8] with the left-hand side vector x.

Or thinking column-wisely, the b can be calculated as:

$$\sum_j x_j A_{*j} = b \tag{4.3}$$

Where x_j is a scalar value, i.e. the jth element of the left-hand side vector x, and A_{*j} is a column vector, i.e. the jth column of A. These two ways of expressing the matrix-vector multiplication offers two possibilities for assigning the dimension of vectorization. In Eq. (4.2), the product between the row vector of A and x is a perfect choice for SIMD operation. And so is the case with the column vector of A and a coefficient of x [9] in Eq. (4.3). In short, while one dimension of A is reserved for the vector operation (row

[5] The Intel Xeon processors also have vector units, but shorter than those in MIC. It supports SSE 2/3/4 (128-bit SIMD vectors) or AVX (256-bit SIMD vector). The IMCI supported in KNC is a 512-bit vector ISA. And the next generation KNL will support AVX-512. See subsection 3.4.1 for more details.

[6] The matrix-vector multiplication by blocks is excluded here. It will be left to later discussion.

[7] A_{i*} is the ith row of A, the "$*$" represents the presence of all elements.

[8] The dot product between two vectors can be cast into 2 vector instructions: a multiplication and a add reduction.

[9] Although x_j is only a scalar value, it can be broadcast into all of the SIMD lanes to perform the vector multiplication with the column vector of A.

dimension in Eq. (4.2) and column dimension in Eq. (4.3)), the other dimension is available for multithreading, which is in our favor, because there is no dependency between the threads, meaning it is free to try different thread affinity or scheduling strategies, except in Eq. (4.3) the ready column vectors [10] need to be summed/reduced into one. Unfortunately, there is no such mechanism that allows to efficiently deal with this situation. Let us go over two viable solutions. The first one is to perform threads reduction for each single element of b. The second is to do the tree reduction upon all threads. At each tree node, a vector addition is carried out between two intermediate vector results. It is imaginable that as the size of b grows larger, the time cost for the first option increases linearly. The second option may alleviate that effect, but it wastes at least half of vector processing units on chip. And both solutions require a large number of synchronizations, which is considerably harmful to performance. Although there is this upside for Eq. (4.3) that is superior to Eq. (4.2): the portion of x that is kept in a core is small and reused all the time, so it would stay in cache and not congest the bandwidth. But still, this virtue is not enough to counteract its flaw. So we choose to stay with the implementation guided by the Eq. (4.2). And the matrix is still stored in row-major order.

The Cilk+ C/C++ array notation and intel simd pragma have been selected to be the explicit vectorization methods. The Table 4.1 demonstrates the vectorized dot product expression in terms of Cilk+ array notation and Intel simd pragma.

Table 4.1 Explicitly Vectorized dot product using respectively Cilk+ array notation and Intel simd pragma.

Intel Cilk+ C/C++ Extensions for Array Notations	`b = __sec_reduce_add(A[N:2N]*x[0:N]);`

Intel specific simd pragma	`#pragma simd reduction(+:tmp)`
	`for(int i=0;i<N;i++)`
	`{`
	`  b+=A[N+i]*x[i];`
	`}`

With the help of these directives, the compiler is able to generate corresponding vector instructions. In fact, it is possible to add "`vectorlengthfor(ScalarType)`" at the end of simd pragma. This will further indicate the compiler to generate the vector instructions for a specific value (float, double, etc.).

[10] As temporary results, $x_j A_{*j}$ should be kept within each thread.

In our experiments, the rows of A are assigned to different threads using OpenMP, Cilk, and TBB [11]. And the dot product described in Eq. (4.2) is vectorized using techniques listed in Table 4.1. We tried all possible combinations. We also varied the number of threads, thread affinity, scheduling policy (for OpenMP), and task grain size. The Figure 4.5 shows the best results obtained in different architectures. In order to set a baseline, we also include the reference results obtained on GPU using cuBLAS [12], on CPU and MIC using Intel MKL [13].

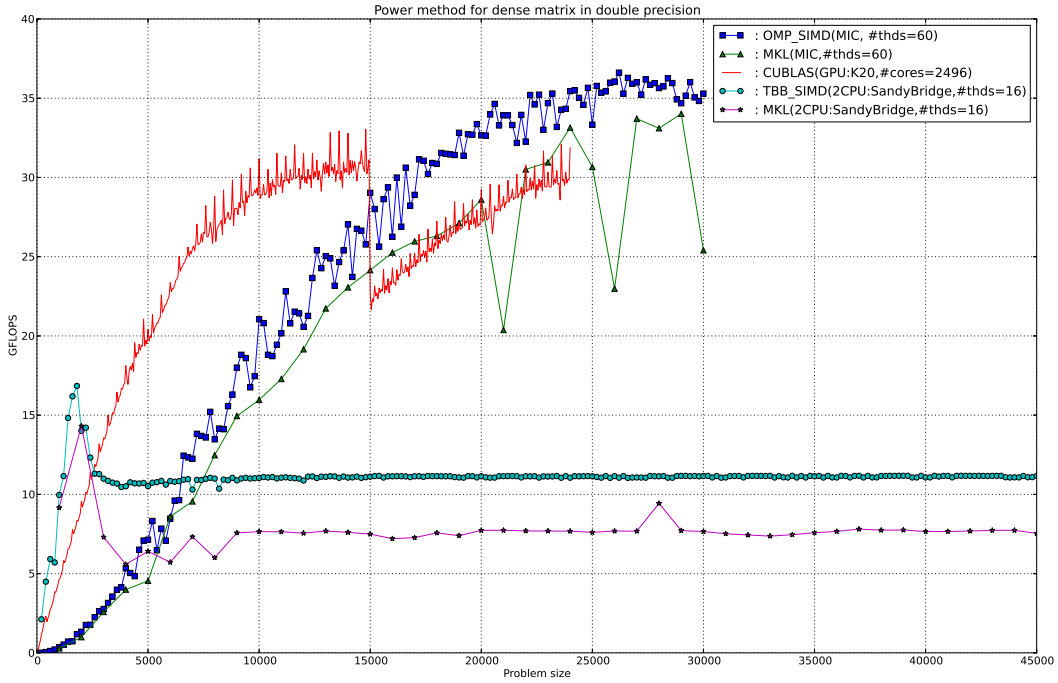


Figure 4.5 Best results obtained in different architectures. In MIC (KNC), the best approach is OpenMP combined with Intel simd pragma. While in the host processor (CPU) it is TBB combined with Intel simd pragma. The reference results are offered by control experiments executed on GPU using cuBLAS, on CPU and MIC using MKL. All the results are obtained with the Lehmer matrix [96].

As shown in Figure 4.5, the best performance on KNC is achieved with 60 OpenMP threads using "guided" scheduling strategy, each issuing vector instructions generated by Intel simd pragma. On CPU, the optimal solution is the combination of TBB and also Intel simd pragma. The difference between these two cases is that the former uses work-sharing runtime scheduler through OpenMP loop parallelism (`#pragma omp parallel for`) while

[11] For more detailed syntax, see subsection 4.2.1.

[12] The cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime.

[13] The MKL is short for "Math Kernel Library". Basically it is the BLAS library implemented by Intel.

the latter is based on work-stealing scheduling [14]. The downside of work-sharing schedulers is the contention for the public task queue. On the other hand, work-stealing is also not free meal. The stealing has a cost that is proportional to the distance between the thief thread and the victim thread. Considering the large on-chip network of KNC, although it is bidirectional, it does not change the fact that the ring interconnect is one-dimensional. Since the victim thread is randomly chosen, the on-chip transport incurred by stealing may affect quite a few cores along the way as it occupies part of the ring network. In such a bandwidth-bound problem, the bandwidth is precious. However, the contention problem of work-sharing scheduling should also be amplified by the number of cores on KNC. Why it performs better than work-stealing? There must be a way to neutralize the contention problem in OpenMP.

OpenMP provides 4 kinds of loop scheduling strategies, as listed in the below table.

Table 4.2 Four different loop scheduling types supported by OpenMP [102].

Type	Specific
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team

OpenMP also allows to specify the scheduling type to one of the three types ("static", "dynamic", "guided") during runtime through the environment variable "OMP_SCHEDULE". The "static" policy does not introduce extra scheduling overhead, because the loops are statically assigned. But it often has load imbalance issue. The "dynamic" and "guided" are more advanced in balancing the workload among the threads. In the case of KNC, we want

[14] It is important to note that from version 3.0, OpenMP also supports task parallelism. OpenMP task model is based on two activities: packaging and execution. The encounting thread adds task to pool, and other threads execute tasks in the pool. But nothing in OpenMP 3.0 specification requires the use of a work-stealing scheduler. For more information about work-sharing and work-stealing, see section 2.2.

to avoid the situation that a lot of cores contend for retrieving next chunk from the top of the work queue when they finish what's on hand.

The kernel that we are designing is dense matrix-vector multiplication. And we have demonstrated that it is more appropriate to assign the rows of matrix to threads. Since the matrix is dense, the workload for each row is quite similar. If assign equal-sized workload to each thread, as does in "dynamic" scheduling, it is imaginable that these threads finish their work at roughly the same pace and come back almost simultaneously for new assignment. This is exactly the contention that we are trying to avoid. However, the "guided" policy may considerably alleviate such contention while doing equally well in load balancing as "dynamic". That's why KNC achieves better performance with "guided" OpenMP. The host processor (Sandy Bridge), on the contrary, has a much smaller stealing cost thanks to its point-to-point interconnect [70] and narrower distance between cores. Therefore TBB obtains better performance on host processor while "guided" OpenMP suits KNC better.

It is noted that on KNC the best performance is achieved with 60 threads instead of more. In subsection 3.4.1 we mentioned that more threads can help to feed KNC with more instructions to execute thus hide pipeline and memory access latencies. It is still true, except that other factors may also come into play. Again, we want to stress that in this dense kernel, the memory access is regular and predictable. Using 60 threads means 1 thread per core, the local cache resource (L1 cache, local L2 cache, TLB cache) is exclusive to the only thread attached to the core [15]. It is easy for hardware to prefetch the following data. Assuming there are now 2 threads per core, each one of them must process different row chunks stored in disparate memory locations. Two threads will load their respective data by turns into the L1 and L2 cache. Hardware prefetcher may get confused and fail to correctly prefetch the data. Furthermore, visit distant memory locations will also burden the TLB cache, which may cause memory page faults. Taking these factors into consideration, using more than 1 thread per core may not always be the optimal setting for KNC. The performance with MKL, which peaks at 60 threads, also confirms this point.

Another interesting phenomenon observed in Figure 4.5 is that the GPU suffers a sharp performance drop at a matrix size of 15000×15000. The execution on GPU is carried out by the cuBLAS library for an architectural comparison. The whole power iteration is computed on GPU. The discontinuity is present for both single and double precision implementations. Therefore we conclude that it is rather a cuBLAS implementation artifact that may depend on the CUDA runtime decision or the use of shared memory.

[15] The thread affinity has been set so that OpenMP threads are bound to physical processing units and do not move away.

Let us go back to the intepretation of Figure 4.5. The KNC peaks at 38 GFLOPS, which attains almost 90% of sustainable peak performance (see section 4.2). It is also remarkable that a single KNC excels 2 (dual-socket) Sandy Bridge processors by a factor of 3 when they both reach their best. The host processors also achieve a very high percentage (94%) of maximum sustainable performance. Our implementations on both KNC and host processors outperform the manufacturer provided BLAS library, namely the Intel MKL. It shows that for similar dense applications, leveraging the exisiting code for many-core architecture is feasible even when it is difficult [16] to apply the manufacturer provided library. The use of OpenMP, TBB or Cilk, together with Cilk+ array notation or Intel simd pragma is relatively simple. It requires limited developing time, thus reduce the time-to-solution [39] (TTS, defined as the addition of developing time and execution time). Compared with MIC, GPU is also very promising in processing such parallel workload. As we can see in Figure 4.5, a comparably priced GPU has a better performance on first half of matrix sizes (the horizontal coordinate). It is because that the use of shared memory (comparable to cache in CPU) is explicit and direct.

At this point, we ask ourselves if we can do even better than what we had? The matrix-vector multiplication by blocks might be a path. Because it helps to keep small portion of x in cache and reuse it as much as possible. In the meanwhile, the reduction of column vectors could be less expensive under certain circumstances. But if we stay with the standard row-major matrix storage, we still have TLB and prefetching issues. In our experiments, the matrix-vector multiplication by blocks does not obtain results as good as that by rows. Since we go for a standard storage as stated in chapter 1, we have no intention to adapt the storage to hardwares.

4.3 Numerical Context for Sparse Matrix-Vector Multiplication

In real applications such as CFD, the matrices to be solved are usually unstructured and sparse. The dense matrix-vector product kernel is of no use here. We have to investigate the sparse matrix-vector multiplication.

Sometimes knowing simply the dominant eigenvalue is not enough for a scientific application. Sometimes knowing more eigenvalues will help to accelerate the solving of the problem. In both cases, the power method is no longer applied. To compute a subset

[16] For economic or availability reasons.

of eigenvalues, Krylov subspace methods [115, 114, 7, 43] are broadly used to solve non symmetric eigenvalue problems.

Among a collection of Krylov subspace methods, Arnoldi method is typical and representative for the discussion of this section. The Figure 4.6 illustrates the explicitly restarted arnoldi method (ERAM). Basically, it projects the matrix onto a subspace with substantially reduced dimension. The unitary projection of A is an upper Hessenberg matrix [65] that is expected to have the same eigenvalues as A when the method is converged [17].

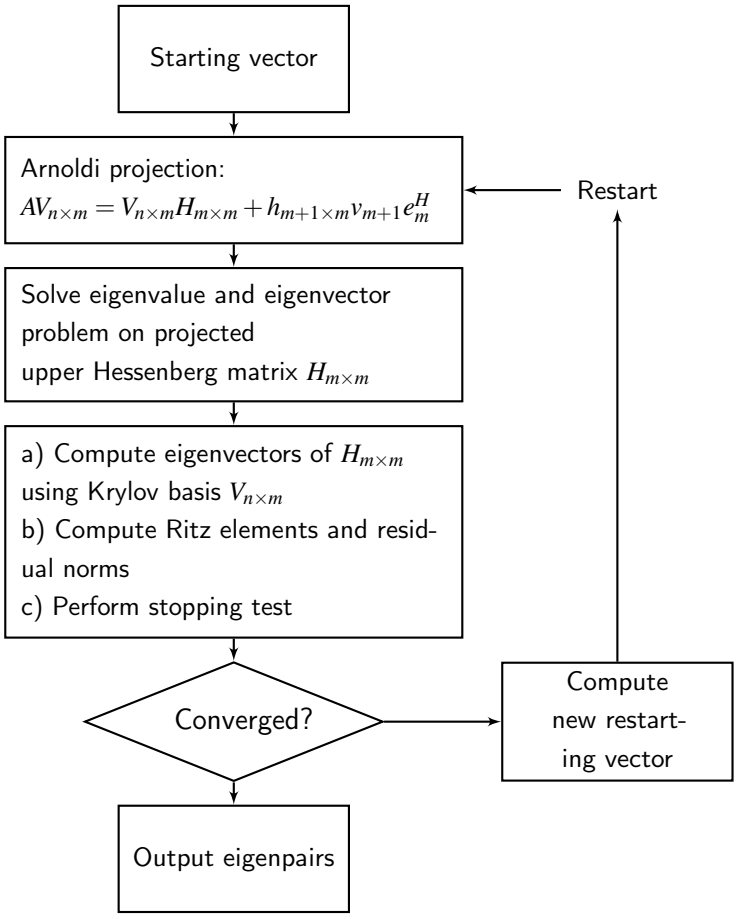


Figure 4.6 Block diagram illustrating the explicitly restarted Arnoldi method (ERAM).

4.4 Sparse Matrix-Vector Multiplication

In general, the most computationally expensive step in ERAM is the Arnoldi projection [17]. There are m iterations to perform to form the base $V_{n\times m}$. Under each iteration, depending

[17] It depends on the sparsity of matrix.

on which variant of Arnoldi process is used [18], there can be as much as 5 matrix-vector multiplications. Compared with dense matrix-vector multiplication, sparse matrix-vector multiplication (SpMV) is more complicated and hard to obtain good performance. It is the key to an efficient Arnoldi method, as well as many other iterative methods including conjugate gradient (CG) and generalized minimum residual (GMRES). More generally, the SpMV is fundamental to a broad spectrum of scientific and engineering applications. That is also the reason why SpMV always draws the attention of HPC community. Different approaches have been studied, including memory blocking (see section 3.5), vectorization, loop optimizations, software prefetching, auto-tuning, format variation, etc. [76, 140, 3, 147, 23, 106, 13, 38].

As in section 4.2, we want to figure out in what way the hardware progress, in terms of many-core architecture, can help us better implement the sparse matrix-vector multiplication. In fact, achieving a high percentage of peak performance on a given architecture for SpMV kernel has spurred the community to devote decades of studies. Due to irregularity nature of sparse matrices, the memory subsystem often appears as the main bottleneck of SpMV's effectiveness in terms of FLOPS. Furthermore, in a shared memory context with a large number of cores such as MIC, the scalability behavior is not so obvious which depends a lot on issues like data locality, data access pattern, and programming models. A common approach to address these problems is to propose a new sparse matrix storage format [84]. In general, a sparse matrix is stored by its nonzero values and additional indexing information of these values. The new storage format, usually derived from an existing format, rearrange the indexing and/or add redundant information in order to improve the data locality and reusability, thus reduce the memory bandwidth requirement. However, certain techniques used in those new formats may become less pertinent as the target architecture evolves. They may need to be adapted accordingly. Another potential downside of creating a new format links with the difficulty in implementing it in a large numerical package such as PETSc [9] or Trilinos [63]. Both PETSc and Trilinos adopt compressed sparse row (CSR) storage as the underlying sparse format. For institution like ours, being able to make use of these mainstream numerical packages is important. Even the external libraries are not used, most of the target applications uses classical CSR or CSC formats. Changing from one format to another just to speed up a numerical kernel will compromise the performance gain obtained from optimization. Therefore, we will use CSR format in this study to store the sparse matrices.

[18] There are several different orthogonalization processes that are applicable in Arnoldi method, including classical Gram-Schmidt (CGS), modified Gram-Schmidt (MGS), classical Gram-Schmidt with re-orthogonalization (CGSR), etc.

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 3 & 0 \\ 0 & 4 & 5 & 0 & 6 \\ 7 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 9 \end{pmatrix}$$

$$\begin{aligned} row\_ptrs &= [0,1,3,6,8,9] \\ col\_inds &= [0,1,3,1,2,4,0,3,4] \\ val &= [1,2,3,4,5,6,7,8,9] \end{aligned}$$

Figure 4.7 An illustrative example for compressed sparse row (CSR) sparse matrix storage format, which is comprised of 3 arrays: *row_ptrs*, *col_inds*, and *val*.

The Figure 4.7 illustrates how a zero-based CSR format keeps track of a 5×5 sparse matrix M. The *row_ptrs* array lists the indices indicating the beginning of each row in terms of their position in *val*. And it is ended by the number of nonzero elements in the matrix. The nonzero entries store their column indices in *col_inds*, and their values in *val*. The sparse matrix is actually linearized using CSR format without memorizing any zero.

Conventionally, hybrid programming, such as MPI+X [19], is more often to be seen in a distributed system where the computing node is regarded as the basic unit that provides the shared memory environment. And our previous study on dense matrix-vector product kernel [27] suggests that pure shared-memory parallel techniques are capable of obtaining good results on MIC. Moreover, some preceding studies [34, 30] hold pessimistic views about hybrid fashion compared to a unified MPI approach. These literatures usually underline the importance of network performance in explaining the gap between different models. However, the MIC architecture is supposed to offer promising on-chip bandwidth. It may benefit from using hybrid programming. Because intuitively, hybrid model should alleviate the scaling pressure by introducing an additional level of parallelism. The data "distributed" over processes can be exploited more locally than a pure multithreaded version. Here in the case of MIC architecture, we refer the hybrid execution to a scenario where the coprocessor resources (cores and caches) are divided into several separate domains and each domain is governed by one MPI process and shared by a number of OpenMP threads. As the non-cache memory of KNC is shared by 61 cores, there is no rule to partition "distributed" zones. To

[19] X can be any shared-memory parallel technique

solve this, we will deduce from the experimental results a performance model based on basic characteristics of test matrices as well as the number of processes and threads. This a posteriori model quantifies the performance issues and helps to predict approximately the behavior of the coprocessor. The deduced mathematical relationship should be instructive for SpMV optimization.

Same as before, we will use KNC as the principal test platform, and host processors along with GPU for architectural comparison.

4.4.1 Vectorized Kernel

For CSR, a natural way to parallelize the SpMV is to assign the subsets of rows to execution units (threads or processes). The elementary operation is then shrinked into the product of a compressed sparse vector with a dense vector. By using the SIMD instruction we insert at the lowest dimension a parallelism resulting from the vectorization. In this direction we propose the row-wise operational kernel for SpMV, which is similar to recent work on SpMV for MIC [84]. The Algorithm 3 delineates the SIMDized kernel that handles the

Algorithm 3 Row-wise vectorized kernel using CSR format (row_ptrs, col_inds, val). "*reg*_*" denotes vector graphic streaming SIMD extension register used by intrinsic functions.

```
 1: reg_y ← 0
 2: start ← row_ptrs[k]
 3: end ← row_ptrs[k+1]
 4: for i = start to end do
 5:     writemask ← (end-i)>8 ? 0xff : 0xff>>(8-end+i)
 6:     reg_ind ← load(writemask, &col_inds[i])
 7:     reg_val ← load(writemask, &val[i])
 8:     reg_x ← gather(writemask, reg_ind, x)
 9:     reg_y ← fmadd(reg_x, reg_val, reg_y, writemask)
10:     i = i + 8
11: end for
12: y[k] = reduce_add(reg_y)
```

row-wise multiplication. The *writemask* functions as a shifting window ensuring only the lower portion of vector being operated when there're less than 8 nonzeros left in a row. The "8" in Algorithm 3 implies 8 double precision floating numbers that occupy the 512-bits SIMD units in MIC. One have to notice that this version does not support symmetric matrix storage format.

4.4.2 Hierarchical Exploitation of Hardware Resources

The second dimension of parallelism is built upon the number of cores. Along with the hierarchical memory subsystem, these computational resources can be exploited by the execution units spawned and managed by the multiprocessing techniques. In most cases, it is easier to implement with a pure model than a hybrid one. In this paper we work on a spectrum where the two extremes are respectively the pure OpenMP and the flat MPI. The hybrid model lies in between. It nests OpenMP threads under MPI process. By combining these two models, we expect to promote the data access pattern and alleviate the scaling pressure occured in the pure model.

