PARALLEL PROGRAMMING AND OPTIMIZATION WITH INTEL® XEON PHI™ COPROCESSORS

HANDBOOK ON THE DEVELOPMENT AND OPTIMIZATION OF PARALLEL APPLICATIONS FOR INTEL® XEON® PROCESSORS AND INTEL® XEON PHI™ COPROCESSORS

SECOND EDITION

COLFAX INTERNATIONAL

ANDREY VLADIMIROV | RYO ASAI | VADIM KARPUSENKO

PARALLEL PROGRAMMING AND OPTIMIZATION WITH INTEL® XEON PHI™ COPROCESSORS

HANDBOOK ON THE DEVELOPMENT AND OPTIMIZATION
OF PARALLEL APPLICATIONS
FOR INTEL® XEON® PROCESSORS
AND INTEL® XEON PHI™ COPROCESSORS

Second Edition

Andrey Vladimirov, Ryo Asai and Vadim Karpusenko

Electronic book built: January 4, 2019
Last revision date: January 4, 2019

ISBN: 978-0-9885234-2-5

About the Authors

Andrey Vladimirov, PhD, is Head of HPC Research at Colfax International. His primary interest is the application of modern computing technologies to computationally demanding scientific problems. Prior to joining Colfax, A. Vladimirov was involved in computational astrophysics research at Stanford University, North Carolina State University, and the Ioffe Institute in Russia, where he studied cosmic rays, collisionless plasmas and the interstellar medium using computer simulations.

Ryo Asai is a Researcher at Colfax International. He develops optimization methods for scientific applications targeting emerging parallel computing platforms, computing accelerators and interconnect technologies. Ryo holds a B.S. degree in Physics from University of California, Berkeley.

Vadim Karpusenko, PhD, is Principal HPC Research Engineer at Colfax International involved in training and consultancy projects on data mining, software development and statistical analysis of complex systems. His research interests are in the area of physical modeling with HPC clusters, highly parallel architectures, and code optimization. Vadim holds a PhD from North Carolina State University for his research in in the field of computational biophysics on the free energy and stability of helical secondary structures of proteins.

Additional publications by these authors
related to Intel MIC architecture programming
may be found at
http://colfaxresearch.com/

Acknowledgements

Second Edition

We cannot thank enough the people who have contributed their valuable time and expertise to write technical reviews of the 2nd edition of this book. They have provided guidance, fixed misconceptions, future-proofed the messages and caught countless bugs: **Ilya Burylov, Gennady Fedorov, Alexandr Kalinkin, Alexandr Kobotov, Vadim Pirogov** (Intel/MKL), **Joseph Curley** (Intel), **Rob Farber** (TechEnablement.com), **Rakesh Krishnaiyer** (Intel), **Lawrence Meadows** (Intel), **John Pennycook** (Intel), **Troy Porter** (Stanford University), **Frances Roth** (Intel), **Jason Sewall** (Intel), **Gergana Slavova** (Intel). Thank you all very much!

First Edition

Authors are sincerely grateful to **James Reinders** for supervising and directing the creation of this book, **Albert Lee** for his help with editing and error checking, to specialists at Intel Corporation who contributed their time and shared with the authors their expertise on the MIC architecture programming: **Bob Davies**, **Shannon Cepeda**, **Pradeep Dubey**, **Ronald Green**, **James Jeffers**, **Taylor Kidd**, **Rakesh Krishnaiyer**, **Chris (CJ) Newburn**, **Kevin O'Leary**, **Zhang Zhang**, and to a great number of people, mostly from Colfax International and Intel, who have ensured that gears were turning and bits were churning during the production of the book, including **Rajesh Agny**, **Mani Anandan**, **Joe Curley**, **Roger Herrick**, **Richard Jackson**, **Mike Lafferty**, **Thomas Lee**, **Belinda Liviero**, **Gary Paek**, **Troy Porter**, **Tim Puett**, **John Rinehimer**, **Gautam Shah**, **Manish Shah**, **Bruce Shiu**, **Jimmy Tran**, **Achim Wengeler**, and **Desmond Yuen**.

1 Introduction 1
1.1 Intel Xeon Phi Coprocessors . . . 2
1.2 MIC Architecture: Developer's Perspective . . . 13
1.3 Applicability of the MIC Architecture . . . 30
1.4 Preparing for Future Parallel Architectures . . . 39
1.5 System Administration with Intel Xeon Phi Coprocessors . . . 46

2 Programming Models 87
2.1 Native Applications and MPI . . . 88
2.2 Explicit Offload Model . . . 101
2.3 Shared Virtual Memory Model . . . 119
2.4 Using Multiple Coprocessors . . . 132
2.5 Offload Programming with OpenMP 4.0 . . . 148

3 Expressing Parallelism 153
3.1 Data Parallelism (Vectorization) . . . 154
3.2 Task Parallelism in Shared Memory: OpenMP . . . 186
3.3 Task Parallelism with Intel Cilk Plus . . . 212
3.4 Process Parallelism in Distributed Memory with MPI . . . 229

4 Optimizing Parallel Applications 261
4.1 Optimization Roadmap for Intel Xeon Phi Coprocessors . . . 261
4.2 Scalar and General Optimizations . . . 267
4.3 Optimizing Vectorization . . . 289
4.4 Optimization of Multi-Threading . . . 311
4.5 Memory Access Optimization . . . 356
4.6 Offload Traffic Control . . . 387
4.7 Optimization Strategies for MPI Applications . . . 396

5 Software Development Tools 427
5.1 Intel Math Kernel Library . . . 427
5.2 Intel VTune Amplifier XE . . . 444

6 Summary and Resources 465
6.1 Parallel Programming and Intel Xeon Phi Coprocessors . . . 465
6.2 Supplementary Code for Practical Exercises ("Labs") . . . 467
6.3 Colfax Developer Training . . . 470
6.4 Additional Resources . . . 471

Bibliography 475

Contents

1 Introduction . . . **1**
1.1 Intel Xeon Phi Coprocessors . . . 2
1.1.1 Technology Overview . . . 2
1.1.2 Conventional Programming, Portable Code . . . 4
1.1.3 Heterogeneous Computing and Clustering . . . 7
1.1.4 Intel Xeon Phi Product Family . . . 8
1.1.5 Intel Xeon Processor E3, E5 and E7 Family . . . 11
1.2 MIC Architecture: Developer's Perspective . . . 13
1.2.1 Knights Corner Die Organization . . . 13
1.2.2 Core Specifications . . . 15
1.2.3 Memory Hierarchy and Cache Properties . . . 17
1.2.4 Integration into the Host System through MPSS . . . 20
1.2.5 Networking with Coprocessors in Clusters . . . 22
1.2.6 File I/O on Coprocessors . . . 24
1.2.7 Common Software Development Tools . . . 25
1.2.8 Intel Xeon Processors versus Intel Xeon Phi Coprocessors: Developer Experience . . . 28
1.3 Applicability of the MIC Architecture . . . 30
1.3.1 Task Parallelism . . . 30
1.3.2 Data-Parallel Component . . . 32
1.3.3 Memory Access Pattern . . . 34
1.3.4 PCIe Bandwidth Considerations . . . 36
1.4 Preparing for Future Parallel Architectures . . . 39
1.4.1 Exascale Computing for the Rest of Us . . . 39
1.4.2 Second Generation MIC Processor, KNL . . . 41
1.4.3 Future-Proof Development Options . . . 44
1.5 System Administration with Intel Xeon Phi Coprocessors . . . 46
1.5.1 Hardware Compatibility . . . 46
1.5.2 Operating Systems . . . 47
1.5.3 Installation and Minimal Configuration of MPSS . . . 48
1.5.4 Controlling the MPSS service . . . 49
1.5.5 Integration of MPSS with InfiniBand: OFED . . . 50

1.5.6 Restoring MPSS Functionality after Kernel Updates 51
1.5.7 Installation of Intel Compilers 52
1.5.8 Installing the OpenCL Runtime and CodeBuilder 54
1.5.9 Quick Functionality Check 56
1.5.10 Overview of Intel MPSS Tools 58
1.5.11 `miccheck`: Basic Troubleshooting 59
1.5.12 `micctrl`: Coprocessor OS Configuration 61
1.5.13 `micflash`: Coprocessor Firmware Updates 64
1.5.14 `micinfo`: Coprocesssor, Firmware, Driver Info 65
1.5.15 `micrasd`: Reliability Monitor, Error Logging 67
1.5.16 `micsmc`: Real-Time Monitoring Tool 68
1.5.17 User Management on Intel Xeon Phi Coprocessors 71
1.5.18 SSH Client Configuration 76
1.5.19 NFS Mounting a Host Export 77
1.5.20 Sharing a Local Disk with VirtIO Block Device 80
1.5.21 Bridged Networking in Clusters with Coprocessors 82
1.5.22 Peer to Peer Communication between Coprocessors 84
1.5.23 Manual Customization of the coprocessor OS 86