We define the SpMV process $y \leftarrow Ax$ as two phases:

1. The computing phase, where all the elements of y are being computed.

2. The communication phase, where y is copied to x.

The communication phase occurred usually in a iterative solver where the SpMV process needs to be repeated until the convergence of the solver. Because the x is occupied and shared by all of the threads, the communication phase of pure OpenMP can't be started before the termination of computing phase. However, with the participation of MPI, these two phases could be partially overlapped. In this case, we collect the computing phase timings that correspond to the slowest MPI process of each execution. These timing data will be used to deduce the performance of SpMV kernel. To obtain statistically meaningful results, we iterated 100 times for each measure of SpMV timing.

In terms of implementations, some conventional optimizations are applied to all of the 3 cases (OpenMP, hybrid MPI/OpenMP and MPI). They are loop unrolling, software prefetching, and streaming stores. In the presence of MPI, the rows of matrix are distributed in a way that each process MPI receives the same number of nonzero elements. The minimal unit of partioning is one row. In the first step we lack the means to forecast the combination of processes and threads that yields the best performance. We simply altered the number of processes and threads and attempted exhaustively all possible groups of processes and threads to seek the best configuration that maximizes the performance for each test matrix. It is noted that the test matrices were processed equally without considering their symmetry. To prevent the oversubscription of a certain area, the process/thread placement is considered. The MPI processes and OpenMP threads are properly bound to the physical processing units. More specifically, the Intel library provides additional environment variable to control the affinity. For threads, we set "`KMP_AFFINITY`" to "`granularity=thread,scatter`". For processes, according to the performance, we set "`I_MPI_PIN_DOMAIN`" to "`omp`" or "`auto`".

4.4.3 Matrix Suite

In practice, we have 3 principles in selecting the test matrices [37]. Firstly, we favor the matrices that have been used in previous literatures. Secondly, the matrices should have a larger volume in memory than 30 MB which is the aggregate L2 cache size of KNC, to neutralize the promotion in temporal locality induced by repeated runs of SpMV kernel. Last but not least, the matrices are selected to be square. The notion of eigenvalue only makes sense to square matrices. We also include a dense 8000×8000 matrix ("dense8000") expressed in CSR format. The basic characteristics of 18 selected matrices are outlined in Table 4.3.

Table 4.3 Table of sparse matrices that are used in performance test of different SpMV kernels.

Name	dim ($\times 10^3$)	nnz ($\times 10^6$)	nnz/dim
mixtank_new	29.957	1.995	66.597
mip1	66.463	10.353	155.768
rajat31	4690.002	20.316	4.332
nd6k	18.000	6.897	383.184
cage15	5154.859	99.199	19.244
crankseg_2	63.838	14.149	221.637
ns3Da	20.414	1.680	82.277
in-2004	1382.908	16.917	12.233
circuit5M	5558.326	59.524	10.709
sme3Db	29.067	2.081	71.595
ldoor	952.203	46.522	48.858
Si41Ge41H72	185.639	15.011	80.863
pdb1HYS	36.417	4.345	119.306
bone010	986.703	71.666	72.632
dense8000	8	64.000	8000
pwtk	217.918	11.634	53.389
torso1	116.158	8.517	73.318

4.4.4 OpenMP and MKL Performances

The pure multithreading programming model is a natural option for MIC architecture as it has been designed into a shared memory system. However, the streaming memory access pattern of SpMV makes the cores hard to run at full speed. Adding more threads helps to hide the memory stall due to cache miss. But the increase of virtual cores will lead to memory contention and network congestion, which degrades scaling performance. At core level, the

vector processing unit is more powerful than scalar pipelines. It is also true that it consumes much more data than scalar instructions. This fact puts more weight on memory subsystem. The load of data is less efficient for x than for *col_inds* and *val*, because only sparse locations of x will be accessed and the rest of the same memory request will be wasted. Problem would be more severe when 60 cores try to contend for different locations of the shared x.

A multithreaded SpMV kernel was implemented on KNC using OpenMP. The MKL version was also tested because it is based on the OpenMP runtime environment therefore comparable to our kernel. We measured the performances using from 1 to 4 threads per core. For each matrix we plot in Figure 4.8 and Figure 4.9 the bars of performance. From top to down different colors correspond to different threads configuration (1, 2, 3, or 4 threads per core). Performances are exhibited in a descending order. Lower part of a better performance bar is covered by a worse performance bar. All of the bars are actually started from the bottom, i.e. 0 GFLOPS. We show them in such way so that the worst performance bar is outmost. The readers can imagine that the worst performance bar is nearest to them.

From two figures we observe a similar trend for both implementations on different test matrices. None of them exhibits in average a better performance. We notice that MKL tends to have better performance when using more threads per core. We believe that it is because MKL has more precise control over threads.

4.4.5 Hybrid MPI/OpenMP Performances

In order to better deal with the issue of thread scaling and alleviate the memory contention, we propose to implement the hybrid MPI/OpenMP SpMV kernel. We expect to promote the efficiency of multithreading, scaling and cache utilization.

The experiments were conducted using all possible combinations of processes and threads with careful pinnings. The MPI processes are distributed evenly across 61 cores [20]. Each process accomodates the same number of threads as any other process. The total number of threads adds up to 60, 120, 180, or 240. All of the threads participate in computing vectorized SpMV kernel (see Algorithm 3), but only the main thread is responsible for the communication. The Algorithm 4 describes the main steps showing how the hybrid MPI/OpenMP model is implemented. In the 4th step of Algorithm 4, the collection of different portions of y can be done through MPI all-to-all communication, which is less efficient. An alternative solution is based on the use of MPI shared memory [50] that allows

[20] Though there is one core that is reserved for the opearting system and responding to hardware events.

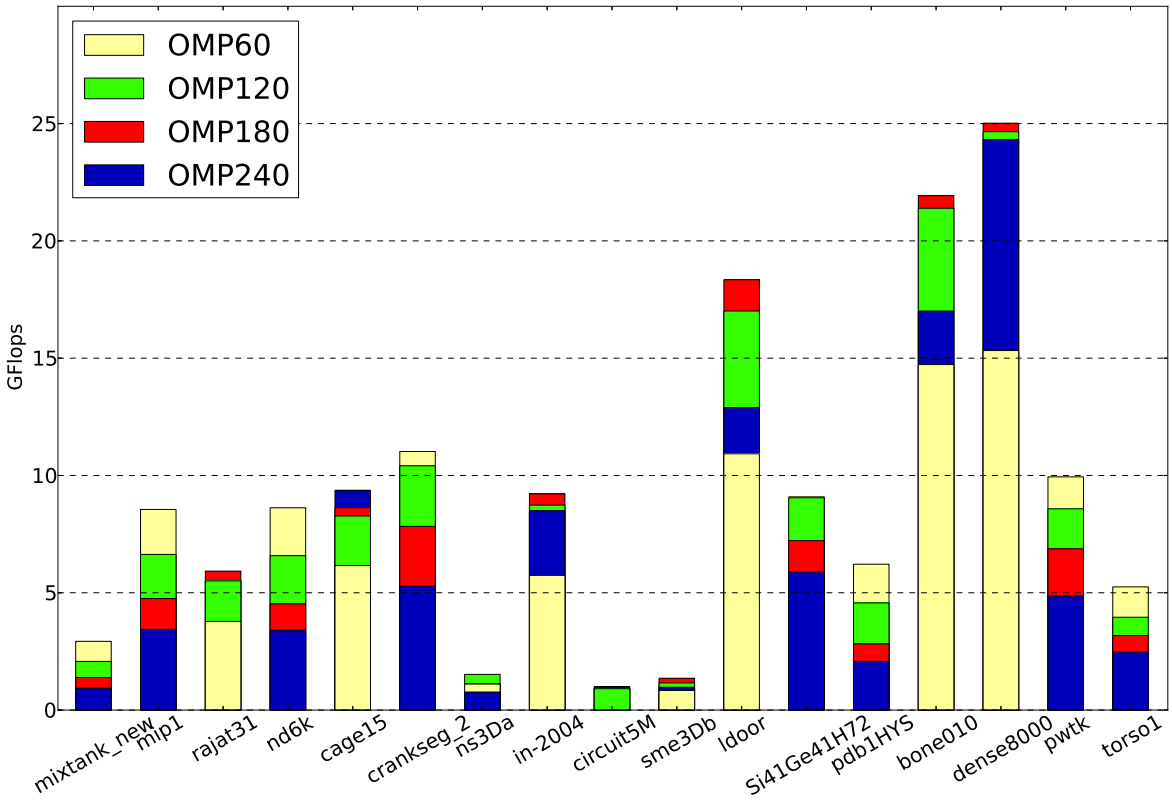


Figure 4.8 Performance bars of SpMV kernel implemented with OpenMP. The performance is measured in terms of GFLOPS. For every test matrix different number of threads (60, 120, 180, 240) is tested in order to find out the best fit [146].

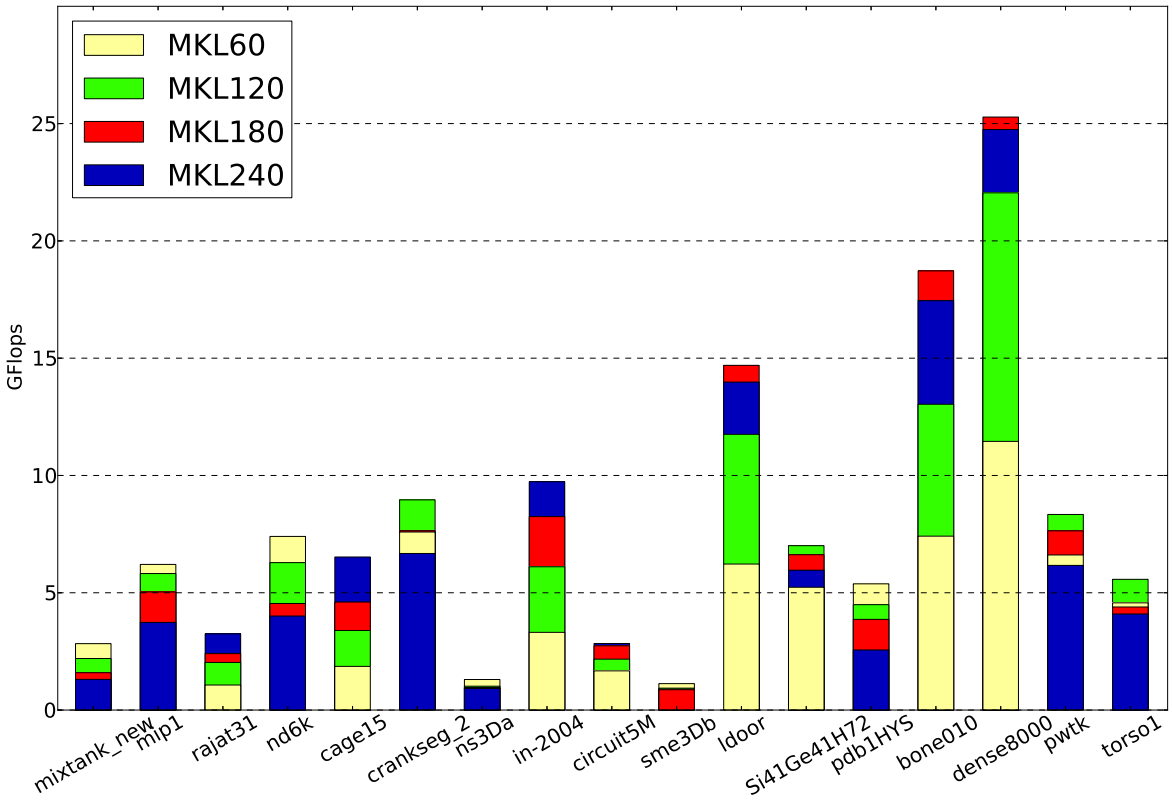


Figure 4.9 Performance bars of SpMV kernel implemented with MKL. The performance is measured in terms of GFLOPS. For every test matrix different number of threads (60, 120, 180, 240) is tested in order to find out the best fit [146].

direct load and store access between processes [21]. This method is more efficient than any MPI communication method.

The gain of hybrid model against pure OpenMP is shown in Figure 4.10. Over the entire matrix suite, the hybrid model exhibits a substantial performance improvement except in one case ("cage15"). The reasoning will be elaborated in section 4.4.8.

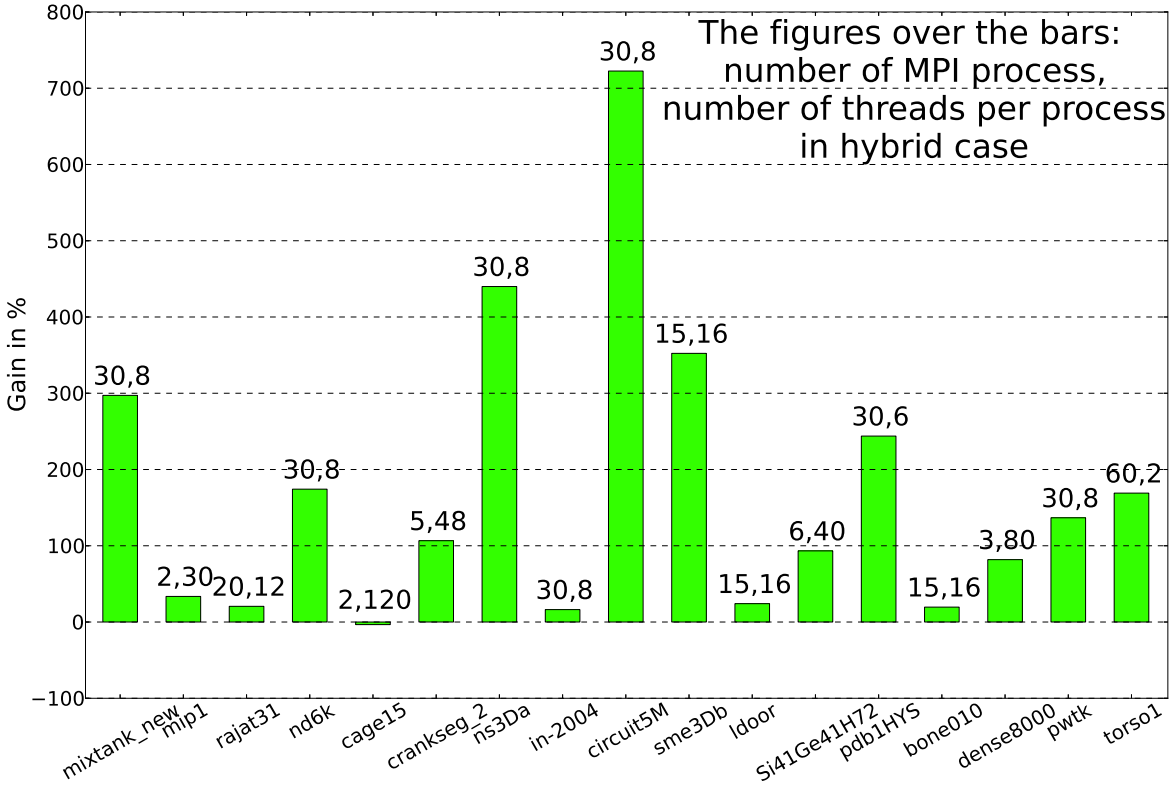


Figure 4.10 Gain in percentage of the hybrid MPI/OpenMP SpMV kernel against the pure OpenMP one. The presented results are the best ones for both SpMV kernels [146].

4.4.6 Flat MPI Performances

We have to point out that we also tested the flat MPI implementation. However, we can not start with more than 2 MPI processes per core due to memory constraints [22]. And the only results that we have are not very interesting. It is easy to imagine that using only 1 (or less

[21] Please be noted that the MPI-1 model has no shared memory concept. And MPI-2 has only a limited distributed shared memory concept. Be sure it is the right version that fully supports MPI shared memory programming.

[22] It is entirely possible to run more than 1 process per core. In fact, the KNC is designed to be able to switch between 4 contexts (see subsection 3.4.1) even though the context switching for a process is much more heavyweight than a thread. However, the use of memory should not exceed the physical limits. Otherwise the program would fail.

Algorithm 4 Hybrid MPI/OpenMP algorithm. Each MPI process accommodates the same number of OpenMP threads as any other MPI process.

1: Distribute row blocks $(rowptrs, colinds, val)$ of A so that each MPI process receives approximately the same number of nonzeros
2: Replicate x on all MPI processes, allocate y on all MPI processes
3: Apply locally the vectorized SpMV kernel (see Algorithm 3) with the threads managed by OpenMP
4: Gather the results from other MPI processes through explicit communication or shared memory zone, and update the local portion of y

than 1) process per core is a waste of computational resource, which will make the execution suffer from many memory stalls with nothing else to do. Therefore we conclude that flat MPI model is not favorable for the SpMV kernel.

4.4.7 Cross-Platform SpMV Performances

Finally, the performances of different SpMV kernels are presented here. The Figure 4.11 demonstrates the performances of hybrid model versus vendor-supplied BLAS libraries across a variety of architectures. In most cases, the hybrid model obtains better performances. Since the CSR format is used for all of the architectures, the results are not representative for the capability of GPU. Because the GPU may perform better with other sparse formats [95]. But this model is indicative and instructive for optimizing the MIC architecture. We note that in some cases the CPU achieves better performance than MIC. We will try to shed more light on it in the next subsection.

4.4.8 Performance Analysis

The experimental results reveal a considerable advantage of hybrid programming model over the pure ones. However, in real practice we do not have time to try the best combination of MPI processes and OpenMP threads. As a consequence, it is imperative to come up with a method to predict the behavior of the machine, or at least forecast the tendency of performance. To accomplish that, we need to learn more about the architecture and the main factors that affect the performance. We will discuss qualitatively the reasons of performance improvement and the primary performance issues. A mathematical relationship will be built based on the understanding. It helps to quantifies the effects of different factors. The effectiveness of the deduced model will be verified at the end of this section.

First thing to understand is the performance improvement because of the mixture of MPI and OpenMP. We argue that is mainly credited to the promotion of data locality and thread

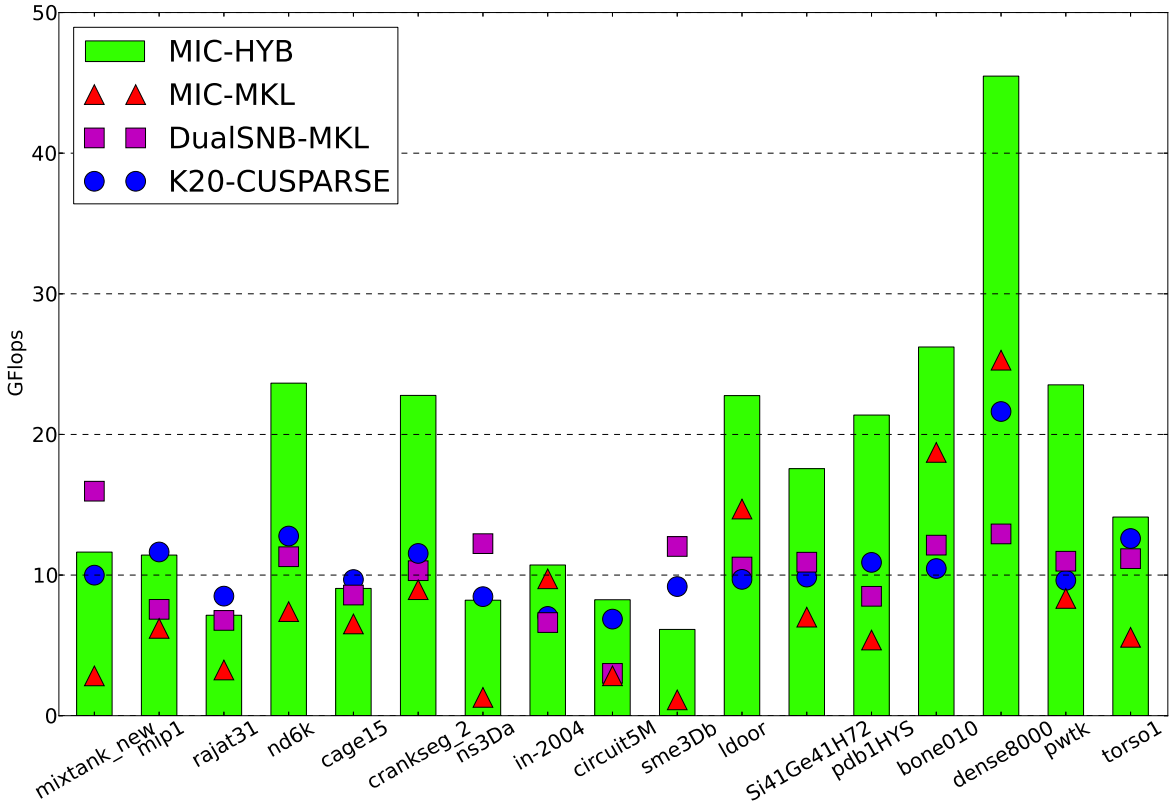


Figure 4.11 Cross-platform performances of different SpMV kernels. The "MIC" refers to the KNC coprocessor. The "HYB" refers to the hybrid MPI/OpenMP model. The "MKL" refers to the Intel Math Kernel Library. The "DualSNB" refers to dual-socket Intel Xeon Sandy Bridge E5-2670 processors. The "K20" refers to the NVIDIA TESLA K20 GPU. The "CUSPARSE" refers to the CUDA BLAS library for sparse matrices [146].

scalability. The promoted data locality improves the data reusability in terms of better cache utilization. It also mitigates the memory contention. Each process keeps a copy of vector x. The rows are distributed to each process based on the amount of work. Therefore these data are spatially local to the process domain. By carefully binding the processes to the physical cores, the data are stored uniformly in the memory space. It is then more likely to generate a higher aggregate bandwidth in the on-chip ring network.

In Figure 4.11, the hybrid performance of "dense8000" even exceeds the maximum sustainable performance indicated by the achievable bandwidth. In this case, the local x abates the memory contention. The software prefetching reduces the cache misses. While in the case of OpenMP, data are shared by all of the threads. They are hardly local to any thread. If we want to make them local, we need to make the data private to all the threads by replicating them. But in that way the replication is too much and will depend on the number of threads, which harms the scalability of the solution.

The scaling factor is relatively easy to understand. In a huge many-core system, the thread scheduling overhead caused by the contention for the central work queue is not a linear function of the number of threads. When the threads grow in number, the thread pool is hard to manage. However, by using the hybrid model, each process is responsable for a small number of threads which makes it relatively easy to scale.