2 Programming Models **87**
2.1 Native Applications and MPI 88
2.1.1 Using Compiler Argument `-mmic` to Compile Native Applications for Intel® Xeon Phi™ Coprocessors 88
2.1.2 Running Native Applications on Using SSH 90
2.1.3 Running Native Applications with `micnativeloadex` 91
2.1.4 Monitoring the Coprocessor Activity with `micsmc` 93
2.1.5 MPI Applications on Intel Xeon Phi Coprocessors 96
2.2 Explicit Offload Model 101
2.2.1 "Hello World" Example in the Explicit Offload Model 101
2.2.2 Offloading Functions 103
2.2.3 Offloading Bitwise-Copyable Data 104
2.2.4 Data and Memory Persistence Between Offloads 106
2.2.5 Asynchronous Offload 108
2.2.6 Target-Specific Code 110
2.2.7 Optional and Conditional Offload, Fall-Back to Host 111
2.2.8 Offload Diagnostics 113
2.2.9 Environment Variables and `MIC_ENV_PREFIX` 114

2.2.10 Proxy Console I/O . . . 116
2.2.11 Review: Explicit Offload Model . . . 117
2.3 Shared Virtual Memory Model . . . 119
2.3.1 Offloading Functions . . . 121
2.3.2 Sharing and Offloading Objects . . . 122
2.3.3 Dynamic Allocation in Shared Virtual Memory . . . 123
2.3.4 Classes in Shared Virtual Memory . . . 125
2.3.5 Placement Operator `new` for Shared Classes . . . 128
2.3.6 Asynchronous Offload . . . 130
2.3.7 Summary for Shared Virtual Memory Model . . . 131
2.4 Using Multiple Coprocessors . . . 132
2.4.1 Multiple Coprocessors with Explicit Offload . . . 133
2.4.2 Multiple Coprocessors in the Shared Virtual Memory Model . . . 138
2.4.3 Multiple Coprocessors with MPI . . . 141
2.5 Offload Programming with OpenMP 4.0 . . . 148
2.5.1 Offload with Pragma Target . . . 149
2.5.2 Data Persistence with Pragma Target Data . . . 150

3 Expressing Parallelism **153**
3.1 Data Parallelism (Vectorization) . . . 154
3.1.1 Vector Instructions: Concept and History . . . 154
3.1.2 Intel Architecture Vector Instruction Sets . . . 155
3.1.3 Is Your Code Using Vectorization? . . . 156
3.1.4 Data Alignment . . . 157
3.1.5 Vector Instructions using Inline Assembly, Compiler Intrinsics and Class Libraries . . . 163
3.1.6 Automatic Vectorization of Loops . . . 166
3.1.7 Extensions for Array Notation in Intel Cilk Plus . . . 171
3.1.8 SIMD-Enabled Functions . . . 173
3.1.9 Assumed Vector Dependence . . . 175
3.1.10 Vectorization Pragmas, Keywords and Compiler Arguments. . . 178
3.1.11 Exclusive Features of the IMCI Instruction Set . . . 181
3.2 Task Parallelism in Shared Memory: OpenMP . . . 186
3.2.1 Multiple Cores and Task Parallelism . . . 186
3.2.2 "Hello World" with OpenMP . . . 188
3.2.3 For-Loops in OpenMP . . . 190
3.2.4 Tasks in OpenMP . . . 194

3.2.5 Shared and Private Variables . . . 198
3.2.6 Synchronization: Avoiding Unpredictable Behavior . . . 202
3.2.7 Reduction: Avoiding Synchronization . . . 209
3.3 Task Parallelism with Intel Cilk Plus . . . 212
3.3.1 "Hello World" in Intel Cilk Plus . . . 213
3.3.2 For-Loops in Intel Cilk Plus . . . 215
3.3.3 Fork-Join Model and Spawning in Intel Cilk Plus . . . 217
3.3.4 Synchronization with Spawned Tasks . . . 219
3.3.5 Reduction: Avoiding Synchronization . . . 221
3.3.6 OpenMP versus Intel Cilk Plus . . . 226
3.3.7 Additional Resources on Shared Memory Parallelism . . . 227
3.4 Process Parallelism in Distributed Memory with MPI . . . 229
3.4.1 Parallel Computing in Clusters with Multi-Core and Many-Core Nodes . . . 229
3.4.2 Program Structure in MPI . . . 235
3.4.3 Point-to-Point Communication . . . 238
3.4.4 MPI Communication Modes . . . 244
3.4.5 Collective Communication and Reduction . . . 253
3.4.6 Further Reading . . . 260

4 Optimizing Parallel Applications 261
4.1 Optimization Roadmap for Intel Xeon Phi Coprocessors . . . 261
4.1.1 Optimization Checklist . . . 261
4.1.2 Expectations . . . 263
4.1.3 Benchmark Methodology . . . 264
4.1.4 Benchmark Computing System . . . 266
4.2 Scalar and General Optimizations . . . 267
4.2.1 Compiler Controls for Optimization . . . 267
4.2.2 Compiler Controls for Precision . . . 269
4.2.3 Optimizing Arithmetic Expressions . . . 275
4.2.4 Programming Practices for High Performance . . . 282
4.2.5 Math Kernel Library for Scalar Arithmetic . . . 287
4.3 Optimizing Vectorization . . . 289
4.3.1 Diagnosing the Utilization of Vector Instructions . . . 289
4.3.2 Unit-Stride Access and Spatial Locality of Reference . . . 290
4.3.3 Regularizing Vectorization Pattern . . . 295
4.3.4 Compiler Hints: Aligned Data Notice . . . 302

4.3.5 Compiler Hints: Pointer Disambiguation 303
4.3.6 Strip-Mining for Vectorization 306
4.3.7 Additional “Tuning Knobs” for Vectorization 310
4.4 Optimization of Multi-Threading 311
4.4.1 Avoiding Synchronization through Parallel Reduction 311
4.4.2 Elimination of False Sharing with Padding 316
4.4.3 Resolving Load Imbalance with Scheduling Control 321
4.4.4 Dealing with Insufficient Parallelism 329
4.4.5 Thread Affinity Optimization 341
4.4.6 Diagnosing Parallel Efficiency, Scalability Tests 354
4.5 Memory Access Optimization 356
4.5.1 General Considerations 356
4.5.2 Loop Tiling 362
4.5.3 Cache-Oblivious Recursive Methods 371
4.5.4 First Touch Allocation and NUMA Policy 376
4.5.5 Cross-Procedural Loop Fusion 380
4.5.6 Advanced Topic: Prefetching 385
4.6 Offload Traffic Control 387
4.6.1 Bandwidth Optimization with Persistent Buffers 387
4.6.2 Masking Offload Latency with Double Buffering 393
4.7 Optimization Strategies for MPI Applications 396
4.7.1 Static Load Balancing 397
4.7.2 Dynamic Work Scheduling 407
4.7.3 Multi-threading within MPI Processes 414
4.7.4 Fabric Control 420

5 Software Development Tools 427
5.1 Intel Math Kernel Library 427
5.1.1 Functions Offered by MKL 428
5.1.2 Linking Applications with MKL. Link Line Advisor 430
5.1.3 MKL on Intel Xeon Phi Coprocessors 432
5.1.4 Automatic offload 433
5.1.5 Compiler-Assisted Offload 439
5.1.6 Native Execution 439
5.1.7 Benchmarks of Select MKL Functions 440
5.2 Intel VTune Amplifier XE 444
5.2.1 System Administration 445

5.2.2 Running VTune 446
5.2.3 Project Management 447
5.2.4 Analysis on the Host CPU 448
5.2.5 Analysis on an Intel Xeon Phi Coprocessor 459

6 Summary and Resources **465**
6.1 Parallel Programming and Intel Xeon Phi Coprocessors 465
6.2 Supplementary Code for Practical Exercises (“Labs”) 467
6.3 Colfax Developer Training 470
6.4 Additional Resources 471

Bibliography **475**

Foreword to the First Edition

We live in exciting times; the amount of computing power available for sciences and engineering is reaching enormous heights through parallel computing. Parallel computing is driving discovery in many endeavors, but remains a relatively new area of computing. As such, software developers are part of an industry that is still growing and evolving as parallel computing becomes more commonplace.

The added challenges involved in parallel programming are being eased by four key trends in the industry: emergence of better tools, wide-spread usage of better programming models, availability of significantly more hardware parallelism, and more teaching material promising to yield better-educated programmers. We have seen recent innovations in tools and programming models including OpenMP and Intel Threading Building Blocks. Now, the Intel® Xeon Phi™ coprocessor certainly provides a huge leap in hardware parallelism with its general purpose hardware thread counts being as high as 244 (up to 61 cores, 4 threads each).

This leaves the challenge of creating better-educated programmers. This handbook from Colfax, with a subtitle of “Handbook on the Development and Optimization of Parallel Applications for Intel Xeon Processors and Intel Xeon Phi Coprocessors” is an example-based course for the optimization of parallel applications for platforms with Intel Xeon processors and Intel Xeon Phi coprocessors.

This handbook serves as practical training covering understandable computing problems for C and C++ programmers. The authors at Colfax have developed sample problems to illustrate key challenges and offer their own guidelines to assist in optimization work. They provide easy to follow instructions that allow the reader to understand solutions to the problems posed as well as inviting the reader to experiment further. Colfax’s examples and guidelines complement those found in our recent book on programming the Intel Xeon Phi Coprocessor by Jim Jeffers and myself by adding another perspective to the teaching materials available from which to learn.