The hybrid model shows other potential advantages as well. It is straightforward to implement it in a numerical software environment such as Trilinos, where the underlying MPI/OpenMP modules are already encapsulated and ready to use.

Despite the overall gain of performance, the hybrid SpMV kernel still performs poorly in some cases compared with other implementations. The poor performance is likely to occur in matrices that have low average number of nonzeros per row [23], such as "rajat31", "cage15", "in-2004", "circuit5M" in Table 4.3. When a matrice lacks nonzero elements, it is hard for vector processing units to be efficient. For one thing, the fraction of useful SIMD slots is low, many are wasted. For another, the memory load latency and vector instruction overhead, such as start-up or writemask overheads, can not be amortized over continuous memory loads and vector instructions, simply because the rows are too short.

This is not the only cause for bad performance. For matrices "ns3Da" and "sme3Db", their performances are not promising in spite of having decent average number of nonzeros per row. The former reasoning does not apply to this case. And the SIMD efficiency should just be fine. We checked the *spy* plot [24] of these two matrices. It shows that the nonzero elements are

[23] The average number of nonzeros per row is defined as the quotient of the total number of nonzeros over the number of rows.

[24] The *spy* is a command in many numerical environments such as Matlab that visualizes the nonzero element distribution of a matrix.

uniformly spreaded over the rows, which means in order to load the corresponding elements of x, irregular locations need to be visited. In the Algorithm 3, we use the `gather` instruction (see step 8) for collecting contents of x. In fact, the hardware gather is not particularly efficient on KNC, especially when it involves distant and very irregular locations.

Besides these two factors, there is this common issue that is present in many parallel problems. In our case, it is the load balance of MPI. Since we use a static row partitioning policy, the execution time of each MPI process can not be dynamically controlled. Sometimes it is fine when the nonzeros are evenly distributed. Sometimes it is not because of some well-filled rows. We do not split a row but assign it as a whole to a process. In future studies, we will develop a dynamic partitioning strategy for a better load balance.

The previously summarized factors are indepedent performance issues. Therefore they can be good performance indicators. We will try to develop a performance model by using these indicators in next subsection.

4.4.9 Performance Modeling

Before the discussion, we would like to define some variables that help to develop the performance model.

Definition 1. *For a given matrix, let the nnz be the number of nonzero elements involved in a subset of rows. If t is the execution time of SpMV computing phase defined in Section 4.4.2, then the performance P of the SpMV kernel is*

$$P = \frac{2\ nnz}{t_{max}}$$

[25] *where t_{max} is the execution time of the slowest MPI process.*

If nnz_{glob} is the total number of nonzero elements of a matrix, then the global performance P_{glob} is computed as

$$P_{glob} = \frac{2\ nnz_{glob}}{t_{max}}$$

In order to properly model the performance, it is necessary to separate the load balance factor from the vectorization rate and irregularity of nonzero elements. Since the t_{max} is the execution time of the slowest MPI process, it is more appropriate to find out the local performance P_{local} which correponds to t_{max}.

$$P_{local} = \frac{2\ nnz_{local}}{t_{max}} = \frac{nnz_{local}}{nnz_{glob}} P_{glob}$$

[25] For each nonzero element, there are two operations to perform: the multiplication and addition.

where nnz_{local} is the number of nonzero elements of the row block assigned to the slowest process. We measured the execution time of the slowest process and noted its rank. The rank helps to identify the row block that is associated with a particular process. As the thread is the basic unit that performs vector instructions, the per-thread performance is meaningful for characterizing the indicators discussed in the last subsection.

Definition 2. *if P is the aggregate performance of n_{thd} number of threads, then the per-thread performance is estimated as*

$$P_{thd} = \frac{P}{n_{thd}}$$

Assume the n_{thd} is the number of threads spawned within the slowest process, then its local per-thread performance is

$$P_{thd} = \frac{P_{local}}{n_{thd}}$$

We think of the average number of nonzeros per row as the first indicator to quantify the SIMD efficiency. Three features are listed in below helping to set up the functional relationship between this indicator and the per-thread performance.

1. If the number of nonzeros equals 0, The performance should also be 0. However, as the average number of nonzeros per row increases, the influence of memory latency and vector overhead should be amortized.

2. The greater the average number of nonzeros per row is, the less amplification the performance gains.

3. The performance should have an upper bound as the number of nonzeros per row is sufficiently large.

According to these features, the relationship should look like a concave increasing curve with a horizontal asymptote as shown in Figure 4.12 by the blue curve with circle markers.

There should also be a second indicator that models the dispersion of nonzero elements. With more nonzero elements in a row there is usually better performance, if given a fixed level of nonzero dispersion. This second indicator should be able to describe the "level of dispersion". The solution by instinct is to measure the cache misses. However, the behavior of cache in modern computer depends on, including but not limited to, cache capacity, cache associativity, cache line width, cache levels, replacement policy. It is unlikely to describe it with a low-cost model. We want a convenient and practical indicator. It turns out that the distance between each pair of contiguous nonzero elements in a row is enough to characterize the dispersion. In our model, we use the average number of occurrences when such distance

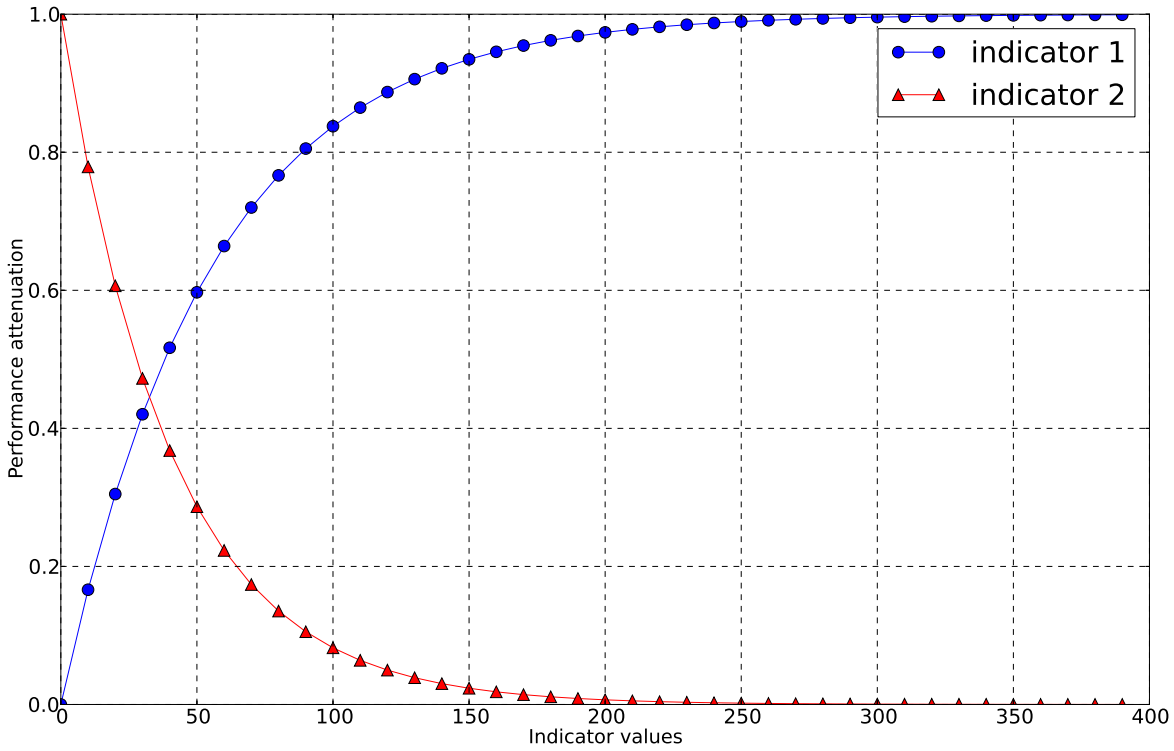


Figure 4.12 Relationship between two indicators and performance. The first indicator is the average number of nonzero elements per row. The second indicator is the average number of occurrences when the distance between any pair of contiguous nonzero elements within a row is greater than 2.

is greater than 2 as the second indicator. Similar to the first indicator, it takes the average over all of the rows.

The red convex decreasing curve with triangle markers in Figure 4.12 depicts the attenuation that the second indicator should cause to the performance.

With these two indicators, we give the exponential-based formula in Eq. (4.4), where $\hat{P}_{thd}$ is the estimation for the per-thread performance, $\overline{nnz}$ is the first indicator, and the $\overline{d}$ is the second indicator.

$$\hat{P}_{thd}(\overline{nnz},\overline{d}) = \alpha\left[1-\exp\left(-\frac{\overline{nnz}}{\varepsilon_1}\right)\right]\exp\left(-\frac{\overline{d}}{\varepsilon_2}\right) \tag{4.4}$$

The experimental data are collected from the slowest MPI process of each run using matrices listed in Table 4.3. These data are used to train the coefficients $(\alpha,\varepsilon_1,\varepsilon_2)$ in regression model given in (4.4).

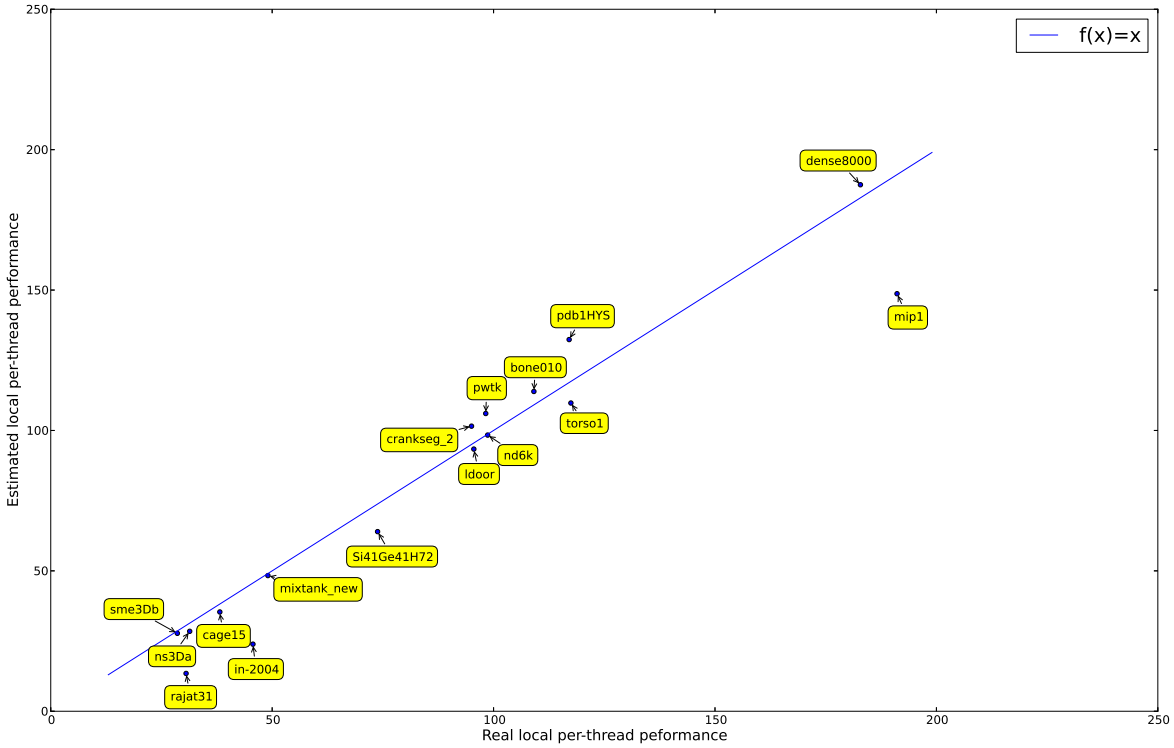


Figure 4.13 Scatter points plot of the real and estimated local per-thread SpMV performances. Each point represents a test matrix. The more a point is close to the line $f(x)=x$, the more accurate the estimated performance is.

The Figure 4.13 plots the scatter points of the real and estimated local per-thread performances over a set of test matrices. The real performance is measured from the slowest MPI

process. The trained coefficients are:

$$\widehat{\alpha} = 187.5,\ \widehat{\varepsilon_1} = 55,\ \widehat{\varepsilon_2} = 40$$

where the $\widehat{\alpha}$ corresponds to the per-thread performance (in MFLOPS) of "dense8000". Considering its sufficiently large average number of nonzero elements per row and regular memory access, the execution of this matrix is deemed optimal.

If we use this model to compute global performance with very little information about the matrices [26], we obtain Figure 4.14. As we can see from it, the estimated results are quite accurate, which confirms the validity of our model.

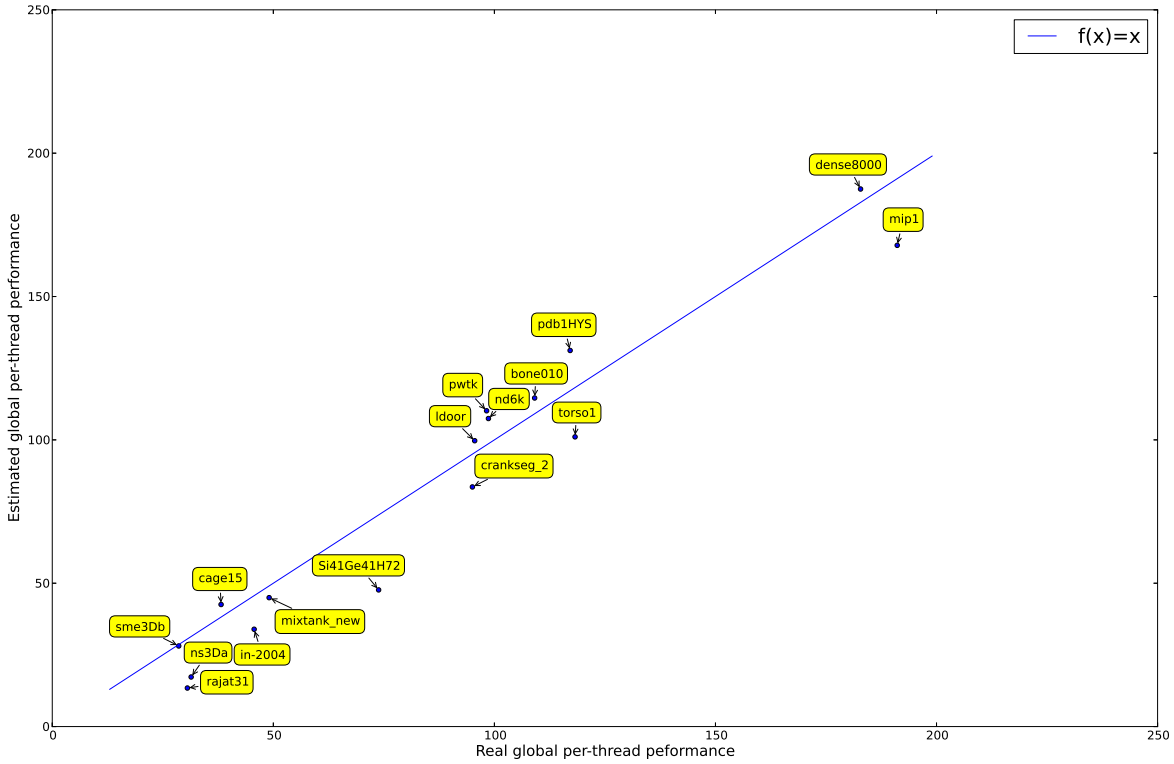


Figure 4.14 Scatter points plot of the real and estimated global per-thread SpMV performances. Each point represents a test matrix. The more a point is close to the line $f(x) = x$, the more accurate the estimated performance is.

At last, we have to mention the fact that we exclude the results for "circuit5M" in Figure 4.13 and Figure 4.14. This matrix has a very unbalanced nonzero distribution. Almost all the time the slowest process is the one that gets only a small number of rows but with a large number of nonzero elements. It makes the standard deviation of nnz σ_{nnz} much greater

[26] We use P_{glob}, nnz_{glob} for each matrix. And we simply use the first row of the row block assigned to the MPI process to compute $\bar{d}$.

than the average $\overline{nnz}$. So the estimation for $\overline{d}$ is not accurate. As a result, the estimation for performance is also not representative so we eliminate it from the presented results.

4.5 Towards next generation of MIC architecture

As the future generations of MIC make their debut, it is possible to extend the SpMV model proposed in this chapter into two dimensions. The first dimension is built on the new MIC processor.

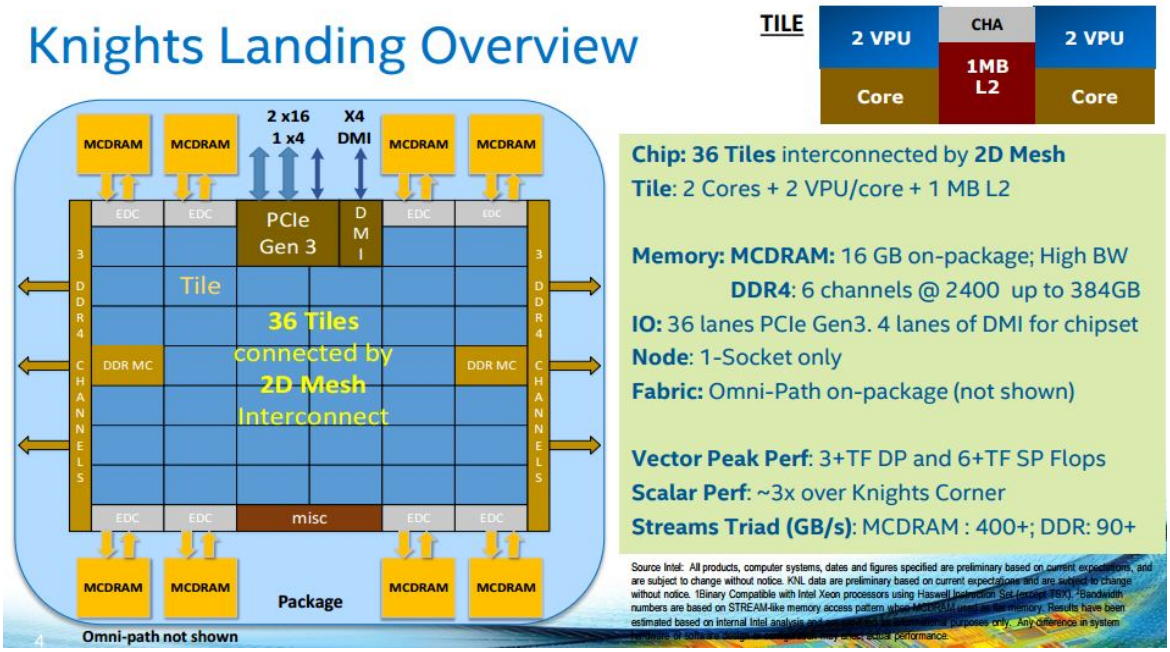


Figure 4.15 The Knights Landing Overview released by Intel.

The Figure 4.15 unveils some architectural highlights of the next generation of MIC architecture: Knights Landing (KNL). Instead of implementing 61 cores uniformly distributed over a ring network, Intel decides to create a notion of "tile" that is placed in a 2D mesh interconnect. A "tile" is the basic unit of on-chip communication. It consists of 2 cores that share a 1 MB L2 cache. This design will not abolish our SpMV kernel. Quite contrarily, it enhances the validity of our model because we have only "virtual zones" before, but there are now tangible physical zones named "tile". The local cache is still the keypoint of optimization. Since the L2 is always coherent, which makes more aggregate L2 cache size with 72 cores. But it can be imagined that the maintenance of coherency is still costly in terms of interconnect bandwidth. The hybrid process/thread model allows to keep only the efficient local "zones" and cut the connection to the rest. The NUMA effect should still exist. But

the improved memory access can be expected thanks to the bidimensional interconnection. Furthermore, KNL offers more flexibility on memory allocation. The fast near memory MCDRAM (Multi-channel DRAM) is configurable. It can play two roles: addressable and allocatable memory or an additional level of cache. Such flexibility will empower the programmer with better control over the memory [27]. Finally, Intel strengthens the vector processing units (VPU). Each core now has 2 VPUs and uses standardized AVX-512. They will continue to serve as the main computing engine for the KNL. And undoubtedly it puts forward higher requirements for parallel algorithms that support vectorized operations, such as what we proposed for the Monte Carlo solver (see chapter 5).

The second dimension extends to the cluster level. In a virtual perspective, only more processes need to be created. But the SpMV model proposed in this chapter may become less pertinent because the factors like intra/inter node communication, load balance, hierarchical data locality, etc. will come into play. The model should be revised in order to go on serving in a larger context.

4.6 Conclusion

This chapter studies two basic linear algebraic operations that are fundamental to the scientific computing.

The first part is about dense matrix-vector multiplication. Because of its regular memory access pattern and balanced workloads, the emerging many-core architecture is capable of achieving much higher performance than the traditional multi-core processor. We studied the use of different multithreading techniques and explicit vectorization methods on MIC. We also discussed the key points to obtain good performance and compared different scheduling strategies. Based on what we learnt from the experiments and analysis, we propose a combined model for this task that achieves better performance than vendor-supplied library.

The second part is about sparse matrix-vector multiplication. The SpMV is the key kernel that constitutes the main process in many iterative numerical methods. The emerging many-core architecture is capable of delivering higher performance. But it also put forward higher requirements for programming. In this chapter, we investigate three programming models: the pure OpenMP, the hybrid MPI/OpenMP, and flat MPI. Starting from a vectorized CSR SpMV kernel, we propose different ways to parallelize it. A set of performance evaluation is conducted on a variety of mainstream architectures by using not only our implementations but also the vendor-supplied BLAS libraries. The results suggest that the hybrid MPI/OpenMP model is very promising on Intel MIC architecture. It can help to reduce the scaling overhead

[27] It also reminds us of the shared memory in NVIDIA GPU.

and promote data locality compared with pure models, therefore improving substantially the performance. It is also straightforward to implement hybrid MPI/OpenMP in numerical software environments such as Trilinos, where the underlying MPI/OpenMP modules are already encapsulated and ready to use.

In order to better understand the performances of hybrid model, we have identified 3 performance indicators, namely the average number of nonzero elements, the average number of occurrences when the distance between any two contiguous nonzeros within a row is greater than 2, along with the load balancing. We studied the impacts of the first two indicators within the last terminated process and develop a performance model based on the experimental data. We also estimated the regression coefficients. The deduced model succeeds to predict the performance using very limited information about the matrix. This model is instructive for not only SpMV optimization, but also other similar applications.

Chapter 5

Algorithmic Improvement of Monte-Carlo Linear Solver

This chapter is about speeding up numerical methods at the algorithmic level. It takes some effort to modify a method in order to create parallelism that is friendly to the hardware. The design of numerical methods usually follows the mathematical outline, which is not necessarily the best for the implementation. However, some insights into the algorithm will allow us to reform and leverage the existing method by making it more parallel.

The numerical context of chapter 4 was the solution of eigenvalue problems. In this chapter, we will focus on the solution of linear systems. In lieu of optimizing the critical component of a scalar method, why not start with a method that exposes already abundant parallelism in the first place? Such method should be a natural fit for the many-core architecture. The Monte Carlo method meets the criteria. Moreover, this method also plays an important role in many scientific and engineering simulations. The contribution that we make to it can be applicable or instructive in other contexts.

5.1 Numerical Context

Solving linear system is one of the fundamental problems in computational science and is essential to a broad class of scientific and engineering disciplines. In reactor physics such as thermal-hydraulics, neutronics, structural mechanics, etc., a system of linear equations is often the arrival point of the description of the physical system. In particular, when solving the Navier-Stokes equations there are linear systems to be solved for each iteration.

The main subject of this thesis is many-core architecture, where basically latency is traded for throughput. In the last chapter, our approach is to provide efficient kernels for

numerical methods. However, common numerical methods often have an approach that tries to solve the problem step by step. In other words, they can not detach themselves from their intrinsic sequential nature. For example, iterative methods still need periodic synchronization points, which is unfavorable for scalability. We will need more than just parallel kernels.

In this light, Monte Carlo technique is a good candidate for an "inherently parallel" method [72]. Monte Carlo methods are a wide range of computational algorithms which depend on repeated random sampling to obtain numerical results. They are of great interest in parallel computing because the samplings are very often independent of one another; therefore, the sequences of computation are independent, which exposes potential parallelism. Such parallelism is well suited for modern processors with a large number of cores.

The use of Monte Carlo technique in linear algebra can date back to the work of von Neumann and Ulam in 1950s [49]. It performs stochastic matrix vector multiplication for solving the linear system [123, 112]. Monte Carlo solvers can not compete with classical iterative methods in numerical convergence. Their convergence typically stagnates at an order of magnitude higher than those of classical iterative methods. But it is more than eligible for computing approximate solution, which is adequate for many scenarios, such as preconditioning, graph partitioning, information retrieval, feature extraction, etc. The theory of Monte Carlo linear solvers is relatively mature except with regards to its stochastic nature: the result of a Monte Carlo solver is not reproducible. This issue will be addressed in later section (5.5) of this chapter. We will modify the method in a way that the solving process becomes deterministic.

In this chapter, we will be first focusing on the conventional and scalable implementation of Monte Carlo solver. Due to several crucial performance bottlenecks of the original Monte Carlo implementation, we propose a modified algorithm that completely overcomes those issues. The improved Monte Carlo solver is easy to implement using the task model; as a result, it becomes more friendly to multi-core and many-core systems. At the end of this chapter, we propose to turn the Monte Carlo method into a preconditioner so that its weaknesses in convergence will no longer be a problem.

5.2 Mathematical Prerequisite

Consider a linear system

$$Ax = b \tag{5.1}$$

where $A \in \Re^{n \times n}$ and $x, b \in \Re^n$. We split A as

$$A = B_1 - B_2 \tag{5.2}$$

where B_1 is taken to be the diagonal of A. Let

$$C = B_1^{-1} B_2 \tag{5.3}$$

$$h = B_1^{-1} b \tag{5.4}$$

Then we have $x = A^{-1}b = (B_1 - B_2)^{-1} B_1 h = (I - B_1^{-1} B_2)^{-1} h = (I - C)^{-1} h$, which leads to the following Equation (5.5):

$$x = Cx + h \tag{5.5}$$

Only under the Neumann series convergence condition $\max\limits_{1 \leqslant i \leqslant n} \sum\limits_{j} \left| C_{ij} \right| < 1$ can Equation (5.5) be solved iteratively as described in Equation (5.6). So the linear system to be solved, namely A, has to be strictly diagonally dominant.

$$x_{k+1} = Cx_k + h \tag{5.6}$$

Suppose $x_0 = h$ and $C^0 = I$, then

$$x_m = C^m x_0 + \sum_{j=0}^{m-1} C^j h = \sum_{j=0}^{m} C^j h \tag{5.7}$$

By constructing a Markov chain [18] of length j, the Monte-Carlo linear algebra techniques can estimate $C^j h$, for $j \geqslant 0$. As $x_0 = h$, the initial state determined by the initial probability vector p can be defined in Equation (5.8) as:

$$h_i = p_i \times w_i, 1 \leqslant i \leqslant n \tag{5.8}$$

with

$$\sum_{i=1}^{n} p_i = 1 \tag{5.9}$$

The w in Equation (5.8) is the "initial weight". The transition matrix of this Markov chain is denoted by P. Similarly there is corresponding "weight" matrix W defined in Equation (5.10):

$$C_{ij} = P_{ij} \times W_{ij}, 1 \leqslant i, j \leqslant n \tag{5.10}$$

with

$$\sum_{i=1}^{n} P_{ij} = 1, 1 \leqslant j \leqslant n \tag{5.11}$$

There are two common choices for defining probabilities if we do not consider the absorbing states: they are the uniform (UM) and the almost optimal (MAO) methods. The MAO method

normalizes every entry in a given vector by dividing them by the sum of the entire vector. The initial probability vector, the initial weight, the transition matrix and the weight matrix can be written as in Equation (5.12) and Equation (5.13):

$$p_i = \frac{|h_i|}{\sum_{j=1}^{n} |h_j|}, w_i = sign(h_i) \times \sum_{i=1}^{n} |h_i| \tag{5.12}$$

$$P_{ij} = \frac{|C_{ij}|}{\sum_{k=1}^{n} |C_{kj}|}, W_{ij} = sign(C_{ij}) \times \sum_{k=1}^{n} |C_{kj}| \tag{5.13}$$

The UM method, as the name suggests, makes all probabilities equal: so we have Equation (5.14) and Equation (5.15):

$$p_i = \frac{1}{n}, w_i = nh_i \tag{5.14}$$

$$P_{ij} = \frac{1}{n}, W_{ij} = nC_{ij} \tag{5.15}$$

Compared with the UM method, the MAO method requires less and shorter chains to reach the same given precision [4]; in other words it guarantees a better convergence rate.

The random walk will visit a set of states in $\{1, 2, \ldots, n\}$. Define the random variables X_i as follows:

$$X_0 = w_{k_0}, X_i = X_{i-1} \times W_{k_i k_i - 1} \tag{5.16}$$

where the k_i is the state visited in the ith step, $i \in [0, j]$. Let δ represent the Kronecker delta function, i.e. $\delta_{ij} = 1$ if $i = j$, and 0 otherwise. It is shown in [41] that

$$(C^j h)_i = E(X_j \delta_{ik_j}), 1 \leqslant i \leqslant n \tag{5.17}$$

From the linearity of expectation we can easily deduce from Equation (5.17) the following:

$$(\sum_{j=0}^{m} C^j h)_i = E(\sum_{j=0}^{m} X_j \delta_{ik_j}) \tag{5.18}$$

The left hand side of Equation (5.18) is the ith component of x_m described in Equation (5.7). When m is sufficiently large, x_m converges to the solution of the linear system Equation (5.1).

In practice, we realize independently N Markov chains and maintain a running sum for the estimates of each component. Every visit of the state k_j updates the k_jth component of the running sum with all other components estimated to be zero. The final estimate of the

solution is obtained by averaging the running sum over N as specified in Equation (5.19):

$$E(\sum_{j=0}^{m} X_j \delta_{ik_j}) \approx \frac{1}{N} \sum_{n=1}^{N} (\sum_{j=0}^{m_n} X_j \delta_{ik_j}) \tag{5.19}$$

An updated random variable X_j is being added to the running sum at each revision of the latter. The increment becomes less important to the accuracy of the final estimate when X_j gets small compared to the total number of Markov chains N, given that N is the denominator of the estimator. Therefore the magnitude of X_j can be used to control the length of each Markov chain. As indicated in Equation (5.19), the lengths of different chains do not have to be equal.

5.3 Parallel Algorithm and Implementation

The pratice of Monte-Carlo method in linear algebra has a long history. Our revisiting this method is encouraged by its potential for parallelism given the current structure of high performance computing facilities. The essential part of Monte Carlo linear algebra method is the random walk. This key process has to be performed a sufficiently large number of times in order to obtain an acceptable accuracy for the solution. Therefore the independent random walk is a good starting point for a parallel algorithm.

5.3.1 Parallel Algorithm Using CSR Sparse Format

For the reasons discussed previously (see section 1.2 and 4.4), we will continue to use the CSR format for storing sparse matrices. As always, it consists of 3 arrays: $rowptrs, colinds, val$. The test matrix for this study comes from a real thermal-hydraulic application, which happens to be symmetric. When storing a symmetric matrix, the CSR format only stores the upper triangle of it. However, our parallel algorithm does not target particularly symmetric matrices. Accordingly we transform the symmetric CSR storage to a normal one as if the matrix were non-symmetric. Later in Algorithm 5 we are going to compute the transition probabilities. If the MAO method is used, according to Equation (5.13), the probabilities should be computed along the columns. Since the CSR format is row major and the test matrix is symmetric, we simply use the corresponding row instead of column. This is the only place in this chapter where we actually take advantage of matrix symmetry. In practice, if the symmetry is not present, we need to do nothing but simply replace the CSR with CSC (Compressed Sparse Column) format. It is important to note that the symmetry is not a requirement for the

methods or algorithms that we develop in this chapter. They would work just fine for both symmetric and non-symmetric matrices.

Before starting the random walks, some preliminary work needs to be performed on the right-hand side vector b and CSR value array val. In fact, the random walk only needs initial weight vector and weight matrix as indicated in Equation (5.12) and Equation (5.13) without further access to the original matrix and right-hand side. So it is convenient and economic to override the val and b with the weight matrix and initial weight vector.

Algorithm 5 specifies the preliminary phase of the parallel implementation. The random walk will visit a set of states in $\{1,2,\ldots,n\}$, where n is the dimension of the solution vector. At each step of the random walk it has to decide the next state according to the transition probabilities. Here we choose the MAO method (see Equation (5.12), Equation (5.13)) to configure the initial and transition probabilities. There are 2 reasons for it. First, the MAO method requires fewer and shorter chains than the UM method does to achieve the same precision [4]. As for the time complexity, in the case of dense matrices, it is indeed much easier for UM to select the next state. Because its probability follows standard uniform distribution ($p, P \sim U(0,1)$), a simple sampling from a $U(0,1)$ distribution will do. With MAO method we have non-equal transition probabilities. To determine the next state, we first draw a random number from $U(0,1)$ distribution as the probability. Then we compare it with the cumulative distribution of discrete transition probabilities (see Equation (5.12), Equation (5.13)). The purpose is to find out the interval in cumulative distribution the random number falls into, and the corresponding state will be selected as the next state of the random walk. However, for sparse matrix MAO is at least no worse than UM. MAO's transition probabilities are proportional to the magnitude of the elements. So zero entries would never be chosen. This means $O(\log\widetilde{n})$ time complexity if using binary search, where $\widetilde{n}$ is the number of nonzero elements in a column. There is no difference for the MAO method in dealing with either dense or sparse matrix. But the UM method necessitates more steps for each realization of random walk. Since it is possible for UM to choose any state, even those with zero entries, and the sparse matrix is kept in a compressed form, let us assume it is CSR, then the UM has to loop over the indices to check if the selected state has a nonzero value. This additional cost is computationally equivalent to a binary search in MAO, which means that they have same time complexity $O(\log\widetilde{n})$ [1] As a result, the MAO method is chosen for computing the transition probabilities.

Algorithm 6 describes the stochastic process and main procedure of the Monte Carlo solver. It is a single realization of a random walk. The steps 2 to 3 decide the initial state of the random walk and the steps 7 to 10 renew the states. It is noted that the binary search is

[1] UM is more convenient for the very first step. But it is only one step and thus it makes no big difference.

Algorithm 5 Parallel Implementation of Monte Carlo Method for solving symmetric sparse linear system stored in CSR format: Preliminary Phase.

Input: Strictly diagonally dominant symmetric sparse matrix $A \in \Re^{n\times n}$ in CSR format (***rowptrs***, ***colinds***, ***val***). An array ***diagID*** indicating the indices of diagonal entries in *val*. Right-hand side ***b***

Output: Initial weights ***w*** (stored in b). Value array of weight matrix ***W*** (stored in *val*). Cumulative distributions for initial probabilities ***cd_h*** and for transition probabilites ***cd_C***

Step 1: Let $B_1 = diag(A), B_2 = B_1 - A$, store $B_1^{-1}B_2$ in A, and $B_1^{-1}b$ in b. Now C (see (5.3)) is stored in A, h (see (5.4)) is stored in b, the orginal values in *val* and b are no longer needed

1: **for** $i = 1 : n$ **do**
2: $b[i] \leftarrow b[i]/val[diagID[i]]$
3: **for** $j = rowptrs[i] : rowptrs[i+1]$ **do**
4: $val[j] \leftarrow -val[j]/val[diagID[i]]$
5: **end for**
6: $val[diagID[i]] \leftarrow 0$
7: **end for**

Step 2: Use MAO method (see (5.12), (5.13)) to compute the cumulative distributions for both initial probabilities and transition probabilities. The results are stored in *cd_C* (of same size as *val*) and *cd_h* (of size n).

8: **for** $i = 1 : n$ **do**
9: $cd\_h[i] \leftarrow |b[i]|/sum(|b[1:n]|) + cd\_h[i-1]$
10: $s \leftarrow sum(|val[rowptrs[i] : rowptrs[i+1]]|)$
11: **for** $j = rowptrs[i] : rowptrs[i+1]$ **do**
12: $cd\_C[j] \leftarrow |val[j]|/s + cd\_C[j-1]$
13: **end for**
14: **end for**

Step 3: Compute the initial weights and the weight matrix. Store them in b, *val*. Again, the original values in *val* and b are no longer needed

15: **for** $i = 1 : n$ **do**
16: $b[i] \leftarrow sign(b[i]) \times sum(|b[i]|)$
17: $s \leftarrow sum(|val[rowptrs[i] : rowptrs[i+1]]|)$
18: **for** $j = rowptrs[i] : rowptrs[i+1]$ **do**
19: $val[j] \leftarrow sign(val[j]) \times s$
20: **end for**
21: **end for**

Algorithm 6 Conventional Implementation of Monte Carlo Method for solving symmetric sparse linear system stored in CSR format: random walk.

Input: CSR arrays (***rowptrs***, ***colinds***). Initial weight $\boldsymbol{w}$. Value array of weight matrix $\boldsymbol{W}$. Cumulative distributions for initial probabilities ***cd_h*** and for transition probabilites ***cd_C***. Total Chain length threshold $\boldsymbol{\delta}$

Output: ***rsum*** (the estimate of solution)

1: Initialize the running sum *rsum* of size n to 0
2: draw a random number $r \sim U(0,1)$ by using random generator
3: Binary search the ordinal number of r's corresponding interval in *cd_h*, and store the result in $state$
4: $X \leftarrow w[state]$
5: $rsum[state] \leftarrow rsum[state] + X$
6: **while** $|X| > \delta$ **do**
7: draw a random number $r \sim U(0,1)$ by using random generator
8: Binary search the zero-based ordinal number of r's corresponding interval in $cd\_C[rowptrs[state] : rowptrs[state+1]]$, and store the result in j
9: $j \leftarrow rowptrs[state] + j$
10: $next\_state \leftarrow colinds[j]$
11: $X \leftarrow X \times W[j]$
12: $rsum[next\_state] \leftarrow rsum[next\_state] + X$
13: $state = next\_state$
14: **end while**

applied to locate the generated probability r in the cumulative distribution cd_h and cd_C. The steps 4 to 5 and 11 to 12 update the random variable and use it to refresh the running sum. Each random walk constructs an independent Markov chain, the length of the chain can be controlled by the random variable X that is updated at each step. Since the increment contributed by X is diminishes as the chain gets longer, we use a parameter δ as the threshold for X, beneath which the chain performs no further random step. This acts as our stopping criterion for the random walk. The length of each random walk varies because of its stochastic nature. All running sums of different random walks will be summed up and averaged over the number of random walks. The result is the final estimation for the solution $x = A^{-1}b$.

The random walks need to be repeated a large number of times in order to achieve an accurate estimation of the solution. Because of their independence, they can be easily parallelized. Each random walk can be assigned to an execution unit (thread or process). But there is more to discuss about the details of the implementation. In order to avoid race conditions, we let each thread keep a local copy of the running sum. After they finish their job, they perform an atomic update to the global running sum. this comes at the price of a little additional memory, which is affordable.

The second issue is the random generator. We use the Mersenne Twister based random number generator [87] provided by MKL. The computer can only generate pseudo-random numbers: basically, there is a period beyond which the generated random numbers may repeat. Mersenne Twister has a long random sequence with a period of $2^{19937} - 1$. It is possible to create multiple random streams that collect random numbers from that sequence. If all the random numbers are generated within one random stream, the randomness of those numbers should just be fine. The downside of keeping one random stream is the serialization of random number generation because the stream is shared by all the threads. The solution is to create a dedicated random stream for each thread. Now we have multiple random streams initialized with different seeds. These streams are independent because they are designed to deliberately pick different locations from the random sequence. But as the number of streams increases, especially when we need a large number of threads, the randomness would be at stake. To prevent that from happening, we keep the existing random streams alive till the end of execution without creating new stream every time a new parallel section is encountered. We also use the `lrand48` function to engender random seeds for the random streams. The seed of `lrand48` is the global rank of thread.

The preceding discussion only involves thread-level parallelism. The Monte Carlo solver can be easily extended to a larger configuration, if there is available computing resources. We describe in Algorithm 7 a "boss-worker" model implemented with MPI at node level.

Algorithm 7 Parallel Monte Carlo stochastic linear solver based on message passing model.

Initialize N_{proc} processes ranked from 0 to $N_{proc}-1$, the process ranked 0 is the scheduler and the collector of results ("boss rank"), and the others are worker ranks.
if boss **then**
    Initialize an array *rsum* of size n to 0
    for $i = 1 : N_{tot}$ **do**
        Receive the local estimation from any source and store it in buf
        $rsum \leftarrow rsum + buf$
    end for
    $rsum[1:n] \leftarrow \frac{1}{N_{tot}} rsum[1:n]$
    Tell all the other processes to stop running
else
    Perform N_{sub} times of random walk, and store the result in *rsum*
    if boss tells me to stop **then**
        Stop
    else
        Send *rsum* to boss
    end if
end if

As different realizations of random walk are independent, they can be dispatched to any available computing resource. The boss is in charge of receiving the results and broadcasting the stop sign. The Algorithm 7 actually exhibits a prominent advantage, which is a very low amount of communication. Only the local solution needs to be sent. Communications often appear to be the principal bottleneck of the parallel execution. In our case, since the communication is limited, a good scalability can be expected. Yet the random number generation can still be a problem when we increase the number of processes number. We will have a larger discuss on these issues in the next sections, with the support of numerical results.

Assuming there are in total N random walks, Figure 5.1 depicts how the parallel Monte Carlo linear solver is implemented. The random walks develop along the directed edges. At each step within a random walk, there is a transition of state to perform unless the stopping criterion is met. Each column, which represents a particular random walk, is executed in parallel with other columns.

5.3.2 Considerations for multi-core and many-core architecture

To squeeze the best performance out of many-core architecture is tricky, especially for the parallel implementation of the Monte Carlo solver. According to our prior experience

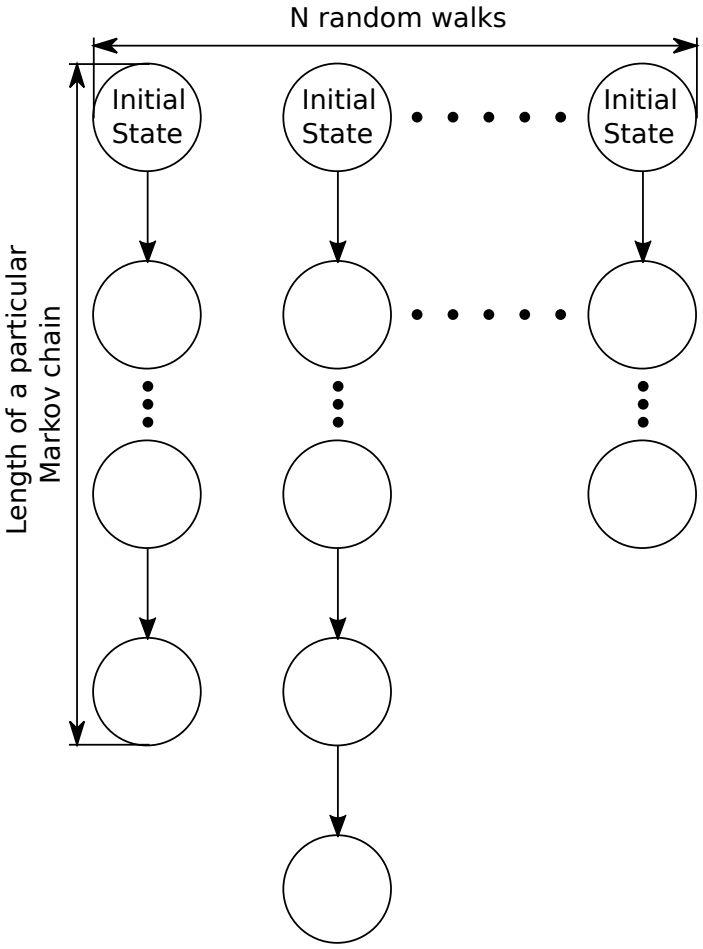


Figure 5.1 Parallel Monte Carlo linear solver: performing N times the key stochastic process (random walk). In the diagram, a node represents a particular state of the Markov chain (or a step in the random walk), and the directed edges represent transitions.

on KNC, we recognize the importance of both multithreading and vectorization to the performance. Basically, the parallel Monte Carlo linear solver lacks structured, continuous and regular memory access. As shown in Figure 5.1, the alternation of states is a serial line that supports hardly any vectorization: almost every operation depends on the result of the previous one. There is only one place that may benefit from the SIMD instructions: that is the random number generation. Since there is a demand for a great amount of random numbers, the effect of vectorization should be significant for both KNC and host processors.

The use of multithreading is obvious, and it is shown in chapter 4 that hybrid MPI/OpenMP model would help to mitigate the scheduling overhead and increase data locality. We have a similar scenario here: multiple MPI processes can be launched on the same coprocessor with each process managing a pool of threads. Some data will be replicated over the processes in the hope of generating higher aggregate on-chip bandwidth. It may also help with the cache-hit rate.