In the quest to learn, it takes multiple teaching methods to reach everyone. I applaud these authors in their efforts to bring forth more examples to enable either self-directed or classroom oriented hands-on learning of the joys of parallel programming.

James R. Reinders
Co-author of “Intel® Xeon Phi™ Coprocessor High Performance Programming"
© 2013, Morgan Kaufmann Publishers
Intel Corporation
March 2013

Preface to the Second Edition

A lot has happened in Intel's "parallel universe" since the publication of the first edition of this book in March 2013. The family of Intel Xeon Phi coprocessors has grown to three series: 3100, 5100 and 7100, offering a range of performance tiers and prices. Active-cooling Intel Xeon Phi coprocessors were introduced, allowing workstation users to take advantage of the Intel Many Integrated Core (MIC) architecture. Plans were released for future Intel MIC architecture products, based on the Knights Landing chip, and capable of acting as a stand-alone CPU. In the CPU domain, Intel Xeon processors based on the Haswell architecture were released, supporting a new instruction set AVX2 and new functionality.

On the software tools side, the Intel Parallel Studio XE 2015 suite was improved to accommodate the new parallel framework standards: OpenMP 4.0 and MPI 3.0. The evolution of Intel VTune Amplifier XE has added many useful functions for automated diagnostics of performance issues. Intel compilers produce more user-friendly optimization reports than before, and have become even smarter about automatic vectorization and other optimizations.

The work in the users' domain did not stand still, either. With a large number of case studies and research articles on applications for the Intel MIC architecture, it is accurate to say that the developer ecosystem has been established. We are proud to say that Colfax has made a considerable contribution to this progress with the first edition of "Parallel Programmin and Optimization with Intel Xeon Phi Coprocessors". In the years 2013 and 2014, over 1000 science and industry experts at tens of locations across North America have been students of the Colfax Developer Training based on this book. Their experience and feedback, along with the innovations in the Intel tools, have built a solid case for the publication of the second edition of "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors".

Among the numerous new features of the second edition, the ones that stand out are:

1. The details unveiled by Intel of the present and future MIC processors, including Knights Landing;

2. Discussion of configuration and system administration of clusters with Intel Xeon Phi coprocessors, including InfiniBand support, bridged network configuration and storage setup;

3. Additional applications based on case studies of our research in 2013–2014 included in the text as references, as well as practical exercises;

4. Console listings, example codes and hyperlinks to online manuals accurate as of Intel Parallel Studio XE 2015, Intel MPSS 3.4.1 and CentOS 7.0 Linux;

5. New programming models made available in OpenMP 4.0;

6. Deeper review of the Intel Math Kernel Library support for the MIC architecture;

7. More convenient page format and font size for on-screen reading, and

8. Numerous updates to the text improving the clarity and depth of the discussion.

We hope that you find this book to be a valuable resource on "all things Xeon Phi", and, as always, we value your feedback. The HPC research department of Colfax International can be reached by email at phi@colfax-intl.com, and the latest updates on our work can be found at research.colfaxinternational.com.

Preface to the First Edition

Welcome to the Colfax Developer Training! You are holding in your hands or browsing on your computer screen a comprehensive set of training materials for this training program. This document will guide you to the mastery of parallel programming with Intel® Xeon® family products: Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors. The curriculum includes a detailed presentation of the programming paradigm for Intel Xeon product family, optimization guidelines, and hands-on exercises on systems equipped with Intel Xeon Phi coprocessors, as well as instructions on using Intel® software development tools and libraries included in Intel® Parallel Studio XE.

These training materials are targeted toward developers familiar with C/C++ programming in Linux. Developers with little parallel programming experience will be able to grasp the core concepts of this subject from the detailed commentary in Chapter 3. For advanced developers familiar with multi-core and/or GPU programming, the training offers materials specific to the Intel compilers and Intel Xeon family products, as well as optimization advice pertinent to the Many Integrated Core (MIC) architecture.

We have written these materials relying on key elements for efficient learning: practice and repetition. As a consequence, the reader will find a large number of code listings in the main section of these materials. In the extended Appendix, we provided numerous hands-on exercises that one can complete either under an instructor's supervision, or autonomously in a self-study training.

This document is different from a typical book on computer science, because we intended it to be used as a lecture plan in an intensive learning course. Speaking in programming terms, a typical book traverses material with a "depth-first algorithm", describing every detail of each method or concept before moving on to the next method. In contrast, this document traverses the scope of material with a "breadth-first" algorithm. First, we give an overview of multiple methods to address a certain issue. In the subsequent chapter, we re-visit these methods, this time in greater detail. We may go into even more depth down the line. In this way, we expect that students will have enough time to absorb and comprehend the variety of programming and optimization methods presented here. The course road map is outlined in the following list.

- Chapter 1 presents the Intel Xeon Phi architecture overview and the environment provided by the MIC Platform Software Stack (MPSS) and Intel Parallel Studio XE on Many Integrated Core architecture (MIC). The purpose of Chapter 1 is to outline what users may expect from Intel Xeon Phi coprocessors (technical specifications, software stack, application domain).

- Chapter 2 allows the reader to experience the simplicity of Intel Xeon Phi usage early on in the program. It describes the operating system running on the coprocessor, with the compilation of native applications, and with the language extensions and CPU-centric codes that utilize Intel Xeon Phi coprocessors: offload and virtual-shared memory programming models. In a nutshell, Chapter 2 demonstrates how to write serial code that executes on Intel Xeon Phi coprocessors.

- Chapter 3 introduces Single Instruction Multiple Data (SIMD) parallelism and automatic vectorization, thread parallelism with OpenMP and Intel Cilk Plus, and distributed-memory parallelization with MPI. In brief, Chapter 3 shows how to write parallel code (vectorization, OpenMP, Intel Cilk Plus, MPI).

- Chapter 4 re-iterates the material of Chapter 3, this time delving deeper into the topics of parallel programming and providing example-based optimization advice, including the usage of the Intel Math Kernel Library. This chapter is the core of the training. The topics discussed in this Chapter 4 include:

  i) scalar optimizations;
  ii) improving data structures for streaming, unit-stride, local memory access;
  iii) guiding automatic vectorization with language constructs and compiler hints;
  iv) reducing synchronization in task-parallel algorithms by the use of reduction;
  v) avoiding false sharing;
  vi) increasing arithmetic intensity and reducing cache misses by loop blocking and recursion;
  vii) exposing the full scope of available parallelism;
  viii) controlling process and thread affinity in OpenMP and MPI;
  ix) reducing communication through data persistence on coprocessor;
  x) scheduling practices for load balancing across cores and MPI processes;
  xi) optimized Intel Math Kernel Library function usage, and other.

  If Chapter 3 demonstrated how to write parallel code for Intel Xeon Phi coprocessors, then Chapter 4 shows how to make this parallel code run fast.

- Chapter 6 summarizes the course and provides pointers to additional resources.

Throughout the training, we emphasize the concept of portable parallel code. Portable parallelism can be achieved by designing codes in a way that exposes the data and task parallelism of the underlying algorithm, and by using language extensions such as OpenMP pragmas and Intel Cilk Plus. The resulting code can be run on processors as well as on coprocessors, and can be ported with only recompilation to future generations of multi- and many-core processors with SIMD capabilities. Even though the Colfax Developer Training program touches on low-level programming using intrinsic functions, it focuses on achieving high performance by writing highly parallel code and utilizing the Intel compiler's automatic vectorization functionality and parallel frameworks.

The handbook of the Colfax Developer Training is an essential component of a comprehensive, hands-on course. While the handbook has value outside a training environment as a reference guide, the full utility of the training is greatly enhanced by students' access to individual computing systems equipped with Intel Xeon processors, Intel Xeon Phi coprocessors and Intel software development tools. Please check the Web page of the Colfax Developer training for additional information: http://www.colfax-intl.com/xeonphi/

Welcome to the exciting world of parallel programming!

List of Abbreviations

ALU	Arithmetic Logic Unit
AO	Automatic Offload
AVX	Advanced Vector Extensions (SIMD standard)
BLAS	Basic Linear Algebra Subprograms
CAO	Compiler Assisted Offload
CCL	Coprocessor Communication Link
CFD	Computational Fluid Dynamics
CLI	Command Line Interface
CPI	cycles per instruction
CPU	Central Processing Unit, used interchangeably with the terms "processor" and "host" to indicate the Intel Xeon processor, as opposed to the Intel Xeon Phi coprocessor
CRI	Core Ring Interconnect
DAPL	Direct Access Programming Library
DFFT	Discrete Fast Fourier Transform
DGEMM	Double-precision General Matrix-Matrix Multiply
DMA	Direct Memory Access
DSS	Direct Sparse Solver
DTD	Distributed Tag Directory
ECC	Error Correction Code
FFT	Fast Fourier Transform

FMA	Fused Multiply-Add
FP	Floating-point
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GDDR	Graphics Double Data Rate memory
GFLOP	Gigaflop, 10^9 floating point operations.
GFLOP/s	Performance metric. Unless stated otherwise, refers to theoretical peak performance of the multiply and add operation(s), or to the performance of the HPC Linpack Benchmark
GPGPU	General-Purpose Graphics Processing Unit
GUI	Graphical User Interface
HPC	High Performance Computing
I/O	Input/Output
IMCI	Initial Many-Core Instructions
IP	Internet Protocol
ISA	Instruction Set Architecture
ITAC	Intel Trace Analyzer and Collector
KNC	Knights Corner
KNL	Knights Landing
LAPACK	Linear Algebra Package
LRU	Least Recently Used, a cache replacement policy
MESI	Modified/Exclusive/Shared/Invalid, a cache coherency protocol
MKL	Math Kernel Library

MMB	Maximum Memory Bandwidth
MMIO	memory-mapped I/O
MMX	Multimedia Extensions (SIMD standard)
MPI	Message Passing Interface
MPSS	Manycore Platform Software Stack
NFS	Network File Sharing Protocol
NUMA	Non-Uniform Memory Access
OEM	Original Equipment Manufacturer
OFED	OpenFabrics Enterprise Distribution
OpenCL	Open Computing Language
OS	operating system
PARDISO	Parallel Direct Sparse Solver
PCIe	Peripheral Component Interconnect Express
PFLOP	Petaflop, 10^{15} floating point operations. See also GFLOP
PMU	Performance Monitoring Unit
PSM	Performance Scaled Messaging
QPI	Quick Path Interconnect
RAM	Random Access Memory
RCI ISS	Iterative Sparse Solvers based on Reverse Communication Interface
RCP	Recommended Customer Price
RDMA	Remote Direct Memory Access
RNG	Random Number Generator

ScaLAPACK	Scalable Linear Algebra Package
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessor
SSE	Streaming SIMD Extensions (SIMD standard)
SSH	Secure Shell protocol
SVML	Short Vector Math Library
TD	Tag Directory
TDP	thermal design power
TFLOP	Teraflop, 10^{12} floating point operations. See also GFLOP
TLB	Translation Lookaside Buffer
TMI	Tag Matching Interface
TPP	Theoretical Peak Performance
TSX	Transactional Synchronization Extensions
VML	Vector Mathematical Library
VSL	Vector Statistical Library

CHAPTER 1
Introduction

This chapter introduces the Intel manycore architecture and positions Intel Xeon Phi coprocessors in the context of parallel programming.

Even though the focus of this book is on Intel Xeon Phi coprocessors, we will also briefly discuss the Intel Xeon family CPUs. This is necessary to put the performance characteristics of Intel Xeon Phi coprocessors in proper perspective.

Our approach to comparing CPUs and the manycore architecture builds upon the first question that the designer of a computing system may ask: does it make more sense spend the budget for setup costs and operational expenses on all-CPU nodes, or purchase fewer nodes, but enhance them with coprocessors? Naturally, technical specifications alone cannot be used to answer this question. This question can be answered only by benchmarks of specific applications in combination with power measurements, total cost analysis, and additional factors such as development effort, available rack space, administrative burden, etc.

This chapter will help to set expectations for the potential of the Intel manycore architecture for the reader's outstanding computing challenges.

1.1. Intel Xeon Phi Coprocessors

1.1.1. Technology Overview

Intel Xeon Phi coprocessors have been designed by Intel Corporation as a supplement to the Intel Xeon processor family. The coprocessors feature the Intel manycore architecture, which enables fast and energy-efficient execution of some High Performance Computing (HPC) applications.

In most Intel communications, the term “manycore”, refers to the architecture of the Intel Xeon Phi product family, while “*multi-core*” architecture referes to the Intel Xeon family processors.

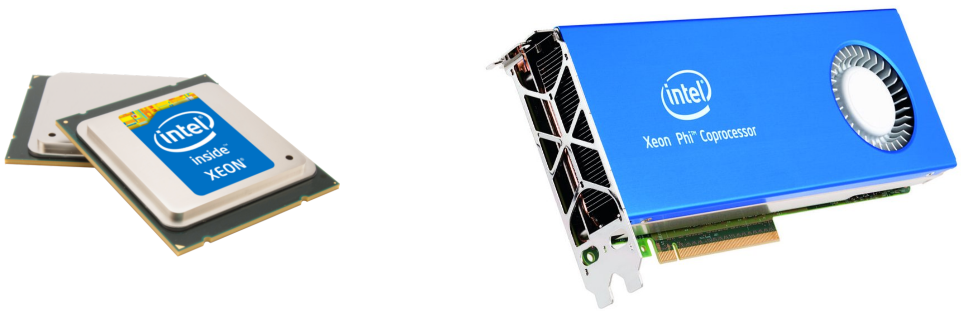


Figure 1.1: *Left*: multi-core Intel Xeon processors (CPUs), *Right*: manycore Intel Xeon Phi coprocessor. Relative sizes are not to scale.

The manycore architecture may yield more performance per watt of power and per dollar of setup costs than traditional multi-core CPUs. However, not every application can be accelerated by manycore coprocessors. Intel Xeon Phi coprocessors derive their high performance from multiple cores, dedicated vector arithmetic units with wide vector registers, and cached onboard GDDR5. High energy efficiency is achieved through the use of low clock speed x86 cores with lightweight design suitable for parallel HPC applications. Therefore, only highly parallel applications supporting vectorized arithmetic with well-behaved (or negligible) memory traffic will thrive on the manycore architecture.

Figure 1.2: Examples of computing system solutions featuring the Intel Xeon Phi coprocessors. *Left*: A Colfax Workstation CXP7450 with two Intel Xeon Phi coprocessors. *Right*: A Colfax Server CXP9000 with eight Intel Xeon Phi coprocessors. Relative sizes not to scale.

First generation Intel Xeon Phi coprocessors based on the Knights Corner (KNC) chip are end-point Peripheral Component Interconnect Express (PCIe) devices. They can be installed on the PCIe bus and operated in coprocessor-ready computing systems, including workstations (e.g., Figure 1.2, left) and servers (e.g., Figure 1.2, right).

An Intel Xeon Phi coprocessor cannot operate without a CPU-based host system, which is the reason for terming these products *coprocessors*. Because they reside on the PCIe bus and have their own on-board RAM, coprocessors do not share memory address space with the CPU. Consequently, the mere presence of a coprocessor in a system does not automatically improve the performance of applications running on the CPU. To utilize the MIC architecture, the application or the cluster execution manager must be aware of the presence of a coprocessor.

The usage model of the second generation Intel MIC based on the Knights Landing (KNL) chip will be different. The second generation chip will be available as a standalone processor, as well as a PCIe-endpoint device. For the standalone processor version, applications need not be coprocessor-aware in order to be accelerated. However, a prerequisite for accelerated performance is optimization of the application code for multi-core and manycore architectures. See Section 1.4 for more information.

1.1.2. Conventional Programming, Portable Code

This section describes the value proposition of the Intel MIC architecture.

Established Programming Models

Because of the similarity of the manycore and multi-core architectures, an Intel Xeon Phi coprocessor can execute applications compiled from the same C/C++ or Fortran code as an Intel Xeon processor. Furthermore, Intel Xeon processors and Intel Xeon Phi coprocessors support the same parallel frameworks and require similar code optimization methods. This is a significant advantage of the Intel manycore architecture over computing accelerator technologies (GPGPUs and FPGAs).

The process of application porting to GPGPUs typically involves discarding and re-writing from scratch the compute-intensive pieces of code. This process is time consuming and prone to the introduction of new bugs, because the application cannot be tested until porting is complete.

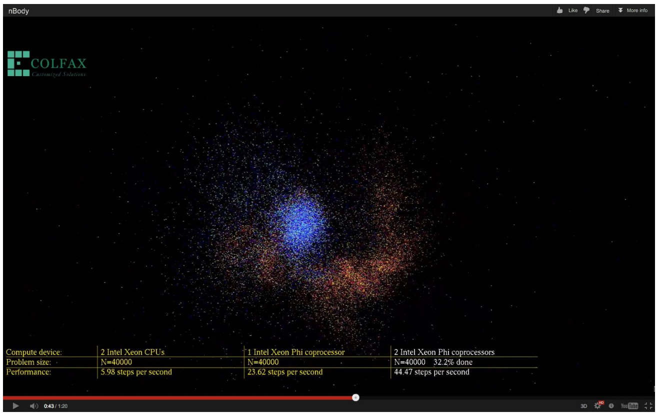
In contrast, it is usually possible to port a code designed for many-core systems to the MIC architecture. After that, the programmer can incrementally adapt (optimize) the application to the coprocessor platform. Such easy porting is very important for projects that require modernization of millions lines of legacy scientific and industrial applications.

It is fair to say that Intel Xeon Phi coprocessors are *easy to program* because they use the same languages, frameworks and principles as general-purpose Intel architecture CPUs, which are familiar to the overwhelming majority of developers. At the same time, Intel Xeon Phi coprocessors are only useful in the context of *parallel programming*, which is not the comfort zone for the majority of CPU application developers. This book aims to assist the developers in understanding the programming methods required to leverage parallelism in both Intel Xeon processors and Intel Xeon Phi coprocessors.