5.3.3 Experiments on Convergence

In this section we will discuss the convergence properties of the parallel Monte Carlo solver via experimental results. In section 5.3.1, we mentioned a parameter δ used to control the length of the Markov chain. The formula Equation (5.20) shows how we compute it:

$$\delta = coef \times n_{walks} \times n \tag{5.20}$$

where n_{walks} is the number of random walks, n is the matrix dimension [2]. δ actually represents a minimum acceptable increment to the running sum. Because it will be averaged over n_{walks}, we put the second term into Equation (5.20). In order to ensure the equal chance for all the states to be visited, we put the third term into Equation (5.20). In such a way, $coef$ reflects a normalized level without regard to matrices. In our experiments, the $coef$ is set to 10^{-15}.

The test matrix for this study comes from a real thermal-hydraulic application. It is called "Matrice_Morse_Sym1000", which is a symmetric sparse square matrix of size 1000×1000. It comes from the solution of the Navier-Stokes equation and it has 6400 nonzero elements.

Besides our conventional experimental platform of KNC and dual-socket Sandy Bridge processors (see subsection 3.4.2), we also use the lab cluster "Poincaré" for the scalability study. The "Poincaré" is an IBM system, composed mainly of iDataPlex dx360 M4 servers [3]. It has 92 computing nodes equipped with 2 processors of Intel Sandy Bridge E5-2670 [4]

[2] Square matrix.

[3] The IBM System x iDataPlex dx360 M4 is a half-depth, dual-socket server designed for data centers that require high performance, yet are constrained on floor space, power, and cooling infrastructure.

[4] 2.6 GHz octacore processor, which makes 16 cores per node.

and 32 GB of memory per node. The "Poincaré" also has 4 GPU nodes equipped with 2 processors of Sandy Bridge E5-2670, 64 GB memory per node, and 2 GPU Tesla K20 (CUDA capability 3.5, 4.8 GB of GDDR memory per GPU. More information about "Poincaré" is available here [85].

We first conduct the convergence experiments for parallel Monte Carlo linear solver on "Poincaré".

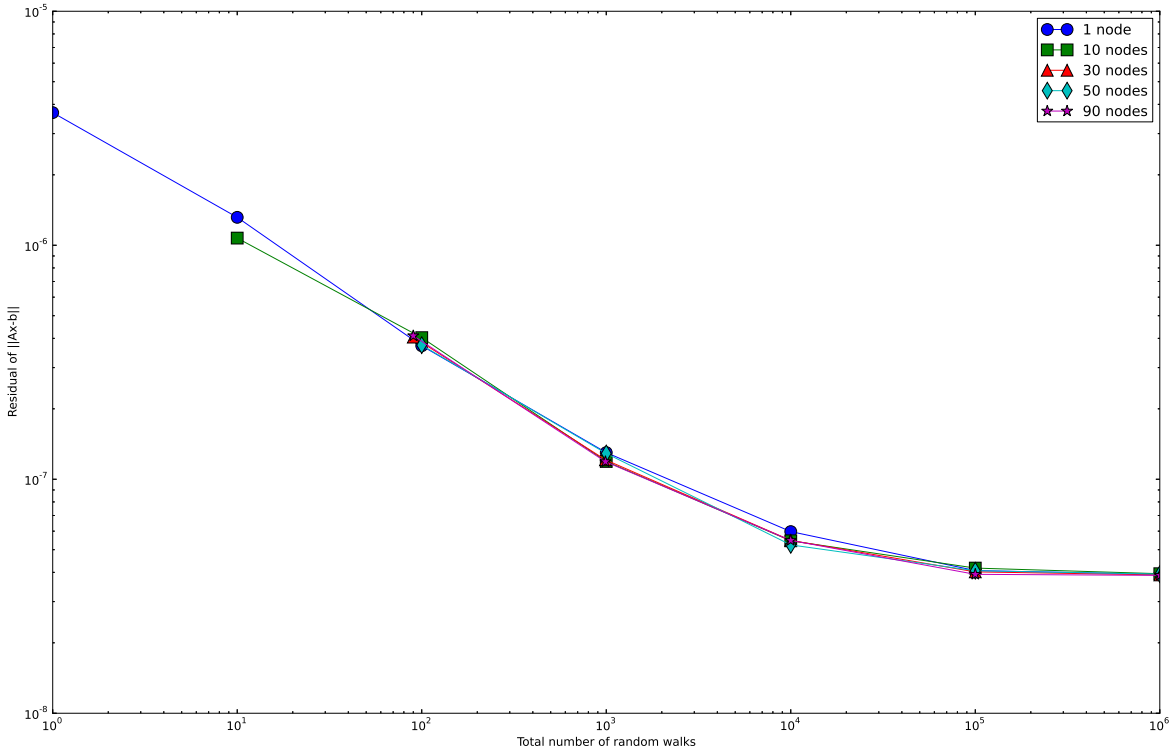


Figure 5.2 Curves of residual for different executions of the parallel Monte Carlo solver. Two parameters are varied for these experiments: 1) the number of random walks indicated by abscissa, 2) the number of computing nodes represented by different curves. A base 10 logarithmic scale is used for both x-axis and y-axis.

The Figure 5.2 shows the convergence as a function of the number of random walks and for different values of the number of nodes. The workload is uniformly distributed over the computing nodes. The horizontal axis is the accumulative count of random walks within each execution. In Figure 5.2, similar convergence trends are observed for executions with different numbers of computing nodes. As discussed in subsection 5.3.1, using more processes requires the creation of more random streams. But up to 90 computing nodes [5], there is still no evident impact on convergence, which means that our strategy with the random streams is effective.

[5]The lab cluster's capacity is 92 computing nodes.

As a comparison, we also launch a plain conjugate gradient (CG) solver without preconditioning for solving the same linear system. The solver interface is provided by MKL. If setting the same precision of convergence as indicated in Figure 5.2 for CG solver, the Monte Carlo solver is 2 to 3 times faster than CG solver using a single node of "Poincaré". But in terms of highest achievable precision, CG solver can do much better than the Monte Carlo solver.

5.3.4 Maximizing Performance on multi-core/many-core Processors

In all of our experiments, the timing is measured in between the step 2 and step 8 in Algorithm 7 by taking the view of the "boss" process.

We first run the Algorithm 6 on both dual-socket Sandy Bridge (SNB) processors and one KNC coprocessor (see subsection 3.4.2). Only one process is initiated on both platforms with multiple threads. The random walks are dispatched to the threads using OpenMP loop scheduling. When the total number N of random walks is small, the performance of dual-socket Sandy Bridge is better than KNC. As the Sandy Bridge is designed to be latency-oriented, it is much better at executing branch statements than KNC, which is very helpful in binary search. However, when N reaches the order of 10^8, the vectorized random number generation begins to show its importance. Because of more and longer SIMD units in KNC, it outruns dual-socket Sandy Bridge at this point. For execution on SNB, 16 threads (1 thread per core) are used. For execution on KNC, different numbers of threads (60, 120, 180, 240) are tested. The experiments show that the optimal execution is achieved with 240 threads, whose performance is 1.03 times faster than using 180 threads, 1.15 times faster than using 120 threads, 1.75 times faster than using 60 threads, and 1.25 times faster than using 16 threads on dual-socket Sandy Bridge.

Recalling what we said in subsection 5.3.2, we may further improve the performance on KNC by mixing more than one MPI process and OpenMP threads. So we evenly divide the domain of KNC cores with regard to common resources (cores, caches) and place one MPI process in each subdomain. In each subdomain, according to the results obtained by using 1 process, we use 4 threads per core. So there would be 240 threads in total. To achieve the best performance, the core affinity should also be taken into account: we have to make sure that the threads are pinned properly to cores for better cache usage and that the highest numbered core (the 61th core) is kept from being used.

Here is the correct configuration for 2 MPI processes running on KNC with 120 threads per process.

```
export I_MPI_PIN=1
```

```
export I_MPI_PIN_DOMAIN=120:compact
export KMP_AFFINITY=granularity=thread,scatter
export KMP_PLACE_THREADS=30c,4t
```

`export I_MPI_PIN=1` is used to activate the process pinning. There are 244 logical cores (61×4 logical cores per physical core) on KNC. We demand only 240 logical cores and leave the 61th physical core [6] unused. These 240 logical cores are partitioned into 2 groups of 120. The "`export I_MPI_PIN_DOMAIN=120:compact`") will set the domain members to be located as close to each other as possible in terms of common resources (cores and cache). `KMP_AFFINITY` defines the thread affinity when the "`granularity` is set to "`thread`".

Table 5.1 Five different thread affinities to use in OpenMP runtime environment supported by Intel.

Type	Specific
none	Does not bind OpenMP threads to particular thread contexts. However, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology
compact	Specifying compact assigns the OpenMP thread <*n*>+1 to a free thread context as close as possible to the thread context where the <*n*> OpenMP thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads.
disabled	Specifying disabled completely disables the thread affinity interfaces. This forces the OpenMP runtime library to behave as if the affinity interface was not supported by the operating system. This includes the low-level API interfaces such as `KMP_SET_AFFINITY` and `KMP_GET_AFFINITY`, which have no effect and will return a nonzero error code
explicit	Specifying explicit assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the `proc_list=` modifier, which is required for this affinity type
scatter	Specifying scatter distributes the threads as evenly as possible across the entire system. Scatter is the opposite of compact. So the leaves of the node are most significant when sorting through the machine topology map

Table 5.1 lists the available thread affinities in OpenMP runtime environment supported by Intel. In this case of parallel Monte Carlo solver, we set the affinity to "scatter". Finally, "`KMP_PLACE_THREADS`" informs each process to use 30 physical cores with 4 threads per core. The joint use of both "`KMP_AFFINITY`" and "`KMP_PLACE_THREADS`" will prevent the highest numbered physical core from being used. Otherwise the "scatter" will take the numbering of the 61th physical core into account.

[6] The logical cores that belong to the 61th physical core are numbered 241, 242, 243, 0.

We tested on KNC with 2, 4, and 10 MPI processes. Compared with the prior case of 1 process and 240 threads, which is the best configuration by that time, the improvements of 14.3%, 21.4%, and 22.5% are observed respectively (see Table 5.2), which confirms our theory in subsection 5.3.2.

Table 5.2 Speedups of parallel Monte Carlo linear solver running on one KNC coprocessor using hybrid MPI/OpenMP model.

Number of processes	Number of threads per process	Speedup
1	240	1
2	120	1.143
4	60	1.214
10	24	1.225

5.3.5 Scalability in Performance

In scalability tests, we use up to 4 KNC coprocessors.

The first experiment is the strong scaling test. We fix the total number of random walks to 4×10^8. For each run, we split evenly the workload among the involved coprocessor(s). The second experiment is the weak scaling test. We fix the number of random walks to 10^8 for each coprocessor.

Figure 5.3 delineates the strong and weak scaling of parallel Monte Carlo linear solver using up to 4 KNC coprocessors with 240 threads per coprocessor. The scaling performances are shown in speedup bars relative to mono-thread performance on one KNC. The ordinate values are indicated above the bars.

The same strong scaling test is also conducted on the cluster "Poincaré" (see subsection 5.3.3). In this case, we fix the total number of random walks to 10^8 and distribute uniformly the workload among the involved computing nodes. Figure 5.4 delineates the strong scaling of parallel Monte Carlo linear solver on "Poincaré" using up to 90 computing nodes with 16 threads per node. The scaling performances are shown in speedup bars relative to the performance on a single computing node. The ordinate values are indicated above the bars.

As shown in Figure 5.3 and Figure 5.4, a good percentage of linear scaling efficiency is achieved on both KNC platform and CPU cluster. In the case of KNC, there is still 80% of perfect scaling performance when 4 KNC coprocessors are used. In the case of Poincaré, there is 50% of perfect scaling performance when 90 computing nodes are used.

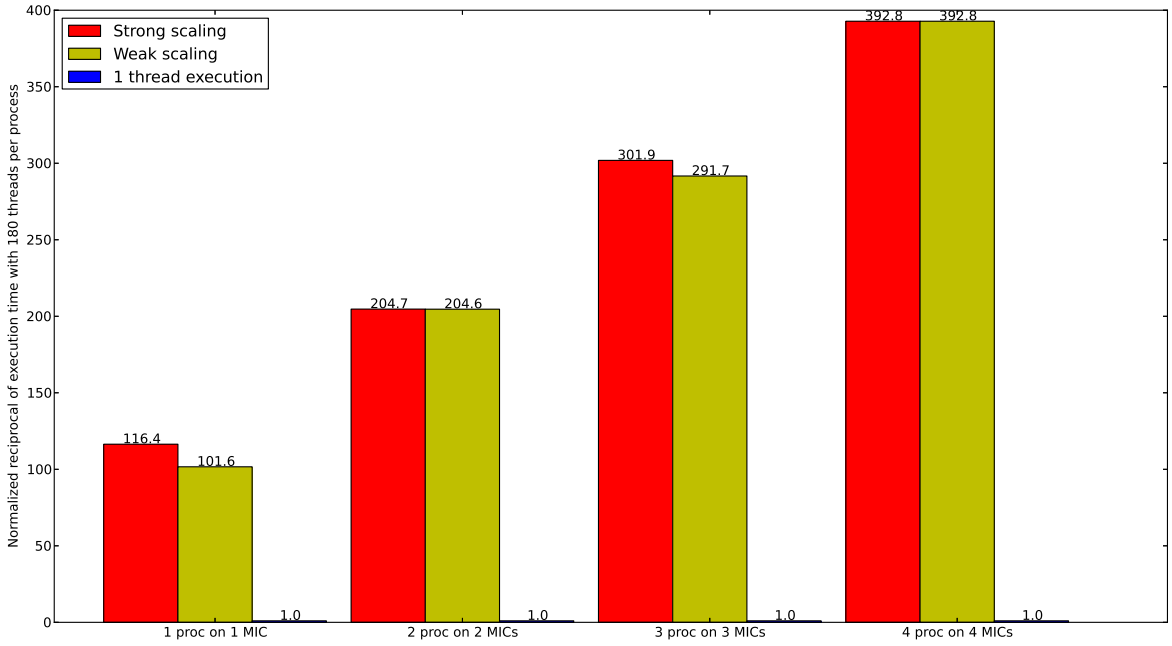


Figure 5.3 Strong and weak scaling of parallel Monte Carlo linear solver using up to 4 KNC coprocessors.

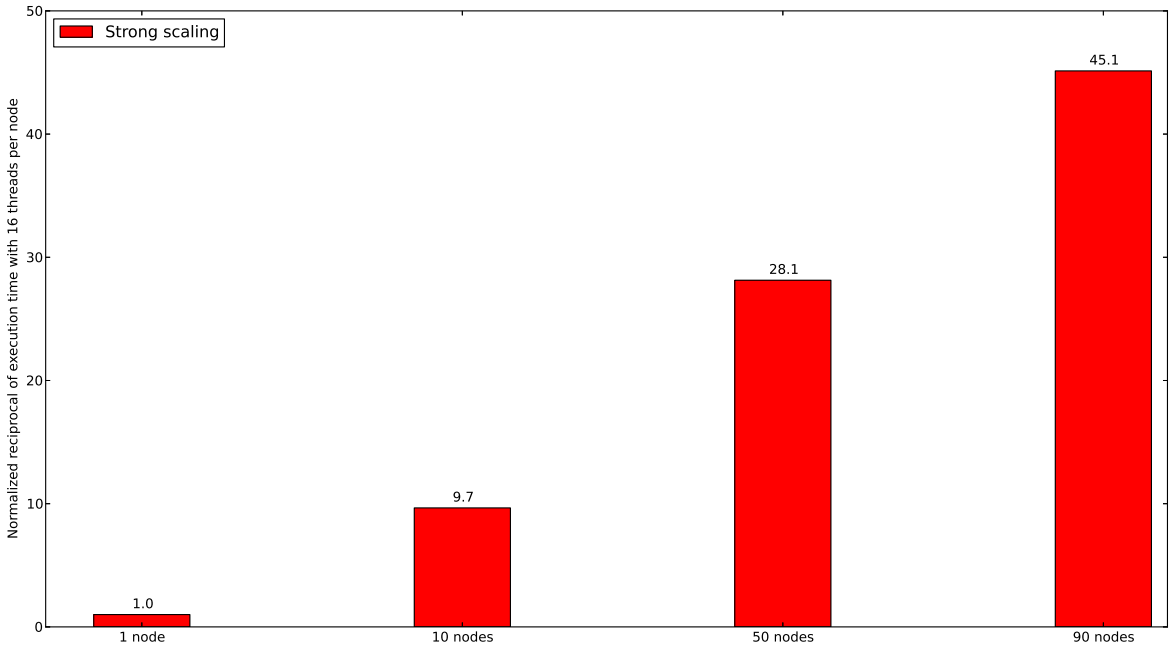


Figure 5.4 Strong scaling of parallel Monte Carlo linear solver on lab cluster "Poincaré" using up to 90 computing nodes.

We argue that the bottleneck of scalability is step 6 in Algorithm 7, The communication can not be continued until the local result is summed to the global one. This can be easily fixed: a quick solution is to assign one thread in the "boss" process to manage the communication, and the rest of the threads participate in updating the *rsum*. A buffer will be needed in case the consuming rate is slower than the receiving rate. The model described in Algorithm 7 can be further extended into a hierarchy. There can be multiple "small bosses" processes that collect running sums and update them in parallel. The final solution will be reduced to the "big boss" process. By doing so the parallel Monte Carlo linear solver is expected to have better scalability. However, we will not go down this road, because in the following sections we are going to introduce another change to the method that brings even more significant improvements.

5.4 Performance Bottlenecks of the Conventional Random Walk

Before introducing the new change to the method, we will itemize in this section all the performance issues that are related to the previous parallel implementation of Monte Carlo linear solver.

5.4.1 Random Number Generation

The random number generation usually takes a nonnegligible part of the total amount of time. In our case for example, over 20% of execution time is devoted to the random number generation. If somehow this procedure could be omitted or simplified, that would be a great saving of time.

In fact, the computing time is not the only cost of random number generation. As we know that the computer generates only pseudo-random numbers, there is a period beyond which the generated numbers will repeat. A random stream is initialized so that it will pick random numbers out of a long random sequence from a starting point. The access to the random stream has to be private for reasons of parallel efficiency. If all the processes or threads share a single random stream, then the serialized accesses may drag considerably the parallel performance. Our solution to this is to initialize a private random stream for each worker (process or thread). However, as the total number of workers increases, the randomness of generated numbers can be affected. And a large number of random streams is hard to manage, which will introduce some overhead to the execution.

5.4.2 Selection of New State

Each time a new random number r is drawn from $U(0,1)$, it should be used to determine the next state by locating its corresponding interval in the cumulative distribution of transition probabilities. In terms of time complexity, the best we can do for the localization in an array is to apply the binary search that will get us an answer in logarithmic time. In practice, the search space is limited to the dimension of the solution vector. It can be even smaller because of the sparsity of matrix [7], so linear search may outperform binary search by vectorizing the code.

Nevertheless, the selection of new states is the most time consuming routine in the Monte Carlo linear solver. In the worst case, half of the execution time is invested in the binary search. As shown in the experiments analyzed in subsection 5.3.4, the KNC is not particularly good at executing branches because it is not latency-oriented. Our experience with vectorized linear search is also not very promising. We have to find a third solution that somehow bypasses the weakness of the many-core architecture like MIC.

5.4.3 Lack of Vectorization

The random walk described in Algorithm 6 is assigned to the execution units (threads or processes) as a whole in our first parallel implementation. If we look into Algorithm 6, we can hardly find any assignment suited for vector units. As shown in Figure 5.1, the transition of states is a serial process, in which there is hardly anything that can be vectorized except for the random number generation. According to our first results discussed in subsection 5.3.4, the effect of the vectorized random number generation is not evident until the total number of random walks has grown very large. The random number generation would be the best candidate for it, while the binary search or the updates of the running sum have too much dependencies which prevent the vectorization of the code. It is far from enough for an efficient parallel Monte Carlo solver to vectorize only the random number generation. The KNC is supposed to achieve high performance with adequate vectorized instructions. Recalling that the cores on KNC can not compete with their powerful counterparts on Sandy Bridge, their superiority comes from the number and the length of its vector processing units. The lack of vectorization on such architecture would waste the most of its hardware resource, which leads to suboptimal executions.

[7] Zero entries never get selected.

5.4.4 Variability of Numerical Results

If we consider the Monte Carlo solver as an operator applied to the right-hand side of the linear system, then this operator varies from run to run. Even if each execution unit is given the same sequence of random numbers, the result still depends heavily on the execution order. It is impossible to reproduce precisely the result of a previous execution. This is due to the stochastic nature of the method.

As mentioned in Section 5.1, the Monte Carlo linear solver does not have the same converging process as other iterative methods. Typically it converges rapidly to a rough estimation of the solution and then stagnates. It has difficulty in obtaining numerically precise solutions. But for those scenarios where an approximate solution is sufficient, it is still an eligible method. Such scenarios include graph partitioning, information retrieval, feature extraction, and of course preconditioning. Our next step in studying Monte Carlo solver is to develop its potential in being a preconditioning method, considering its numerical particularity.

Therefore, the first problem to solve is the variability of Monte Carlo solver induced by its stochastic nature. If the Monte Carlo method acts as an "operator" that is being applied to a vector, the result is expected to be reproducible. The Monte Carlo method that we have now does not yet meet that requirement.

5.5 New Task-Based Reformulation

In order to address the aforementioned performance issues, we propose a new task-based execution model for the Monte Carlo linear solver. The new model particularly changes the way that the random walk is carried out. It will have 3 major improvements. First, it ensures the reproducibility of results. Second, it increases the vectorizability of the algorithm. Third, it eliminates the most time-consuming bottleneck and thus improves the performance.

5.5.1 Implementation Details

If we look back and think about how to tackle the performance issues listed in section 5.4, the best we can do for the random number generation is to vectorize it, and there seems to be no way to improve the binary search. But just imagine that we bypass them, both the random number generation and binary search. What if we do not perform any of those at all? In the previous implementation, a random number is drawn as the probability to determine the next state. Then the process of selection is basically a binary search. The random number

generation and binary search are repeated over and over again for only one reason, which is the determination of future states.

But if we do not actually care what the future state is, in other words, if we select all the possible states, then we simply do not need the random number generation, nor the binary search. Since we are aware of the transition probabilities that are related to all the possible states, it is possible to no longer draw a random number but simply utilize the transition probability to determine the number of times its corresponding state should be visited. In fact, we can go one step further: the number of times of that a state is visited does not need to be an integer. Using integers to represent the "number of times" may agree with the common sense. But in our case it will lead to the loss of information because the transition probabilities are unlikely to be integers. When they are multiplied by an integer number of visits, the results obtained thereof are floating numbers, which also represent future number of visits.