Common Optimization Requirements

It is incorrect to think that the ability to run legacy code “out of the box” on Intel Xeon Phi coprocessors means immediate acceleration. On the contrary, in many cases, the performance of applications just ported to the MIC architecture is disappointing, and code optimization is required. Optimization is often a significantly greater effort than initial porting.

At the same time, the optimization methods used in applications for Intel Xeon Phi are the same methods that are used in applications for general-purpose Intel architecture CPUs. Indeed, case studies show that a code optimized for the MIC platform also runs significantly faster on a CPU (for a synthetic example, see paper [1] illustrated in Figure 1.3; code for a similar application is available among the Supplementary Code for Practical Exercises as Lab 4.01 – see Section 6.2; for realistic examples, refer to [2]).



Compute device:	2 Intel Xeon CPUs	1 Intel Xeon Phi coprocessor	2 Intel Xeon Phi coprocessors
Problem size:	N=40000	N=40000	N=40000 32.2% done
Performance:	5.98 steps per second	23.62 steps per second	44.47 steps per second

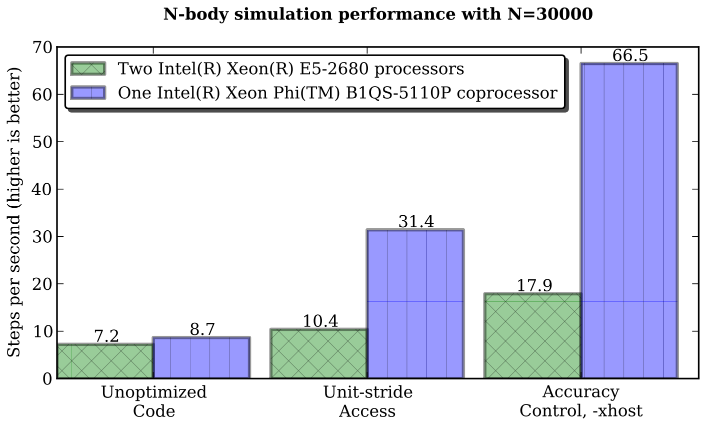


Figure 1.3: The same C language code used for a simple N-body simulation on the CPU and on a coprocessor. See white paper [1] for more information.

Heterogeneous and Accelerated Computnig

From the development maintenance point of view, having a single code for the main processor and for the coprocessor opens doors to heterogeneous computing and public code distribution. A heterogeneous application may utilize the CPU together with the MIC coprocessor, wasting no resources. Public code with support for Intel Xeon Phi coprocessors has the advantage that for users who do not own a coprocessor, the execution can seamlessly fall back to the CPU.

If an application is developed from scratch, rather than ported from a legacy C, C++ or Fortran code, then developers have additional options for ensuring code portability. For example, the OpenCL parallel framework can be used to design a single code for multiple platforms, including the Intel MIC architecture. In practice, however, even though an OpenCL application can run on a CPU as well as on a GPGPU or a MIC coprocessor, it has to be tuned for each platform. At the same time, the similarity of the multi-core CPU and the MIC architectures ensures that a high-level language code optimized for the MIC architecture is also optimal for the CPU.

Portability and Future-Proofing

Portability is an important consideration for many developers. Ideally, a high-level language code developed once should run, with minimal modifications, on other manufacturers' processor architectures, as well as on older and future computing platforms.

Intel Xeon Phi coprocessors are based on the basic architectural elements common in Intel 64 and Itanium architectures, AMD x86 processors, Sun SPARC, IBM Blue Gene, Power architecture, and other general purpose processors: cores, threads, cached memory, vectors. Even though instruction sets and quantitative aspects of technical specifications are not compatible across these architectures, the *approach* to programming computers based on these architectural elements is common.

Future Intel parallel architectures will evolve using the same architectural elements (see Section 1.4). This ensures longevity of high-level language codes for developed today's Intel Xeon Phi coprocessors. See Section 1.4.3 an extended discussion of this topic.

1.1.3. Heterogeneous Computing and Clustering

Programming models for Intel Xeon Phi coprocessors include native execution and offload-based approaches. These approaches enable developers to design a spectrum of hybrid computing models, ranging from multi-core-hosted (i.e., only employing the CPU) to multi-core-centric (i.e., executing on the host system with some operations performed on the coprocessor) to symmetric (i.e., employing the host and the coprocessor on an equal basis) and manycore-hosted (i.e., executing exclusively on a set of coprocessors).

The choice of work division between the host and the coprocessor is dictated by the nature of the application. Highly parallel, vectorized workloads (e.g., linear algebraic calculations) can be executed on the coprocessor as well as on the host. However, serial segments of an application perform significantly better on Intel Xeon processors, and so do applications with stochastic memory access patterns. The overhead of data transport over the PCIe bus should also be taken into consideration.

Figure 1.4 summarizes the development options for systems enabled with Intel Xeon Phi coprocessors.

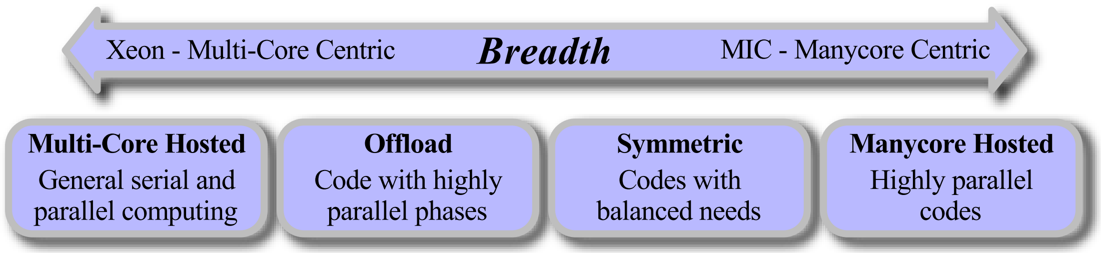

Xeon - Multi-Core Centric
Breadth
MIC - Manycore Centric
Multi-Core Hosted
General serial and parallel computing
Offload
Code with highly parallel phases
Symmetric
Codes with balanced needs
Manycore Hosted
Highly parallel codes

Figure 1.4: Intel architecture benefit: wide range of development options. Breadth, depth, familiar models meet varied application needs. *Diagram based on Intel materials.*

Intel Xeon Phi coprocessors are Internet Protocol (IP)-addressable devices running a Linux operating system (OS). This property enables straightforward porting of code written for the Intel Xeon architecture to the MIC architecture. This, combined with code portability, makes Intel Xeon Phi coprocessors a compelling platform for heterogeneous clustering. In heterogeneous cluster applications, host processors and MIC coprocessors can be used on an equal basis as individual compute nodes.

1.1.4. Intel Xeon Phi Product Family

Intel Xeon Phi coprocessors come in a range of models featuring different thermal design power (TDP), different theoretical peak performance and different memory capacities. Each model is identified by a 5-character code as shown in Figure 1.5.

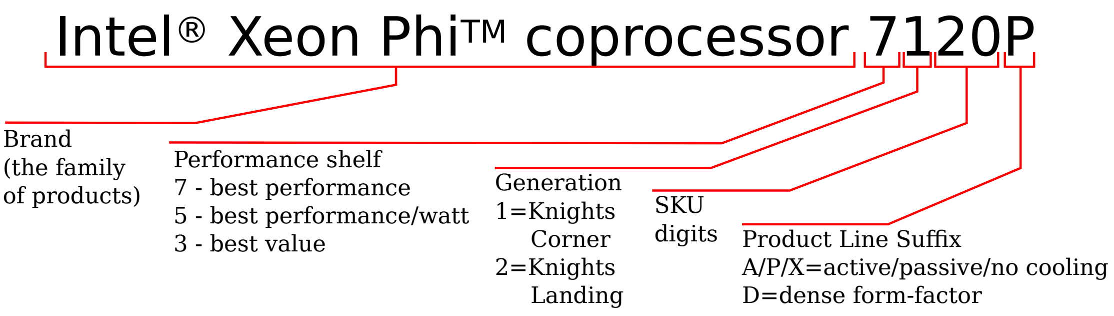


Figure 1.5: Five-character code identifying the model of an Intel Xeon Phi coprocessor.

The first character in the code stands for the performance shelf: 3, 5 or 7. The second character is the product generation. As of the writing of this book (Feb 2015), only generation 1 (KNC) is available. Therefore, available models can be organized into 3 groups: 3100, 5100 and 7100 series.

3100 Series is designed as the price-optimal group. Models in this series contain fewer active cores, less onboard memory, and feature a lower memory bandwidth than in other series. This series is a good choice for compute-bound workloads.

5100 Series is optimized for performance per watt. 5100 Series coprocessors feature lower TDP, contain more memory and cores than the 3100 series, and perform better in memory bandwidth-bound and memory capacity-bound workloads.

7100 Series is the top performing group. It has the greatest core count, memory size and bandwidth of all series. It also comes at a higher price than other series, and greater TDP than the 5100 series.

The third and fourth characters in the code are the SKU digits. These generally indicate the product stepping, and they increase as minor silicone-level improvements are made.

Finally, the fifth character is a letter, which indicates the cooling solution or special usage case of the model.

A stands for active cooling. These coprocessors come inside a heat sink with a built-in and fan, and are suitable for usage in desktop workstations (Figure 1.6, left). This cooling solution is not reliant on system fans, and the built-in fan speed is controlled by an onboard sensor, which allows these coprocessors cards to be quiet in the idle state.

P stands for passive cooling. These coprocessors have come in a heat sink, but have no fan (Figure 1.6, right). They cannot be used in workstations because of imminent overheating, and are designed for servers.

X indicates that no cooling solution is provided, i.e., there is no heat sink on the card. These coprocessors can be used only with custom cooling solutions such as liquid cooling, because normal airflow from common system fans is not sufficient for heat removal.

D is the dense form factor model. It does not have a heat sink, and is smaller in size than the X option. These models are designed for specialized solutions capable of supporting a large density of thermal dissipation.

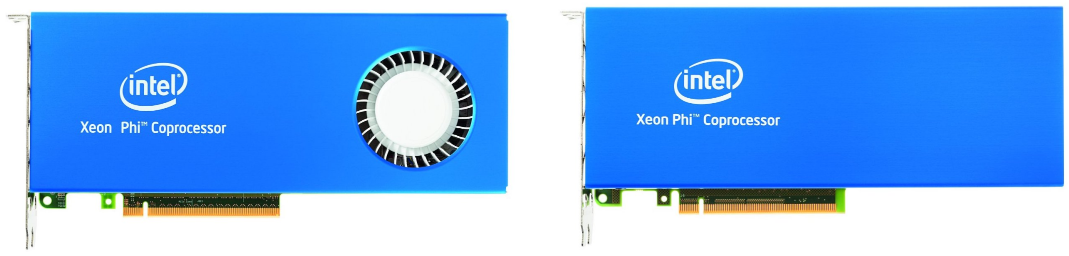


Figure 1.6: Active and passive cooling solutions of Intel Xeon Phi coprocessors.

Model	**TDP (W)**	**Cores**	**Clock (GHz)**	**Turbo Boost**	**RAM (GiB)**	**MMB (GB/s)**	**DP TPP (GFLOP/s)**	**RCP**
3120P	300	57	1.100	no	6	240	1003.2	$1695
3120A	300	57	1.100	no	6	240	1003.2	$1695–1960
5120D	245	60	1.053	no	8	352	1010.9	$2759
5110P	225	60	1.053	no	8	320	1010.9	$2437–2649
7120X	300	61	1.238–1.333	1.0	16	352	1208.3	$4129
7120P	300	61	1.238–1.333	1.0	16	352	1208.3	$4129
7120D	270	61	1.238–1.333	1.0	16	352	1208.3	$4235
7120A	300	61	1.238–1.333	1.0	16	352	1208.3	$4129

Table 1.1: Models of Intel Xeon Phi coprocessors available as of May 2014. Columns contain: model name, thermal design power (TDP) in Watts, number of physical cores, their clock speed, Intel Turbo Boost technology support, onboard memory size in GiB, maximum memory bandwidth (MMB) in GB/s, double precision (DP) theoretical peak performance (TPP) in GFLOP/s, and RCP. RCP is price guidance for bulk purchases by direct Intel customers, subject to change without notice, not a formal pricing offer from Intel or Colfax International.

Table 1.1 summarizes the currently available models of Intel Xeon Phi coprocessors and their specifications. In this table, all quantities are obtained from the Intel Xeon Phi Product Family page, except for the Theoretical Peak Performance (TPP), which is estimated according to Equation (1.1):

$$\frac{\text{TPP}}{\text{GFLOP/s}} = \frac{\text{Clock Speed}}{\text{GHz}} \times \text{FMA} \times \frac{\text{SIMD Register Size}}{\text{sizeof(TYPE)}} \times \text{Cores}. \quad (1.1)$$

For Intel Xeon Phi coprocessors, FMA $= 2$ (the fused multiply-add operation is performed in one cycle), SIMD register size is 512 bits (64 bytes), and the size of double precision numbers is 64 bits (8 bytes). See Section 1.3 and 4.5 for additional discussion.

1.1.5. Intel Xeon Processor E3, E5 and E7 Family

Only the Intel Xeon family server processors are considered in this book in conjunction with Intel Xeon Phi coprocessors. Desktop and mobile device product lines (Intel® Core™, Intel® Atom™, Intel® Pentium®and Intel® Celeron®) are not discussed, because

a) generally, there is no support for Intel Xeon Phi coprocessors in boards, chipsets and BIOS software compatible with consumer CPUs,

b) the set of features, TDP, and cost of consumer processors are very different from server CPUs, which is not suitable for a meaningful comparison.

Intel Xeon family adheres to the numbering scheme shown in Figure 1.7.

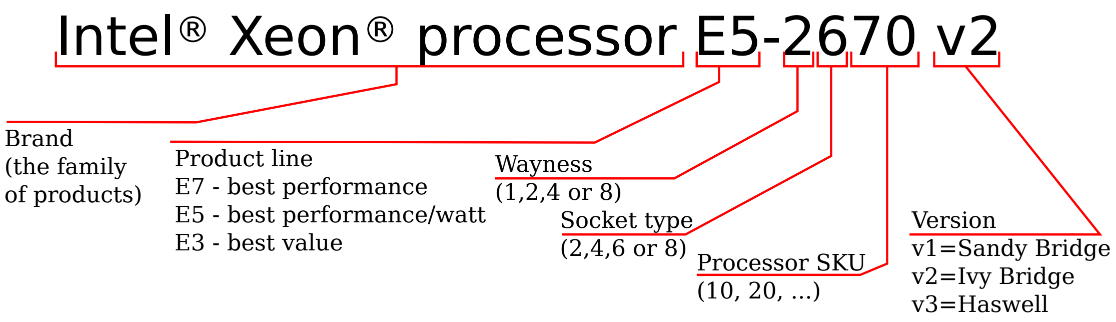


Figure 1.7: Codes identifying the model of an Intel Xeon CPU.

The product line (E3, E5 and E7) for Intel Xeon CPUs is similar to the performance shelf for Intel Xeon Phi coprocessors: E3 is the lowest-cost option, E5 is optimized for best power consumption, and E7 is the top performing line.

Wayness is the maximum number of CPU sockets per node. Two digits of the processor SKU places the CPU within its family. There differences between different SKUs are mostly quantitative. The SKU determines the number of cores, clock speed, maximum memory bandwidth, and cache size.

After the SKU, in some CPU models, an additional suffix “L” is present, indicating a low power consumption model.

Finally, the version of the CPU (v1, v2 or v3) determines the type of processor microarchitecture used in the chip: Sandy Bridge (v1), Ivy Bridge (v2) or Haswell (v3). The difference between versions depends on whether the version update was a "tick" or a "tock". For instance, Sandy Bridge to Ivy Bridge development was a "tick", i.e., a newer, smaller transistor technology was used in v2. As a result, v2 CPUs may have more cores, greater performance and lower power consumption than v1, however, the instruction set is unchanged. In contrast, Ivy Bridge to Haswell update was a "tock", i.e., the same transistor technology as in Ivy Bridge was used to produce an architecturally improved chip. As a result, v3 CPUs support additional instruction sets (in this case, AVX2) and features (e.g., TSX), and operate with a different chipset.

Model	**TDP (W)**	**Cores**	**Clock (GHz)**	**Cache (MiB)**	**MMB (GB/s)**	**DP TPP (GFLOP/s)**	**RCP**
E5-2603	80	4	1.8	10	34.1	57.6	$198
E5-2690	135	8	2.9	20	51.2	185.6	$2057
E5-2603 v2	80	4	1.8	10	42.6	57.6	$202
E5-2697 v2	130	12	2.7	30	59.7	259.2	$2614
E5-2603 v3	85	6	1.6	15	68.0	76.8	$217
E5-2697 v3	145	14	2.6	35	51.0	291.2	$2706

Table 1.2: Some of the models of Intel Xeon processors available as of April 2015. Columns as in Table 1.1. RCP is price guidance for bulk purchases by direct Intel customers, subject to change without notice, not a formal pricing offer from Intel or Colfax International. Values are per socket; double all values for a dual-socket CPU.

Of the multitude of Intel Xeon SKUs, the most important for the discussion in this book are two-way multi-core CPUs. This is because their TDP and cost are comparable to those of a single Intel Xeon Phi coprocessor (see also Section 4.1.2).

Table 1.2 lists key technical specifications of a few selected two-way models of Intel Xeon processors. Note that the quantities in Table 1.2 are reported per socket, so for a two-way machine, they must be multiplied by 2. DP TPP is estimated similarly to Equation (1.1), with SIMD Register Size=256 bits, and an additional factor of $\times 2$ to account for two ALUs in Sandy Bridge and Ivy Bridge architectures, or for FMA in the Haswell architecture (see Section 4.5).

For complete information on the technical specifications of other Intel processors, refer to http://ark.intel.com/.