The purpose of visiting a state x times is to add the updated random variable to the running sum x times. The update to the running sum is indicated in the step 12 in Algorithm 6. Now it should be revised as:

$$rsum[next\_state] \leftarrow rsum[next\_state] + X \times num\_visits \tag{5.21}$$

In this light, the number of visits is more like "the weight for the update". In the old parallel implementation, a state will only be visited once at each step. Visiting a state a non-integer number of times is not an option in that scenario, whereas in this new model, we are allowed to traverse the states in a more "precise" way.

From any given state, there is a number of possibilities for the next state. When we deal with the ensemble of possible next states, there will be more than one update to the running sum, which makes it possible to vectorize the update and thus remedy the lack of vectorization.

Figure 5.5 shows the task tree of the new model in the general case. The root task initiates the new version of the "random walk". It starts from a virtual state, from where it starts to traverse all the possible states. Different from the child task, the root task only visits all the initial states. The random variable X and running sum *rsum* are initialized at this point, and they will be passed to the child tasks. The child task does basically the same thing as the root task, except that it uses the transition probabilities instead of the initial probabilities. As seen from Figure 5.5, there is no longer a need for the cumulative distributions, which in turn helps to simplify the preliminary phase. Compared with the original preliminary phase described in Algorithm 5, the new one will need to revise the step 2 in order to compute directly the initial and transition probabilities. The rest shall remain the same.

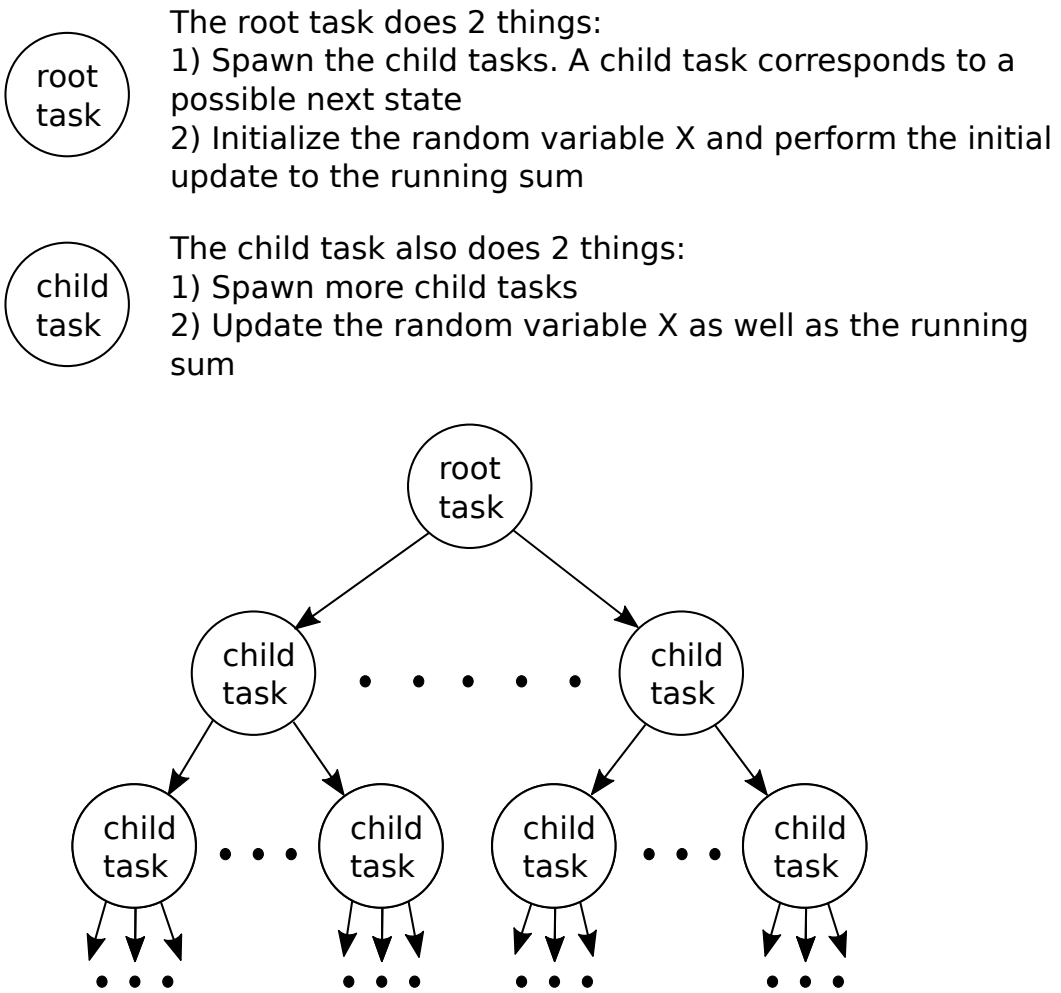


Figure 5.5 New task-based execution model for the Monte Carlo linear solver. In the diagram, the nodes represent the tasks, and the directed edges represent the dependencies between the tasks

The root task is formally defined in Algorithm 8. So is the child task defined in Algorithm 9.

Algorithm 8 Task-based execution model of Monte Carlo solver for sparse matrices stored in CSR format: the root task.

Input: CSR arrays (***rowptrs***, ***colinds***). Initial weight $\boldsymbol{w}$. Value array of weight matrix $\boldsymbol{W}$. Initial probabilities $\boldsymbol{p}$. Total number of random walks $\boldsymbol{N}$. Chain length threshold $\boldsymbol{\tau}$

1: **for** $i = 1 : n$ **do**
2: $num\_visits \leftarrow p[i] \times N$
3: **if** $num\_visits > \tau$ **then**
4: $X \leftarrow w[i]$
5: $rsum[i] \leftarrow X \times num\_visits$
6: spawn a new child task with the inputs (X, i, num_visits)
7: **end if**
8: **end for**

Algorithm 9 Task-based execution model of Monte Carlo solver for sparse matrices stored in CSR format: the child task.

Input: CSR arrays (***rowptrs***, ***colinds***). Initial weight $\boldsymbol{w}$. Value array of weight matrix $\boldsymbol{W}$. Transition probabilities $\boldsymbol{P}$. Random variable $\boldsymbol{X}$. Current state ***curSt***. Number of visits $\boldsymbol{N_v}$. Chain length threshold $\boldsymbol{\delta}$

1: **for** $i = rowptrs[curSt] : rowptrs[curSt + 1]$ **do**
2: $num\_visits \leftarrow P[i] \times N_v$
3: **if** $num\_visits > \tau$ **then**
4: $next\_state \leftarrow colinds[i]$
5: $X' \leftarrow X \times W[i]$
6: $rsum[next\_state] \leftarrow rsum[next\_state] + X' \times num\_visits$
7: spawn a new child task with the inputs $(X', next\_state, num\_visits)$
8: **end if**
9: **end for**

Again, we need to reiterate the fact that we use the corresponding row in lieu of column to compute the transition probabilities by taking advantage of the matrix symmetry. If the matrix is not symmetric, simply replace the CSR with CSC (Compressed Sparse Column) format. The rest of the method remains the same.

Now the first and second issues discussed in section 5.4 are all properly dealt with. There should also be a solution for the fourth problem [8]. In both Algorithm 8 and Algorithm 9 a different parameter τ is used as the stopping criterion. The δ in Algorithm 6 oversees the magnitude of the random variable X while the τ here controls the number of visits.

[8] The third issue will be dealt with in subsection 5.5.3

The reason for introducing this change is nothing but the last performance issue defined in subsection 5.4.4.

If we continue to rely on the magnitude of a random variable in order to limit the length of the random walk, the result will still be dependent on the right-hand side vector. Recall that the random variable X is initialized with the initial weights that are derived from the right-hand side vector. To eliminate the effects of irrelevant quantities, only the matrix-related coefficients should be used by the stopping condition. The crux of the problem lies in the initialization in the root task. We argue that only the root task matters. Because in the child task both the weight, which is used to update the random variable, and the transition probabilities, which are used to update the number of visits, are related only to the matrix. But the initial step conducted by the root task is based on the initial weights and probabilities, which are related to the right-hand side vector. The initial weight is a function of the right-hand side vector. However, there is still a way to define initial probabilities that are irrelevant to the right-hand side vector, by using the UM method. The UM method defines the initial probabilities as $p_i = \frac{1}{n}$, where the n is the dimension of the matrix. With such a configuration, the number of visits is merely a function of the total number of random walks N, the parameter τ and the matrix. The execution of the Monte Carlo solver becomes predictable and reproducible. Whatever the right-hand side vector or the execution order of the tasks, there is always an invariant "operator", which perfectly solves the problem raised in subsection 5.4.4.

The preceding discussion only involves the initial probabilities and weights. The computation of the transition probabilities and the weights is still an open question. If we take a closer look at the random variable X, its general term can be expressed as in Equation (5.22):

$$X = w_{i_1} W_{i_2} W_{i_3} \dots W_{i_x} \tag{5.22}$$

By Equation (5.8) and Equation (5.10), the update to the running sum can be further developed as in Equation (5.23):

$$\begin{aligned} rsum_i &= rsum_i + X \\ &= rsum_i + w_{i_1} W_{i_2} W_{i_3} \dots W_{i_x} \times p_{i_1} P_{i_2} P_{i_3} \dots P_{i_x} \\ &= rsum_i + h_{i_1} C_{i_2} C_{i_3} \dots C_{i_x} \end{aligned} \tag{5.23}$$

It is clear that the random walk path $i_1 \to i_2 \to i_3 \to \dots \to i_x$ dominates the convergence process. In the original parallel implementation, the reason why MAO method leads to a better convergence rate is that MAO selects the path more carefully while UM does it indiscriminately. So UM requires more realizations of random walk than MAO to converge

to the same precision. However, our model proposes an algorithm that does not depend on the choice of the path. After all, all the possible paths will be traversed at each step: that will neutralize the negative impact of UM method on convergence. Now the only difference between choosing UM or MAO is the stopping condition.

Since we use the number of visits as the threshold,the UM method will cause all the child tasks to stop at the same level, while MAO method will tolerate some of the child tasks to go further.

5.5.2 Proof of Equivalence

Suppose we will perform N random walks. Consider a traversal of states which is similar to Figure 5.5, but the nodes represent the states and the edges represent the transitions. Consider also the old traversal of states described in Figure 5.1. If we combine at each level the same states in Figure 5.1, and add a virtual state at the top of graph and direct it to all the level-1 states, we get a graph which is almost the same as Figure 5.5, except for a slightly different number of visits.

When N tends to infinity, the proportion of different states visited at different levels in the combined graph equals the transition probabilities, which gives us two identical traversal of states. In fact, the original one can be considered as the depth-first traversal of the state tree, while the new one can be considered as the breadth-first traversal of the state tree. The essence is the same. Q.E.D.

5.5.3 Optimization

The first optimization targets the root task. Equation (5.8) tells us that the expected value of initial update to the running sum is h. From Equation (5.4) we know that h is the multiplication of inverse of the matrix diagonal and right-hand side vector, which is already available to us in the preliminary phase. Since the UM method is used to define the initial probabilities and weights, there is $p_i = \frac{1}{n}, w_i = nh_i$. It means that there is no longer a need to explicitly compute the running sum in the root task. What we do instead is to add h to the final averaged running sum in order to obtain the final estimation for the solution.

The second optimization targets the vectorization. Since both the root task and child task process several independent child states at a time, it is possible to vectorize the processing. Moreover, there is the `gather` instruction that loads sparse locations of memory into a dense vector register. This will help with the irregular memory access.

The last optimization targets the computing of transition probabilities and weights. As proved in subsection 5.5.1, the UM method is no longer harmful to the convergence. And we

do want a method that helps us to compute more easily and quickly. The main difference between MAO and UM in this case is that MAO needs to check on every next state to see if it meets the stopping condition. UM only checks once, which saves us a lot of if-statements. This will be of great help in accelerating the program. So we will adopt UM as the method to compute the transition probabilities and weights.

The optimized version of Algorithm 8 and Algorithm 9 is elaborated in Algorithm 10 and Algorithm 11. The remainder loop is deliberately ignored for brevity.

Algorithm 10 Task-based execution model of Monte Carlo solver for sparse matrices stored in CSR format: the optimized root task.

Input: CSR arrays (***rowptrs***, ***colinds***). Initial weight $\boldsymbol{w}$. Value array of weight matrix $\boldsymbol{W}$. Total number of random walks $\boldsymbol{N}$. Chain length threshold $\boldsymbol{\tau}$. Vector length $\boldsymbol{m}$

1: $num\_visits \leftarrow N/n$
2: **if** $num\_visits > \tau$ **then**
3: **for** $i = 1 : n$ **do**
4: spawn a new child task with the inputs $(w[i], i, num\_visits)$
5: **end for**
6: **end if**

Algorithm 11 Task-based execution model of Monte Carlo solver for sparse matrices stored in CSR format: the optimized child task.

Input: CSR arrays (***rowptrs***, ***colinds***). Initial weight $\boldsymbol{w}$. Value array of weight matrix $\boldsymbol{W}$. Transition probabilities $\boldsymbol{P}$. Random variable $\boldsymbol{X}$. Current state ***curSt***. Number of visits $\boldsymbol{N_v}$. Chain length threshold $\boldsymbol{\delta}$. Vector length $\boldsymbol{m}$

1: $num\_visits \leftarrow N_v/n$
2: **if** $num\_visits > \tau$ **then**
3: **for** $i = 0 : (rowptrs[curSt+1] - rowptrs[curSt])/m$ **do**
4: $vec_{next\_state} \leftarrow load(\&colinds[rowptrs[curSt] + i \times m])$
5: $vec_W \leftarrow load(\&W[rowptrs[curSt] + i \times m])$
6: $vec_X \leftarrow set1(X)$
7: $vec_{num\_visits} \leftarrow set1(num\_visits)$
8: $vec_{rsum} \leftarrow gather(vec_{next\_states}, \&rsum[0])$
9: $vec_X \leftarrow mul(vec_X, vec_W)$
10: $vec_{rsum} \leftarrow fmadd(vec_X, vec_{num\_visits}, vec_{rsum})$
11: $scatter(\&rsum[0], vec_{next\_states}, vec_{rsum})$
12: spawn m new child task with the inputs $(vec_X[0], vec_{next\_state}[0], num\_visits), \ldots, (vec_X[m-1], vec_{next\_state}[m-1], num\_visits)$
13: **end for**
14: **end if**

Algorithm 11 shows the pseudocode of the SIMDized kernel. First, the elements from *colinds* and *W* are loaded into vectors (line 4 and 5). Next, we set respectively the elements of vec_X and vec_{num_visits} to equal values of *X* and *num_visits*. The elements from *rsum* are gathered into vec_{rsum}. vec_X is first multiplied by vec_W then vec_X and vec_{num_visits} are multiplied and added to vec_{rsum} by the fused multiply-add instruction. Finally the results in vec_{rsum} are scattered into *rsum*. At the end of the child task, new child tasks are spawned.

5.5.4 Runtime Choice

There are plenty of choices for runtime system. A lot of studies have been conducted on DAG-based dynamic task scheduling. Multiple frameworks, including OpenMP, Cilk, TBB, StarPU, QUARK, or others (see section 2.2), can be applied to implement the above tasks.

TBB has implemented a recursive model of task-based parallelism. So it is quite straightforward to use TBB to implement our model. As a first step validating the method, we define and spawn recursively the tasks within TBB. The fundamental strategy of the default task scheduler of TBB is breadth-first theft and depth-first work. The breadth-first theft raises parallelism sufficiently to keep threads busy. The depth-first work rule keeps each thread operating efficiently once it has sufficient work to do. We add a minor optimization to scheduling: at the end of both root and child task, we directly specify the first child task as the next task to run, if there is any. By doing so we avoid the scheduler from deciding which task to pick. Such activity of scheduler will introduce the overhead incurred by putting the task into the ready pool and then getting it back out.

There is one more detail about the memory arrangement for the running sum. Since the updates to the running sum is commutative and associative, only the final reduction of results matters. Since the running sum is modified very frequently, using a shared copy induces a corresponding amount of lock contention, which will ultimately result in a loss of scalability. Hence we split the running sum into separate thread-local pieces. Each thread operates only on its thread-local piece, thus removing the contention for it. In practice, we declare a vector of `combinable<double>` constructs for the running sum. The final results will be obtained by the `combine()` method.

5.5.5 Performance Analysis

In this section we will present the first performance results obtained from a few matrices resulting from the 3D CFD Trio_U code [25]. We also include an external matrix which is publicly available from the university of Florida sparse matrix collection [37].

As before, the experimental environment is comprised of dual-socket Sandy Bridge processors and a KNC coprocessor.

The Table 5.3 lists the structural information of the test matrices, including the matrix names, dimensions, number of nonzero elements, average number of nonzero elements per row, maximum number of nonzero elements per row, and minimum number of nonzero elements per row.

Table 5.3 Table of sparse matrices that are used in performance test of task-based execution model for the Monte Carlo linear solver.

Name	Dimension	nnz	Avg nnz/row	Max nnz/row	Min nnz/row	Source
MorseSym1000	1000×1000	6400	6.4	7	4	Trio_U
MorseSym10000	10000×10000	65800	6.58	7	4	Trio_U
VEF64000	4241×4241	52692	12.42	39	1	Trio_U
VEF8000	649×649	7180	11.06	39	1	Trio_U
bcsstk17	10974×10974	428650	39.06	150	1	Florida

The matrix suite listed in Table 5.3 is used to test both the original parallel implementation and task-based execution model of the Monte Carlo linear solver. The task-based model is implemented using TBB, which is built on a pure multithreading runtime environment. As a comparison, we also test the optimized multithreaded version of the original parallel implementation on the same matrix suite. The original parallel implementation is based on OpenMP runtime system, which achieves its best performance with 240 threads, according to our first experiments (see subsection 5.3.4). Here we do not change the number of threads, and run the solver until the convergence stagnates [9], which takes around 10^6 times of random walks. As for the task-based model implemented with TBB, we simply leave the TBB runtime system to decide how many threads to use [10]. Because the TBB scheduler has been designed to automatically decide on the number of threads [11]. The Table 5.4 reports the results of these experiments.

The performance values shown in Table 5.4 are proportional to the reciprocal of their corresponding execution time. And they are normalized by their counterparts displayed on the second column (the column named "KNC" under "Original Parallel Implementation").

[9] Or we simply consider that the Monte Carlo solver "converges" on a solution that is less accurate than those of the classical iterative solvers.

[10] We observed that the TBB runtime system also used 240 threads.

[11] The reason for not specifying the number of threads, especially in production code, is that in a large software project, there is no way for various components to know how many threads would be optimal for other threads. Hardware threads are a shared global resource. It is best to leave the decision of how many threads to use to the task scheduler.

Table 5.4 Experimental results obtained from testing both the original parallel implementation and task-based model of Monte Carlo linear solver. All the performance values are normalized by the corresponding performance value on the second column (the column named "KNC" under "Original Parallel Implementation"). The performance value is proportional to the reciprocal of execution time, so the bigger the better. The two columns of $\|Ax-b\|$ display the precision of residual when the convergence of solver stagnates. The "KNC" above the column represents the Knight Corner coprocessor, while the "SNB" represents the dual-socket Sandy-Bridge octacore processors. The "vec" refers to the vectorized version.

	Original Parallel Implementation			Task-Based Model				
Name	KNC	SNB	$\|Ax-b\|$	KNC	KNC vec	SNB	SNB vec	$\|Ax-b\|$
MorseSym1000	1	1.56	$\sim 3.85\times10^{-11}$	2.28	2.28	3.58	3.76	3.62×10^{-11}
MorseSym10000	1	1.47	$\sim 6.36\times10^{-12}$	2.03	2.03	5.25	5.41	5.68×10^{-12}
VEF64000	1	1.41	$\sim 3.94\times10^{-9}$	2.67	3.52	3.94	4.29	3.93×10^{-9}
VEF8000	1	1.94	$\sim 8.07\times10^{-10}$	2.33	2.94	6.40	7.04	8.03×10^{-10}
bcsstk17	1	2.03	$\sim 5.45\times10^{-8}$	2.74	3.69	6.49	7.27	5.23×10^{-8}

The two columns of $\|Ax-b\|$ display the precision of the solution. The reason why there is a "$\sim$" symbol in the fourth column is because the results of the original Monte Carlo solver are nondeterministic. But when we eliminate the uncertainty in the model, the results become stable and reproducible. We may even expect from the new method a better accuracy of solution. And this is not the only improvement on the numerical aspects. In some of our experiments, the convergence of the original solver may stagnate at an early stage while the new method would break the stagnation.

As for the performance, the dual-socket SNB has an advantage over KNC in running both the original and newly proposed solver. The KNC is a large system with 61 interconnected cores. We argue that the reason is due to the amount of work that is insufficient for the KNC. The test matrices are all relatively small that fail to supply the KNC with adequate computations. The work-stealing task scheduling also contributes to the inefficiency of KNC. Both Algorithm 8 and Algorithm 9 define a light-weight task, which means that a frequent task context switching would happen. Even though each task performs a vector instruction, when it is done, the L1 cache is probably partly refreshed with the new task and associated data. The vector instructions can hardly be pipelined at full speed. The frequent task switching may also induce the thread to quickly consume its private work queue and steal from others. The stealing cost is proportional to the distance between the thief and the victim. So being a large system with 61 cores, KNC pays more than SNB. There are other performance issues such as irregular and sparse memory access, data locality that

are common to both KNC and SNB. But since the on-chip bandwidth is more important to KNC [12], these problems may further aggravate the performance on KNC.

In Table 5.4, the 6th and 8th column under "Task-Based Model" that are marked with "vec" are the vectorized performances of the task-based model. The performance of explicitly vectorized code is shown in these two columns. We use directly the intrinsic function [13] to vectorize the code as suggested in Algorithm 11. But the program does not necessarily execute the vectorized version all the times. It makes smart decisions between the scalar and vectorized code depending on whether there is enough data to process. When the number of the elements to be processed is shorter than the vector length, the program uses the scalar code. That explains the reason why the vectorized performance is exactly the same as the non vectorized one for both the matrices "MorseSym1000" and "MorseSym10000". According to Table 5.3, the maximum number of nonzeros per row for these two matrices are 7, which is smaller than 8. The vector processing unit in KNC has 8 SIMD lanes for double precision floating numbers. We can observe in Table 5.4 that more nonzero elements per row leads to better vectorized performance. In SNB, the vector instruction is based on AVX2 (256 bits). It achieves a gain of around 10%. This number in KNC is raised to 30%, which again underlines the importance of vectorization on KNC.