1.2. MIC Architecture: Developer’s Perspective

Programming applications for Intel Xeon Phi coprocessors is not significantly different from programming for Intel Xeon processors. Indeed, both devices feature the x86 architecture, support for C, C++ and Fortran, and common parallelization libraries. Therefore, only familiarity with multi-core processor programming is required. However, in order to optimize applications, it is helpful to know some of the architectural properties of the coprocessor. Relevant properties are described in this section.

1.2.1. Knights Corner Die Organization

The KNC die is manufactured using the 22 nm process technology with 3-D Trigate transistors. This technology allows to fit 62 cores and up to 16 GiB of cached GDDR5 memory on a single die. In most production coprocessor models, from 57 to 61 cores are active, and from 6 to 16 GiB of RAM is available. The cores and GDDR5 memory controllers are connected via a bi-directional Core Ring Interconnect (CRI) (see Figure 1.8).

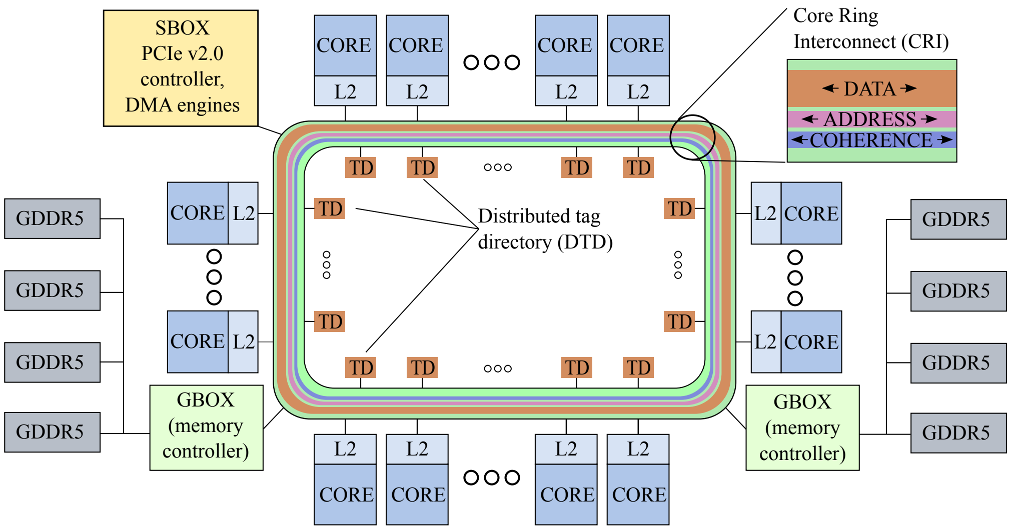


Figure 1.8: Knights Corner die organization. A bi-directional ring interconnects cores, tag directories, onboard memory controllers and PCIe/DMA engines.

The CRI consists of three bi-directional rings:

1. the data ring, as the name suggests, carries application data between cores and memory controllers;
2. the address ring carries commands from cores to other devices for memory fetches, and
3. the acknowledgement ring is used for cache coherency traffic.

In addition to cores, the CRI contains devices that allow the chip to operate as a symmetric multiprocessor:

i) A distributed Tag Directory (TD): multiple TD devices maintain information about cache lines in the L2 caches, and of their states. Together, all TDs form a Distributed Tag Directory (DTD), responsible for maintaining a global cache coherency.
ii) 6 to 8 GBOX units, which are memory controllers for onboard GDDR5 RAM. Each controller has two 32-bit channels delivering up to 5.5 GT/s. The RAM has the Error Correction Code (ECC) capability.
iii) An SBOX (system box) unit, supporting a PCI Express v2.0 logic with eight Direct Memory Access (DMA) channels for data transfer from system to GDDR5 memory.

From the programmer's perspective, this architecture is more non-uniform than the symmetric architecture of an Intel Xeon CPU. Indeed, the latency and bandwidth of communication between two cores depends on the distance between them on the CRI. Therefore, applications for the MIC architecture must, whenever possible, maintain good data locality and avoid synchronization. This will be discussed in greater detail in Chapter 4.

1.2.2. Core Specifications

Each of the 57 to 61 cores of a KNC chip has structure schematically depicted in Figure 1.9.

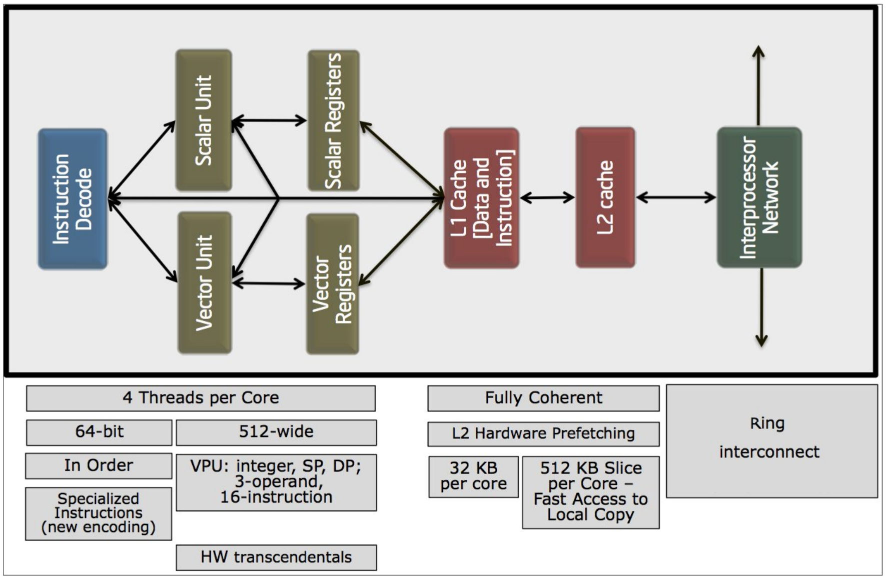


Figure 1.9: The topology of a single Knights Corner core. *Image credit: Intel Corporation.*

Instructions in the executable code are processed on the core either on the scalar, or on the vector arithmetic unit.

The purpose of the scalar unit is to maintain compatibility with applications for the x86 architecture. It allows, for instance, to run a Linux operating system on the coprocessor. The scalar unit may also be used for compute-intensive calculations, however, it is a severely sub-optimal way to use the coprocessor, as the bulk of the compute power is in the vector unit. To put a number on it, it is quoted (e.g., [3]) that the fraction of the chip area used legacy x86 support is only 2%.

The vector unit, present in each core, has 512-bit wide registers and supports the Initial Many-Core Instructions (IMCI) instruction set. This functionality allows SIMD operations on up to 16 single precision floating-point numbers, or up to 8 double precision numbers. The IMCI instructions include floating-point addition, multiplication and Fused Multiply-Add (FMA), division, reciprocal, trigonometric, exponential and logarithmic functions, type conversion, bitwise operations, and other instructions; operations on 32- and 64-bit integers are also supported. See Section 3.1.11 for more detail.

The cores of KNC are successors of the Intel Pentium processor cores. Numerous improved features, however, differentiate KNC from Pentium:

i) Each x86 core on the Knights Corner chip has its own Performance Monitoring Unit (PMU) with advanced features. It is able to count metrics including the number of retired instructions, elapsed cycles, memory controller events, vector unit utilization and statistics, remote cache access statistics, and other. Performance is monitored at the individual thread level.

ii) Advanced power management features in KNC include C3 and C6 states (power conservation in shallow and deep idle cores), PC3 and PC6 states (power conservation on the entire chip in shallow and deep idle states) and Turbo mode in some models.

iii) Streaming store is a bandwidth-boosting feature in KNC for certain memory-intensive workloads. It allows a core to write a cache line to memory without reading it first.

iv) Each KNC core is running 4 hardware threads in a round-robin order. Every hardware thread issues instructions every other cycle, and therefore, two hardware threads per core are necessary to utilize all available cycles. Additional two hardware threads may improve performance in the same situations where hyper-threading improves performance in Intel Xeon processors.

v) Even though KNC cores are in-order, they do not stall upon all cache misses. For read misses, the hardware thread triggering the miss is stalled, while other threads can continue processing. Each KNC core can handle up to 38 asynchronous prefetch requests for both read and write instructions.

vi) KNC cores contain hardware prefetchers for moving data from the main memory to the L2 cache, but no prefetchers for L2 to L1 traffic.

The Level-1 (L1) and Level-2 (L2) caches are embedded in the core. The hierarchical cache structure is a significant component in the KNC productivity. The details of cache organization and properties are discussed in Section 1.2.3

1.2.3. Memory Hierarchy and Cache Properties

Memory hierarchy in KNC features two levels of caches: the L1 cache nearest to the processor (32 KiB per core) and the L2 cache (512 KiB) per core. The caches are 8-way associative, fully coherent using the MESI protocol, with a pseudo-LRU (Least Recently Used) replacement policy. The cache properties of KNC are summarized in Table 1.3.

Parameter	L1 (per core)	L2 (per core)
Size	32 KiB data + 32 KiB instruction	512 KiB
Line size	64 B	64 B
Access time	1 cycle	11 cycles
Associativity	8-way	8-way
Set conflict	4 KiB	64 KiB
TLB Coverage	64×4KiB pages or 8×2MiB pages	64×4KiB pages or 64×2MiB pages
Prefetching	Software	Hardware and software

Table 1.3: Cache properties of the Knights Corner architecture.