If we take two steps back and look at the whole picture, it is clear that the newly proposed model achieves substantial speedups compared with the original parallel solver. In all the cases, a minimum speedup of 2 can be expected.

In the best case scenario ("bcsstk17" on KNC), the acceleration is 3.69. The improvements come from the efforts that are put into solving the performance issues that are raised in section 5.4. In the original parallel implementation, most of the execution time is spent on the random number generation and binary search. Moreover, the original method walks through the states step by step. Each time a state is being visited only once. In the newly proposed model, there is no more random number generation nor binary search. And the repeated visits to the same state that are located at the same level in a state tree are wrapped in a single task. The update to the running sum is weighted using "*num_visits*". All of the above techniques improve significantly the execution efficiency of the Monte Carlo linear solver, which finally results in a substantial acceleration.

In fact, there is an extra bonus by using the newly proposed model. In the original parallel implementation, the total number of random walks N has to be an integer. When the matrix size grows, N has to be increased accordingly. A very large N may cause overflow when it is

[12] The L2 cache-coherency of KNC costs a part of the bandwidth. Also, the SNB has more cache levels and fewer number of cores than KNC, so it is more tolerant on bandwidth.

[13] The intrinsic functions are C style functions that provide access to many vector instructions, including Intel SSE, AVX, AVX-512, and more, without the need to write assembly code.

out of the limited range of IEEE representation. Whereas in the new model, the number of visits is a floating number that has much wider spectrum of representation than the integer [14].

5.6 Toward a Smart-Tuned Linear Solver

As pointed out before, the main weakness of the Monte Carlo linear solver is that it can not converge to the same precision as do the other classical iterative methods. It is fine for it to be applied in scenarios where an approximate solution is sufficient. But for those that require an accurate solution, it would be of limited help. But we should not give up the parallel value in the modified method. A possible solution is to transform it into a preconditioner [64]. We have already accomplished the first step for it: removing the nondeterministic nature of the original implementation. In that way, the new method is ready for being applied to the classical iterative methods such as conjugate gradient (CG) or generalized minimal residual method (GMRES).

Algorithm 12 The preconditioned conjugate gradient method

Input: Solve $Ax = b$ with the preconditioner matrix M
Output: x_{k+1}

$r_0 := b - Ax_0$
$z_0 := M^{-1} r_0$
$p_0 := z_0$
$k := 0$
repeat
  $\alpha_k := \frac{r_k^T z_k}{p_k^T A p_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k A p_k$
  $z_{k+1} := M^{-1} r_{k+1}$
  $\beta_k := \frac{z_{k+1}^T r_{k+1}}{z_k T r_k}$
  $p_{k+1} := z_{k+1} + \beta_k p_k$
  $k := k + 1$
until r_{k+1} is sufficiently small then exit loop

Algorithm 12 shows the preconditioned conjugate gradient method. It is equivalent to applying the conjugate gradient method without preconditioning to the system:

$$E^{-1} A (E^{-1})^T \hat{x} = E^{-1} b \tag{5.24}$$

[14] If we also use a floating number in the original implementation, the type casting will lead to the loss of precision. Because the scheduler asks for an integer to dispatch the job.

where $EE^T = M$, $\hat{x} = E^T x$. The preconditioner matrix M needs to be fixed. In other words, it cannot change from iteration to iteration. If the assumption on the preconditioner is violated, the behavior of the preconditioned conjugate gradient method may become unpredictable. But we fixed this issue for the Monte Carlo method. So it should be alright for the new Monte Carlo solver to be the M in Algorithm 12. As before, we will work on the sparse matrices. Then the conjugate gradient method will depend extensively on the sparse matrix-vector kernel that we have improved in chapter 4. Once we have a preconditioned conjugate gradient method with the acceleration from both the sparse matrix-vector kernel and task-based Monte Carlo preconditioner, we will proceed towards a smart-tuned numerical solver. Based on the conditioning and structural information of the matrix, it is possible to autotune the type of preconditioner, or the parameter used in a preconditioner, such as the stopping condition for the Monte Carlo method, in order to achieve an efficient adaptive solver.

Chapter 6

Conclusion and Perspective

In this thesis, a range of subjects has been visited, including the expression of parallelism by task or data, task scheduling in shared-memory, shared-memory runtime systems and programming tools (see chapter 2), hardware considerations for parallel computing with a focus on the memory subsystem, heterogeneous systems, and many-core architecture (see chapter 3). We have looked at dense matrix-vector multiplication [27], sparse matrix-vector multiplication [146] (see chapter 4), and the stochastic method for solving linear systems [145], which we have remodeled so as to make it more friendly to parallel computing [144] (see chapter 5). We have also discussed in papers [28, 24, 26] the design of generic algorithms using numerical frameworks like Trilinos, as well as the portability, scalability and efficiency of highly parallel Krylov eigensolver running on supercomputers that consist of many-core accelerators, such as MIC and GPU.

This study has been driven by the real computational needs coming from industrial applications. We have striven to provide solutions for some of the most important problems, such as the solution to the linear systems or to eigenvalue problems. But we are not interested in just solving specific problems that are well defined in an application context. Instead, we place our focus on issues like task scheduling, data locality, memory access, or general-purpose computing kernel and algorithmic design. The theory and implementation of these topics are the cornerstone of scientific computing. In section 6.1 we are going to review the keypoints of this study, and synthesize the rules for programming on many-core architecture. On the conclusion of this study, section 6.2 will give some perspectives for future research. We expect our work to be not only the fulfilment of a research program but also the starting point for further results.

6.1 Synthesis

Many-core architecture and heterogeneous systems is the major trend for future supercomputers. This is reflected by the proportion of such configurations in today's best supercomputers [125]. The fastest supercomputer "Tianhe-2" has just kept its first place for the fifth time in the top500 list [1]. With 16,000 computer nodes, each comprising two Intel Ivy Bridge Xeon processors and three Xeon Phi chips, it represents the world's largest installation of Ivy Bridge and Xeon Phi chips, counting a total of 3,120,000 cores. The main computing power is delivered by the many-core architecture. This is not an accident but it represents the choice pushed by both the microelectronic industry and the scientific community. Some of the hardware reasons for parallel computing that support our view are discussed in chapter 1 and chapter 3.

For the previous reasons, we put our focus in this study on many-core architecture, especially the Intel Many Integrated Core architecture. Generally speaking, any parallel code that works for the multicore processors would automatically be runable on many-core coprocessors at the price of a second compiling. This is because they are all based on the same conceptual model as the CPU (see section 3.1) and share the same ISA (x86). However, a simple compilation is not a magic wand that will allow us to obtain free performance gain. The fact is that the performance will usually drop even though the many-core processor is supposed to deliver more computing power. According to our experience, if the performance is the top concern, then the secret ingredients are nothing but an appropriate programming model and a thorough understanding of the hardware.

A KNC coprocessor serves as a shared-memory system, which can be viewed as a SMP-on-a-chip. Common shared-memory programming interfaces are applicable to KNC, but they follow different philosophies of scheduling tasks. Basically there are two major categories of task scheduling: work-sharing and work-stealing.

Work-sharing assumes the existence of commutativity between the tasks [2] so that they can be executed in any order. This is implemented in OpenMP in terms of loop scheduling: the work defined inside the loop is the basic task that may be affected to any thread.

Work-stealing admits the kinship between two tasks. A task can only have a single parent except for the root task. The child task is spawned by its parent before being placed into the thread-private work queue. A thread without any task simply steals from its "wealthy neighbors". This runtime is implemented in TBB and Cilk/Cilk+. In general, the programmer

[1] http://www.top500.org/featured/top-systems/tianhe-2-milkyway-2-national-university-of-defense/

[2] There is no dependency among the tasks, or the dependency has been taken care of so that the order of execution does not affect the correctness of result.

is in charge of defining the tasks that may spawn a child. The execution is launched with the root task.

Together with the handling of scheduling issues, vectorization is very important as well. The individual core of MIC is weak and simplified in its architecture compared to those of the cutting edge processors. In our experiments with the dense matrix-vector product kernel, the pure multithreading performance is poor, and this is not a problem of scheduling. It is simply because we ignore the powerful vector processing unit that is the computing engine of MIC.

There exists several ways to drive the vector engines. In our experiments we tested different runtime interfaces pairing with dissimilar explicit vectorization methods. For the dense matrix-vector multiplication, the memory access is regular. The per task workload is also sufficient, so the vector instructions can be pipelined at full speed. The main bottleneck comes from the scheduling overhead. Both work-sharing and work-stealing have their downsides. For the first one is the contention for the central queue, for the second one is the stealing cost. By taking the lesser of two evils, we obtain very promising performance on KNC that even outruns the MKL. The best combination of multithreading interface and explicit vectorization method has been proposed for both MIC and host processors.

Then we visited the sparse matrix-vector multiplication. The common manner of parallelizing a program on a supercomputer is to consider the computing chips like MIC as a shared-memory node and apply the previously mentioned multithreading methods to do a portion of work. The inter-node communication is managed by MPI through message passing. This may works just fine for applications like dense matrix-vector multiplication, because the memory access here is not a salient problem, and the scheduling overhead is not notable because the thread always has enough work to do and will not frequently ask for more. However, the sparse matrix-vector multiplication has almost opposite features: its memory access is irregular and sparse; the amount of work for each task can be very variable. We are no longer in a favorable position to consider the KNC as a "SMP-on-a-chip". We have to take into account various factors such as the bandwidth for the cost of maintaining the L2 cache coherency, the contention for the shared data, the memory page and TLB efficiency, etc.

Let us come back to MPI and shared-memory techniques, say OpenMP. MPI creates processes that have separate address spaces while OpenMP manages a pool of threads that share the heap memory space. A process doesn't necessarily correspond to a physical unit: it simply sets aside a zone that is independent of the rest. Data that are allocated within this private area are more local, and there is greater freedom and less overhead to manage the local threads. In fact, we can blur the boundaries defined by physical units: any place

that is big enough to host a process can be used as a separate domain as if it were a logical processor. In this way, the problem becomes whether the partitioning of on-chip domain really optimizes the thread scheduling and data locality.

We practiced this idea on KNC. The task that multiplies the row of matrix by the left-hand side vector x is vectorized using intrinsic functions. By carefully choosing the partitioning, the hybrid MPI/OpenMP obtains significant performance gains compared with the pure multithreading or flat MPI approaches. The hybrid programming indeed mitigates the thread scheduling overhead and increases data locality. With regard to the flat MPI approach, obviously the context switching for a thread is much lighter than for a process, and we have to count on threads to dynamically balance the work. Since it lacks an effective runtime scheduler for processes, the current solution is to use the static scheduling for work distribution among the processes. We then extracted a performance model from the experimental results. The model characterizes the effect of two major performance factors, allows to improve performance predictions. It is also instructive for further optimization or similar applications.

Our study suggests that many-core architecture favors the algorithms that exhibits plentiful parallelism. After all, even if every part of an algorithm is perfectly parallelized, it can still be serialized by synchronization points between different parts. The synchronization amplifies the impact of the slow execution units and degrades the performance with idle waiting. If the algorithm is inherently parallel, the performance would not be compromised by synchronization. More importantly, such algorithms allow the many-core or multi-core architectures to better show their strengths, which is crucial for the future supercomputers.

The most eligible candidate is the Monte Carlo method, widely used in nuclear physics as well as many other scientific domains. When applied to the solution of linear systems, it conducts random walks that form Markov chains. Each random step generates a random number and compares it with a transition probability to determine the next state. At each step, a random variable is updated by the corresponding transition weight; this variable is subsequently added to a running sum. The transition probabilities and weights are derived from the linear system to be solved. The initial probabilities and weights are derived from the right-hand side vector. The random walks have to be repeated a sufficiently large number of times in order for the running sum to converge to the solution. Since each random walk is independent of any other random walk, they can be easily assigned to any execution units.

We designed a parallel Monte Carlo solver. The experiments showed that on both our lab cluster of 90 computing nodes and a workstation of 4 KNC cards, this parallel implementation is scalable. In order to improve the performance of the Monte Carlo parallel solver on many-core architecture, some of the techniques developed in previous study have been implemented,

including the hybrid use of MPI and OpenMP. Only a moderate improvement was observed. However, due to the limitation of the Monte Carlo method, there seems to be no further room for the improvement. The hardware, on the other hand, still has more to offer, especially in the case of MIC. The random walk of Monte Carlo method exposes nothing to vectorization, meaning that it renounces to half of the computing power available on MIC. Besides, the original Monte Carlo method spends too much time on the random number generation and the pairing binary search. The last potential drawback of Monte Carlo method is the non-reproductibility of results caused by its stochastic nature.

In order to address all of the above issues, we proposed a task-based execution model for the Monte Carlo linear solver. It reforms thoroughly the process of random walk. Basically it combines all the states that can be visited at a given level. Instead of visiting one state, the new model visits all the possible states. This change allows the Monte Carlo method to skip the random number generation and the binary search. Updating a group of states instead of a single one also provides us with the room for vectorization. By changing the stopping condition and initial probability setting, we managed to turn the method into a deterministic process. According to our experiments, the new model exceeds the original parallel implementation in every way. We observed substantial improvement in performance and also better numerical properties. As long as the entire matrix fits into the memory, the new model should be as scalable as the old one.

6.2 Future Research

This study answers some of the questions we had chosen to addressed.

The improvements of the efficiency of computing kernels and parallel method has advanced our understanding of multi-core and many-core based high performance computing and we hope they will help in reaching one day exascale applications. However, because of the limited time and resources, our effort could only address a few of the many issues of high performance computing. Admittedly, we do not have the answer for many challenging problems, but maybe we were able to identify some paths for future study.

In the short time of less than a year that we had at our disposal, we have not exhausted all the possibilities of research on Monte Carlo linear solvers: Section 5.6 presents a clear guidance for it.

In within a year, as the KNL will become available, it will be possible to extend the SpMV model proposed in chapter 4 to two dimensions: from KNC to KNL, and from KNL to the cluster of KNL. More details can be found in section 4.5.

In the longer term, more topics can have to be revisited. In my opinion, the mainstream of parallel computing research based on the many-core architecture is really about the mapping between the following 4 elements:

$$\text{Algorithm} \rightarrow \text{vectorized tasks} \rightarrow \text{threads} \rightarrow \text{physical cores} \tag{6.1}$$

The mapping from algorithm to vectorized tasks resembles what we did to the Monte Carlo solver in this thesis. Another noteworthy example is the Tall Skinny QR (TSQR) [40]. Traditional QR factorization is hard to be parallelized. TSQR splits the tall and skinny matrix into many smaller pieces and factorizes independently, thus creating more parallelism. Moreover, some of the operations can be vectorized. In fact, the notion of "task" can be quite resilient. As we described in chapter 2, it can be as small as an instruction, or as big as a subroutine. In Multiple Explicitly Restarted Arnoldi Method (MERAM) [46], each ERAM can be deemed as a task, a relatively heavy-weight one.

The mapping from vectorized tasks to threads implicates the design of task scheduling and runtime system. We mainly focused on two scheduling policies in this study: the work-sharing and work-stealing. But they both have their own restrictions. The work-sharing requires the tasks to be commutative, which means they can hardly have any dependencies among them. The work-stealing model is built on the fact that each task spawns its own child, which means a task can only have one parent. For more complicated representation of tasks, they can surely have more than one parent. The kinship here is actually the synonym of data dependency. In this case, a scheduler can be implemented to resolve the dependencies between tasks according to their input and output data. The resolution of dependency will take extra time, but it is the price to pay for this approach. We said before that the classical numerical methods may require synchronization points which degrade the performance. Consider the Cholesky-based matrix inversion [2, 78]. Conventionally, it is decomposed into 3 subroutines; even if each subroutine is perfectly parallelized, the synchronous parallel execution is still suboptimal, as shown in Figure 6.1. However, the task scheduling based on the graph resolving can perfectly overlap the tasks that have no dependencies between them and achieve a more balanced, asynchronous execution, as shown in Figure 6.2. What we learn from this case is that a more complicated scheduler will help the classical numerical methods to achieve a more efficient execution at the cost of extra computation. It would not be possible if it were not for the DAG-based scheduler.

In Chapter 4, the hybrid use of MPI processes and OpenMP threads creates the possibility for hierarchical scheduling that alleviates the scheduling overhead and increase the data locality. But the hybrid MPI/OpenMP model is not the only mean to achieve efficiency. An alternative solution lies in designing a runtime system that supports the scheduling by groups

of threads. For example, the inter-group scheduling may use the work-stealing model while within the thread group they may share the local task queue. In this way both contention and stealing overhead can be reduced. However, this approach still has a drawback, which is not the case in the hybrid MPI/OpenMP model. There is no group-private data that are only shared within the group: the data item is either shared by all the threads or private to one.

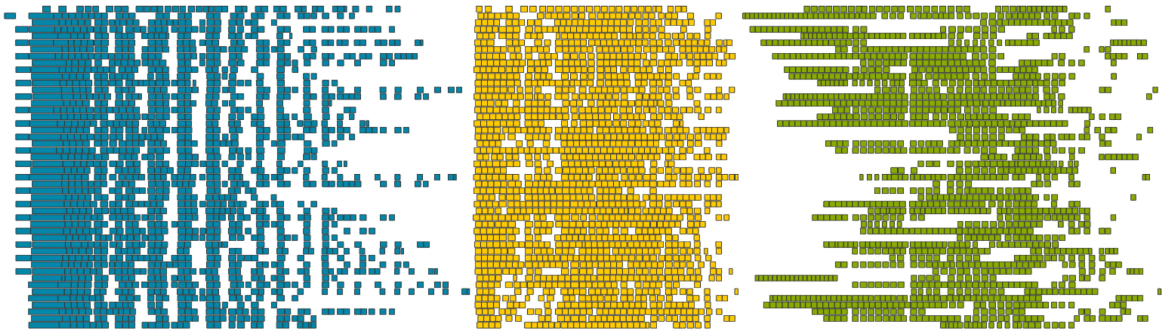


Figure 6.1 The task scheduling of the Cholesky-based matrix inversion that is implemented with synchronization points.

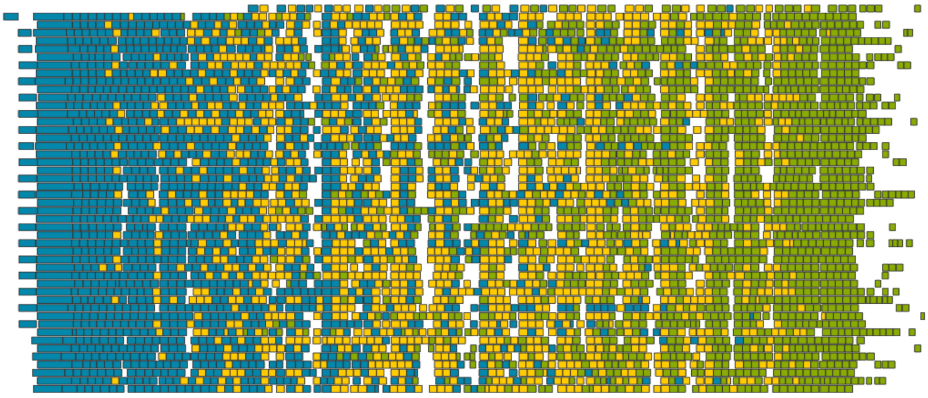


Figure 6.2 The task scheduling of the Cholesky-based matrix inversion that is implemented without synchronization points.

Finally, the mapping from threads to physical cores also forms a part of the runtime system, and it can be affected by the operating system as well. In our study, we usually bind the threads to the physical cores to achieve better cache usage. Executing the instructions from the same context is more likely to hit the cache when it is still "hot". In this case, there is one fewer level of abstraction. But sometimes, the dynamic mapping of threads to cores is necessary, especially when the hardware is shared by multiple applications.

Before the final concluding remarks, we would like to discuss the programming languages and tools for high performance computing. In this study, we specifically restricted our

programming tools to those that are widely used and publicly available, such as OpenMP, Cilk/Cilk+, TBB, MPI, intrinsic functions, etc. This was for the purpose of generality and re-usability. In fact, there are a variety of thread implementations: some low level instances include the C++ standard thread library, the Boost thread library, Microsoft Windows thread API, POSIX threads (or Pthreads), etc. These are primordial threading APIs without runtime scheduling support. They are not easy to use and sometimes are platform dependent. But they can be treated as the starting point for a self-defined runtime library.

Our work was done with C and C++: hey are efficient programming languages that are often used in performance concerned code. In the numerical community, Fortran also retains its vitality in spite of its seniority and it can be found in many legacy codes that are still in use. That is one reason for its popularity; another reason is credited to its upgrading. There is a recent version that supports coarrays (Coarray Fortran, a.k.a CAF). The CAF is capable of defining distributed arrays assuming that the global memory address space is logically partitioned. The hardware-specific data locality can be modeled in the partitioning of the address space. CAF has been qualified as "partitioned global address space" (PGAS) language. The same parallel programming model can be found in Unified Parallel C (UPC) from Berkeley, Fortress from Sun, Chapel from Cray, X10 from IBM, Global Arrays (GA) from Pacific Northwest National Laboratory, etc. As stated before, the runtime system is one way to accomplish the mapping from vectorized tasks to threads. The PGAS language is an alternative high-level technique for it. The current PGAS languages rely on the compilers to distribute the data and may be implemented on top of a MPI library. They offer the ease of writing parallel programs, but the performance still needs to be improved.

There is another technique that might be of great value in the future high performance computing: that is the template metaprogramming (TMP). This refers to use of the C++ template system to perform computation at compile-time within the code. C++'s template system is Turing-complete; in principle, it is capable of computing anything computable. A typical example would be the compile-time loop unrolling. The TMP can be used to create vector classes that take the vector length as the template parameter and perform the computations on vector using `for` loop. Since the vector length, as the template parameter, is a constant at compile time, the compiler should be able to optimize the code by unrolling the `for` loop. The TBB that was used in this study is a C++ template library. We believe that there is more to expect from the C++ template system.

All of the above discussions target one goal, which is the exascale computing. As the day comes, we will see more advanced and smart methods that run on the more powerful computers. It is our honor and duty to make it happen.

Bibliography

[1] Adams, M. L. and Larsen, E. W. (2002). Fast Iterative Methods for Discrete-Ordinates Particle Transport Calculations. *Progress in Nuclear Energy*, 40(1):3–159.

[2] Agullo, E., Bouwmeester, H., Dongarra, J., Kurzak, J., Langou, J., and Rosenberg, L. (2011). Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. In Palma, J., Daydé, M., Marques, O., and Lopes, J. a., editors, *High Performance Computing for Computational Science - VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 129–138. Springer Berlin Heidelberg.

[3] Akbudak, K., Kayaaslan, E., and Aykanat, C. (2012). Hypergraph-partitioning-based models and methods for exploiting cache locality in sparse-matrix vector multiplication. Technical report, Bilkent University.

[4] Alexandrov, V. N. (1998). Efficient parallel monte carlo methods for matrix computations. *Mathematics and Computers in Simulation*, 47:113–122.

[5] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computing Conference*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C.

[6] Anderson, M., Ballard, G., Demmel, J., and Keutzer, K. (2011). Communication-avoiding qr decomposition for gpus. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 48–58.

[7] Aquilanti, P.-Y. (2011). Méthodes de krylov réparties et massivement parallèles pour la résolution de problèmes de géoscience sur machines hétérogènes dépassant le pétaflop. Ph.D. dissertation from University of Lille 1.

[8] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198.

[9] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. (1997). Efficient management of parallelism in object oriented numerical software libraries. In Arge, E., Bruaset, A. M., and Langtangen, H. P., editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press.

[10] Ballard, G., Demmel, J., Grigori, L., Jacquelin, M., Nguyen, H. D., and Solomonik, E. (2014). Reconstructing householder vectors from tall-skinny qr. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1159–1170.

[11] Baudron, A.-M. and Lautard, J.-J. (2007). Minos: a simplified pn solver for core calculation. *Nuclear Science and Engineering*, 155(2):250–263.

[12] Bell, N. and Garland, M. (2008). Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation.

[13] Bell, N. and Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11.

[14] Berrendorf, R. and Nieken, G. (2000). Performance characteristics for openmp constructs on different parallel computer architectures. *Concurrency - Practice and Experience*, 12(12):1261–1273.

[15] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA. ACM.

[16] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748.

[17] Boillod-Cerneux, F. (2014). Nouveaux algorithmes numériques pour l'utilisation efficace des architectures de calcul multi-cœurs et hétérogènes. Ph.D. dissertation from University of Lille 1.

[18] Brémaud, P. (1999). *Markov chains : Gibbs fields, Monte Carlo simulation and queues*. Texts in applied mathematics. Springer, New York, Berlin, Heidelberg.

[19] Brinskiy, M., Lubin, M., and Dinan, J. (2015). Chapter 16 - mpi-3 shared memory programming introduction. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 305 – 319. Morgan Kaufmann, Boston.

[20] Brock, D. C., editor (2007). *Understanding Moore's Law: Four Decades of Innovation*, volume 98. The University of Chicago Press.

[21] Brown, F. B. (2014). New hash-based energy lookup algorithm for monte carlo codes. Technical report, LA-UR-14-24530. Los Alamos National Laboratory.

[22] Brueckner, R. (2015). China May Develop Two 100 Petaflop Machines Within a Year. http://insidehpc.com/2015/08/china-may-develop-two-100-petaflop-machines-within-a-year/. [uploaded 26-August-2015].

[23] Buluç, A., Williams, S., Oliker, L., and Demmel, J. (2011). Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*.

[24] Calvin, C., Boillod-Cerneux, F., Ye, F., Galicher, H., and Petiton, S. (2014). Programming Paradigms for Emerging Architectures Applied to Asynchronous Krylov Eigensolver. In *SIAM-PP 2014 - SIAM Conference on Parallel Processing For Scientific Computing, Portland, Oregon, February 18-21, 2014*.

[25] Calvin, C., Cueto, O., and Emonot, P. (2002). An object-oriented approach to the design of fluid mechanics software. *ESAIM: Mathematical Modelling and Numerical Analysis - Modélisation Mathématique et Analyse Numérique*, 36(5):907–921.

[26] Calvin, C., Petiton, S., and Ye, F. (2013a). Krylov Basis Orthogonalization Algorithms on Many Core Architectures. In *SIAM Annual Meeting, San Diego, California, July 8-12, 2013*.

[27] Calvin, C., Ye, F., and Petiton, S. G. (2013b). The exploration of pervasive and fine-grained parallel model applied on intel xeon phi coprocessor. In Xhafa, F., Barolli, L., Nace, D., Venticinque, S., and Bui, A., editors, *3PGCIC*, pages 166–173. IEEE.

[28] Calvin, C., Petiton, S., Ye, F., and Boillod-Cerneux, F. (2014). Efficient and portable krylov eigensolver on many core architectures. In *SNA + MC 2013 - Joint International Conference on Supercomputing in Nuclear Applications + Monte Carlo*, page 04104.

[29] Cao, C., Herault, T., Bosilca, G., and Dongarra, J. (2015). Design for a soft error resilient dynamic task-based runtime. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 765–774.

[30] Cappello, F. and Etiemble, D. (2000). Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Supercomputing '00, Washington, DC, USA. IEEE Computer Society.

[31] Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

[32] Chen, Wai-Kai, . (2003). *Memory, microprocessor, and ASIC*. Boca Raton : CRC Press. Includes bibliographical references and index.

[33] Choi, J., Singh, A., and Vuduc, R. W. (2010). Model-driven autotuning of sparse matrix-vector multiply on gpus. In Govindarajan, R., Padua, D. A., and Hall, M. W., editors, *PPOPP*, pages 115–126. ACM.

[34] Chow, E. and Hysom, D. (2001). Assessing performance of hybrid mpi/openmp programs on smp clusters. Technical report, Lawrence Livermore National Laboratory.

[35] Cramer, T., Schmidl, D., Klemm, M., and an Mey, D. (2012). Openmp programming on intel xeon phi coprocessors: An early performance comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44.

[36] Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55.

[37] Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25.

[38] Daydé, M. J. and Duff, I. (1997). The use of computational kernels in full and sparse linear solvers, efficient code design on high-performance risc processors. In Palma, J. and Dongarra, J., editors, *Vector and Parallel Processing – VECPAR'96*, volume 1215 of *Lecture Notes in Computer Science*, pages 108–139. Springer Berlin Heidelberg.

[39] Demmel, J., Dongarra, J., Fox, A., Williams, S., Volkov, V., and Yelick, K. (2009). Accelerating time-to-solution for computational science and engineering. *SciDAC REVIEW*, pages 46–57.

[40] Demmel, J., Grigori, L., Hoemmen, M., and Langou, J. (2012). Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239.

[41] Dimov, I. T. and Karaivanova, A. N. (1999). A power method with monte carlo iterations. *Recent Advances in Numerical Methods and Appl.*, II:239–247.

[42] Dongarra, J. J., Faverge, M., Ltaief, H., and Luszczek, P. (2013). Achieving Numerical Accuracy and High Performance using Recursive Tile LU Factorization. *Concurrency and Computation: Practice and Experience*.

[43] Dubois, J. (2011). Contribution à l'algorithmique et à la programmation efficace des nouvelles architectures parallèles comportant des accélérateurs de calcul dans le domaine de la neutronique et de la radioprotection. Ph.D. dissertation from University of Lille 1.

[44] Duran, A., Corbalán, J., and Ayguadé, E. (2008). Evaluation of openmp task scheduling strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg. Springer-Verlag.

[45] Eisenlohr, J., Hudak, D., Tomko, K., and Prince, T. (2012). Dense linear algebra factorization in OpenMP and Cilk Plus on Intel MIC: Development experiences and performance analysis. In *TACC-Intel Highly Parallel Computing Symp.*

[46] Emad, N., Petiton, S., and Edjlali, G. (2005). Multiple explicitly restarted arnoldi method for solving large eigenproblems. *SIAM Journal on Scientific Computing*, 27(1):253–277.

[47] Feng, W. C. and Cameron, K. W. (2015). The Green500 List - June 2015. http://www.green500.org/news/green500-list-june-2015. [published June-2015].

[48] Flynn, M. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909.

[49] Forsythe, G. E. and Leibler, R. A. (1950). Matrix inversion by a monte carlo method. *Math. Comp.*, 4(31):127–127.

[50] Forum, M. P. I. (2012). MPI: A message-passing interface standard version 3.0.

[51] Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223.

[52] Fujita, N., Fujii, H., Hanawa, T., Kodama, Y., Boku, T., Kuramashi, Y., and Clark, M. (2014). Qcd library for gpu cluster with proprietary interconnect for gpu direct communication. In Lopes, L., Žilinskas, J., Costan, A., Cascella, R., Kecskemeti, G., Jeannot, E., Cannataro, M., Ricci, L., Benkner, S., Petit, S., Scarano, V., Gracia, J., Hunold, S., Scott, S., Lankes, S., Lengauer, C., Carretero, J., Breitbart, J., and Alexander, M., editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 251–262. Springer International Publishing.

[53] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA.

[54] Gropp, W., Hoefler, T., Thakur, R., and Lusk, E. (2014a). *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press.

[55] Gropp, W., Lusk, E., and Skjellum, A. (2014b). *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* 3rd edition. MIT Press.

[56] Grosch, H. (1991). *Computer: Bit Slices from a Life*. Third Millenium Books.

[57] Guttman, D., Arunachalam, M., Calina, V., and Kandemir, M. T. (2015). Chapter 21 - prefetch tuning optimizations. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 401 – 419. Morgan Kaufmann, Boston.

[58] Haidar, A., Luszczek, P., Tomov, S., and Dongarra, J. (2014). Heterogenous acceleration for linear algebra in multi-coprocessor environments. In *High Performance Computing for Computational Science - VECPAR 2014 - 11th International Conference, Eugene, OR, USA, June 30 - July 3, 2014, Revised Selected Papers*, pages 31–42.

[59] Heinecke, A., Klemm, M., Pflüger, D., Bode, A., and Bungartz, H.-J. (2011). Extending a highly parallel data mining algorithm to the intel ® many integrated core architecture. In Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Martino, B. D., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S. L., Traff, J. L., Vallée, G., and Weidendorfer, J., editors, *Euro-Par Workshops (2)*, volume 7156 of *Lecture Notes in Computer Science*, pages 375–384. Springer.

[60] Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[61] Hentschel, B., Göbbert, J. H., Klemm, M., Springer, P., Schnorr, A., and Kuhlen, T. W. (2015). Packet-oriented streamline tracing on modern SIMD architectures. In *Eurographics Symposium on Parallel Graphics and Visualization, Cagliari, Sardinia, Italy, May 25 - 26, 2015.*, pages 43–52.

[62] Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[63] Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and Williams, A. (2003). An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories.

[64] Heroux, M. A., Vu, P., and Yang, C. (1991). A parallel preconditioned conjugate gradient package for solving sparse linear systems on a cray y-mp. *Appl. Num. Math*, 8:93–115.

[65] Hildebrand, B. F. (1987). *Introduction to Numerical Analysis: 2Nd Edition*. Dover Publications, Inc., New York, NY, USA.

[66] Holmen, J., Humphrey, A., and Berzins, M. (2015). Chapter 13 - exploring use of the reserved core. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 229 – 242. Morgan Kaufmann, Boston.

[67] HUGUES, M. (2011). Un paradigme de programmation multi-niveaux pour le calcul numérique sur les machines post-petascales et exascales. Ph.D. dissertation from University of Lille 1.

[68] Im, E.-J. and Yelick, K. (2001). Optimizing sparse matrix computations for register reuse in sparsity. In *Proc. of ICCS*, pages 127–136.

[69] Im, E.-J., Yelick, K., and Vuduc, R. (2004). Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158.

[70] Intel (2009). An introduction to the Intel QuickPath Interconnect.

[71] Intel (2012). Intel® Many Integrated Core Architecture - Advanced. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. [online material].

[72] Jakovits, P., Kromonov, I., and Srirama, S. (2011). Monte carlo linear system solver using mapreduce. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 293–299.

[73] Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[74] Jesshope, C. and Egan, C. (2006). *Advances in Computer Systems Architecture: 11th Asia-Pacific Conference, ACSAC 2006, Shanghai, China, September 6-8, 2006, Proceedings*. LNCS sublibrary: Theoretical computer science and general issues. Springer.

[75] Koesterke, L., Boisseau, J., Cazes, J., Milfeld, K., and Stanzione, D. (2011). Early experiences with the intel many integrated cores accelerated computing technology. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, pages 21:1–21:8, New York, NY, USA. ACM.

[76] Kourtis, K., Goumas, G., and Koziris, N. (2010). Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Trans. Architec. Code Optim.*, 7(3):16:1–16:31.

[77] Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Basermann, A., and Bishop, A. R. (2012). Sparse matrix-vector multiplication on gpgpu clusters: A new storage format and a scalable implementation. In *IPDPS Workshops*, pages 1696–1702. IEEE Computer Society.

[78] Krishnamoorthy, A. and Menon, D. (2011). Matrix inversion using cholesky decomposition.

[79] Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471.

[80] Leppänen, J. (2009). Two practical methods for unionized energy grid construction in continuous-energy monte carlo neutron transport calculation. *Annals of Nuclear Energy*, 36(7):878–885.

[81] Lewis, E. E. and Miller, W. F. (1984). *Computational methods of neutron transport*. New York : Wiley. "A Wiley-Interscience publication.".

[82] Li, S. and Newman, C. (2015). A structured performance optimization framework for simultaneous heterogeneous computing. Technical report, Software and Services Group, Intel Corporation, Hillsboro, Oregon.

[83] Li, Y., Dongarra, J., and Tomov, S. (2009). A note on auto-tuning gemm for gpus. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg. Springer-Verlag.

[84] Liu, X., Smelyanskiy, M., Chow, E., and Dubey, P. (2013). Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA. ACM.

[85] Maison de la Simulation (2015). Description of the cluster "poincaré". https://groupes.renater.fr/wiki/poincare/public/description_de_poincare.

[86] Marčuk, G. and Lebedev, V. (1986). *Numerical Methods in the Theory of Neutron Transport*. Harwood Academic Publishers.

[87] Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30.

[88] McCool, M., Reinders, J., and Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[89] Mellor-Crummey, J. and Garvin, J. (2004). Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl.*, 18(2):225–236.

[90] Monakov, A. and Avetisyan, A. (2009). Implementing blocked sparse matrix-vector multiplication on nvidia gpus. In Bertels, K., Dimopoulos, N., Silvano, C., and Wong, S., editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657 of *Lecture Notes in Computer Science*, pages 289–297. Springer Berlin Heidelberg.

[91] Monakov, A., Lokhmotov, A., and Avetisyan, A. (2010). Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 111–125, Berlin, Heidelberg. Springer-Verlag.

[92] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[93] Nath, R., Tomov, S., Dong, T. T., and Dongarra, J. (2011a). Optimizing symmetric dense matrix-vector multiplication on gpus. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 6:1–6:10, New York, NY, USA. ACM.

[94] Nath, R., Tomov, S., and Dongarra, J. (2011b). Accelerating gpu kernels for dense linear algebra. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 83–92, Berlin, Heidelberg. Springer-Verlag.

[95] Neelima, B., Reddy, G., and Raghavendra, P. (2014). Predicting an optimal sparse matrix format for spmv computation on gpu. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1427–1436.

[96] Newman, M. and Todd, J. (1958). The evaluation of matrix inversion programs. *Journal of the Society for Industrial and Applied Mathematics*, 6(4):pp. 466–476.

[97] Nishtala, R., Vuduc, R. W., Demmel, J. W., and Yelick, K. A. (2007). When cache blocking of sparse matrix vector multiply works and why. *Appl. Algebra Eng., Commun. Comput.*, 18(3):297–311.

[98] NVIDIA (2014a). Nvidia nvilink high-speed interconnect: Application performance. Technical report. (Whitepaper).

[99] NVIDIA (2014b). Summit and sierra supercomputers: An inside look at the u.s. department of energy's new pre-exascale systems. Technical report. (Whitepaper).

[100] O'Brien, E. (2015). Chapter 17 - coarse-grained openmp for scalable hybrid parallelism. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 321 – 334. Morgan Kaufmann, Boston.

[101] Oliker, L., Li, X., Husbands, P., and Biswas, R. (2002). Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Rev.*, 44(3):373–393.

[102] OpenMP Architecture Review Board (2013). OpenMP application program interface version 4.0.

[103] Patterson, D. A. and Hennessy, J. L. (2008). *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition.

[104] Petiton, S. and Emad, N. (1996). A data parallel scientific computing introduction. In Perrin, G.-R. and Darte, A., editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 45–64. Springer Berlin Heidelberg.

[105] Petiton, S. G. (1992). Parallel subspace method for non-hermitian eigenproblems on the connection machine (cm2). *Appl. Numer. Math.*, 10(1):19–35.

[106] Pinar, A. and Heath, M. (1999). Improving performance of sparse matrix-vector multiplication. In *Proc. ACM/IEEE Supercomputing*.

[107] Potluri, S., Tomko, K., Bureddy, D., and Panda, D. (2012). Intra-mic mpi communication using mvapich2: Early experience. In *TACC-Intel Highly-Parallel Computing Symposium*.

[108] Pérez, J. M., Badia, R. M., and Labarta, J. (2008). A dependency-aware task-based programming environment for multi-core architectures. In *CLUSTER*, pages 142–151. IEEE Computer Society.

[109] R. Nath, S. T. and Dongarra, J. (2010). *Blas for GPUs, Scientific Computing with Multicore and Accelerators*. Chapman Hall/CRC Computational Science, 1 edition.

[110] Raskulinec, G. M. and Fiksman, E. (2015). Chapter 22 - {SIMD} functions via openmp. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 421 – 440. Morgan Kaufmann, Boston.

[111] Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition.

[112] ROSCA, N. (2006). Monte carlo methods for systems of linear equations. *Studia Univ. BABES-BOLYAI Mathematica*, LI.

[113] Russell, R. M. (1978). The cray-1 computer system. *Commun. ACM*, 21(1):63–72.

[114] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition.

[115] Saad, Y. (2011). *Numerical methods for large eigenvalue problems*. SIAM.

[116] Saini, S., Chang, J., and Jin, H. (2014). Performance evaluation of the intel sandy bridge based nasa pleiades using scientific and engineering applications. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pages 25–51. Springer.

[117] Saule, E. and Catalyurek, U. (2012). An early evaluation of the scalability of graph algorithms on the intel mic architecture. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1629–1639.

[118] Saule, E., Kaya, K., and Çatalyürek, Ü. V. (2013). Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. *CoRR*, abs/1302.1078.

[119] Scheben, F. (2011). Iterative methods for criticality computations in neutron transport theory. Ph.D. dissertation from University of Bath.

[120] Schmidl, D., Terboven, C., Wolf, A., an Mey, D., and Bischof, C. (2010). How to scale nested openmp applications on the scalemp vsmp architecture. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 29–37.

[121] Seaton, M., Mason, L., Matveev, Z. A., and Blair-Chappell, S. (2015). Chapter 23 - vectorization advice. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 441 – 462. Morgan Kaufmann, Boston.

[122] Souza, P., Borges, L., Andreolli, C., and Thierry, P. (2015). Chapter 24 - portable explicit vectorization intrinsics. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 463 – 485. Morgan Kaufmann, Boston.

[123] Srinivasan, A. and Aggarwal, V. (2009). Stochastic linear solvers. In *SIAM Linear Algebra Proceedings*.

[124] Stallings, W. (1996). *Computer Organization and Architecture (4th Ed.): Designing for Performance*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[125] Strohmaier, E., Dongarra, J., Simon, H., Meuer, M., and Meuer, H. (2015). Top500 lists June 2015. http://www.top500.org/lists/2015/06/. [published June-2015].

[126] Sullivan, J. (2014). New U.S. Supercomputers To Break 100 Petaflop Benchmark, Use Nvidia NVlink. http://www.tomsitpro.com/articles/nvidia-ibm-supercomputers,1-2346.html. [uploaded 14-November-2014].

[127] Terboven, C., an Mey, D., Schmidl, D., Jin, H., and Reichstein, T. (2008). Data and thread affinity in openmp programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, pages 377–384, New York, NY, USA. ACM.

[128] Tomov, S., Nath, R., Ltaief, H., and Dongarra, J. (2010). Dense linear algebra solvers for multicore with gpu accelerators. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW),*, pages 1–8.

[129] Trader, T. (2014). Fujitsu Targets 100 Petaflops Supercomputing. http://www.hpcwire.com/2014/08/12/fujitsu-targets-100-petaflops-supercomputing/. [uploaded 12-August-2014].

[130] Vázquez, F., Fernández, J. J., and Garzón, E. M. (2011). A new approach for sparse matrix vector product on nvidia gpus. *Concurr. Comput. : Pract. Exper.*, 23(8):815–826.

[131] Vladimirov, A. (2013a). Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors. Technical report, Colfax Research.

[132] Vladimirov, A. (2013b). Test-driving intel xeon phi coprocessors with a basic n-body simulation. Technical report, Colfax Research.

[133] Vladimirov, A., Asai, R., and Karpusenko, V. (2015). *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors,* 2nd edition. Colfax International.

[134] Volkov, V. and Demmel, J. (2008). Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11.

[135] von Neumann, J. (1993). First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75.

[136] Vuduc, R. W. and Moon, H.-J. (2005). Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of the First International Conference on High Performance Computing and Communications*, HPCC'05, pages 807–816, Berlin, Heidelberg. Springer-Verlag.

[137] Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., and Wang, Y. (2014). *High-Performance Computing on the Intel Xeon Phi(TM): How to Fully Exploit MIC Architectures*. Springer Publishing Company, Incorporated.

[138] Wienke, S., Plotnikov, D., an Mey, D., Bischof, C., Hardjosuwito, A., Gorgels, C., and Brecher, C. (2011). Simulation of bevel gear cutting with gpgpusâperformance and productivity. *Computer Science - Research and Development*, 26(3-4):165–174.

[139] Willcock, J. and Lumsdaine, A. (2006). Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 307–316, New York, NY, USA. ACM.

[140] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA. ACM.

[141] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.

[142] Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., and Dongarra, J. (2015). Hierarchical dag scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 156–165.

[143] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24.

[144] Ye, F., Calvin, C., and Petiton, S. (2015a). An efficient task-based execution model for stochastic linear solver on multi-core and many-core systems. In *18th IEEE International Conference on Computational Science and Engineering - CSE 2015 - Porto, Portugal, October 21 - October 23, 2015, Revised Selected Papers*.

[145] Ye, F., Calvin, C., and Petiton, S. (2015b). Intel Xeon/Xeon Phi platform oriented scalable Monte Carlo Linear Solver. In *M&C + SNA + MC 2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method, Nashville, Tennessee, April 19–23, 2015*.

[146] Ye, F., Calvin, C., and Petiton, S. (2015c). A study of spmv implementation using mpi and openmp on intel many-core architecture. In Daydé, M., Marques, O., and Nakajima, K., editors, *High Performance Computing for Computational Science – VECPAR 2014*, volume 8969 of *Lecture Notes in Computer Science*, pages 43–56. Springer International Publishing.

[147] Yzelman, A. N. and Bisseling, R. H. (2011). Two-dimensional cache-oblivious sparse matrix-vector multiplication. *Parallel Computing*, 37(12):806–819.