Associativity

Eight-way associativity strikes a balance between the low overhead of direct-mapped caches and the versatility of fully-associative caches. An 8-way set associative cache chooses, for each memory address, one of 8 ways of cache (i.e., cache segments) into which the data at that memory address be placed. Within the way, the data can be placed anywhere.

Replacement Policy

The Least Recently Used policy is such behavior of a cache that when a cache line has to be evicted from cache in order to load new data, a line is evicted from least recently used set. LRU is implemented by dedicated hardware units in the cache.

Set Conflicts

To the developer, an important property of multi-way associative caches with LRU is the possibility of *set conflict*. A set conflict may occur when the code processes data with a certain stride in virtual memory. For KNC, the stride is 4 KiB in the L1 cache and 64 KiB in L2 cache. With this stride, data from memory must be mapped into same set, and, if LRU is not functioning properly, some data may be evicted prematurely, causing performance loss.

Coherency and Tag Directory

A coherent cache guarantees that when data is modified in one cache, copies of this data in all other caches will be correspondingly updated before they are made available to the cores accessing these other caches. In KNC, L2 caches are not truly shared between all cores; each core has its private slice of the aggregate cache (see Figure 1.8). Therefore, the coherency of the L2 cache comes at the cost of potential performance loss when data is transferred across the ring interconnect.

To accelerate the coherency protocol, the information about the states of cached lines are distributed across 64 isolated tag directories (TDs) operating in parallel. There is a system-wide mapping that assigns any physical memory address to a specific TD.

When a core requests a cache line, the local TD is checked first. If the line address maps to the local TD, the chip receives the address, state and owner (one of the L2 caches) of the line, and the coherency protocol is used to continue work with this cache line. In this way, coherency checks are parallelized on the hardware level.

The distributed L2 cache and distributed tag directory lead to another consequence. In workloads with access to shared data, some cache lines may be stored in L2 caches of multiple cores. On the one hand, this improves the access time to these cache lines. On the other hand, it reduces the effective usable size of the L2 cache.

Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer, or TLB, is a cache residing on each core, that speeds up the lookup of the physical memory address corresponding to a virtual memory address. Entries, or *pages* in TLB can vary in the amount of memory that they map. The physical size of the TLB places restrictions on the number of pages on the total address range stored in TLB. When memory address accessed by the code is not found in TLB, the TLB entries must be re-built in order to look up that address. This causes a *data page walk* operation, which is fairly expensive compared to the misses in L1 and L2 caches. Optimal TLB page properties depend on the memory access pattern of the application. As with other cache function, TLB performance can generally be improved by increasing the locality of data access in time and space.

Prefetching

Another important property of caches is *prefetching*. During the program execution, it is possible to request that data must be fetched into cache *before* the core uses this data. This diminishes the impact of memory latency on performance. Two types of prefetching are available in the KNC architecture: *software prefetching*, when the prefetch instruction is issued by the code in advance of the data usage, and *hardware prefetching*, when a dedicated hardware unit in the cache learns the data access pattern and issues prefetch instructions automatically. The L2 cache in KNC has a hardware prefetcher, while the L1 cache does not. Normally, Intel compilers automatically introduce L1 prefetch instructions into the compiled code. However, in some cases it may be desirable to manually tune the prefetch distances or to disable software prefetching when it introduces undesirable TLB misses. See Section 4.5.6 for more information on this topic.

Additional Reading

A comprehensive source on microprocessor caches is the book "Computer Architecture: a Quantitative Approach" by Hennessy and Peterson [4].

1.2.4. Integration into the Host System through MPSS

From a developer's perspective, an Intel Xeon Phi coprocessor is a compute node with an IP address and a Linux operating system running on it. That said, the coprocessor:
- responds to `ping`;
- runs an SSH (Secure Shell Protocol) server, which allows users to log into the coprocessor and obtain a shell;
- hosts virtual or NFS filesystems with standard Linux ownership and permissions,
- and is capable of running other services such as Network File Sharing Protocol (NFS) and Message Passing Interface (MPI).

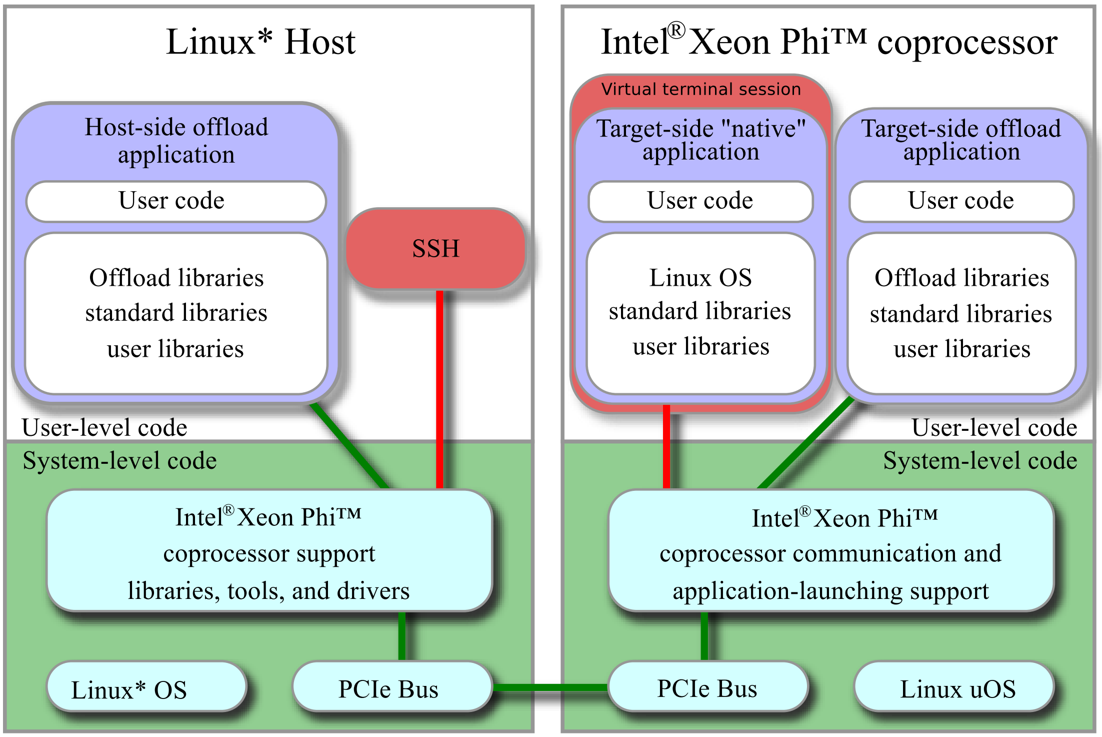

Linux* Host
Intel® Xeon Phi™ coprocessor
Host-side offload application
User code
Offload libraries standard libraries user libraries
SSH
Virtual terminal session
Target-side "native" application
User code
Linux OS standard libraries user libraries
Target-side offload application
User code
Offload libraries standard libraries user libraries
User-level code
System-level code
User-level code
System-level code
Intel® Xeon Phi™ coprocessor support libraries, tools, and drivers
Intel® Xeon Phi™ coprocessor communication and application-launching support
Linux* OS
PCIe Bus
PCIe Bus
Linux uOS

Figure 1.10: MPSS, Manycore Platform Software Stack, contains a driver for the Intel Xeon Phi coprocessor, a runtime environment for offload applications, and a Linux OS for the coprocessor.

On the operating system level, the above mentioned functionality is provided by the Manycore Platform Software Stack (MPSS), a suite of tools including drivers, daemons, command-line and graphical tools. The role of MPSS is to boot the coprocessor, load the Linux operating system, populate the virtual file system, and to enable the host system user to interact with Intel Xeon Phi coprocessor in the same way as the user would interact with an independent compute node on the network.

Figure 1.10 illustrates the role of MPSS in the operation of an Intel Xeon Phi coprocessor. User-level code for the coprocessor runs in an environment that resembles a compute node. The network traffic is carried over the PCIe bus instead of network interconnects.

User applications can be built in two ways:

1. For high performance workloads, Intel compilers can be used to compile C, C++ and Fortran code for the MIC architecture. Intel compilers are not a part of the MPSS; they are distributed in additional software suite Intel Parallel Studio XE (see Section 1.1.2 and Section 1.2.7).
2. For the Linux operating system running on Intel Xeon Phi coprocessors, a specialized version of GNU Compiler Collection (GCC) is available. This specialized GCC is used to compile the Linux distribution for the coprocessor. However it is not able to compile code with vector instructions for the MIC architecture, which makes it unsuitable for high performance computing applications.

1.2.5. Networking with Coprocessors in Clusters

Intel Xeon Phi coprocessors do not have onboard Ethernet or InfiniBand ports. However, because Intel Xeon Phi coprocessors run an operating system, they are capable of supporting network protocols and communicating with hosts and peers over virtualized Ethernet and InfiniBand fabrics.

The virtual Ethernet fabric with TCP/IP stack support is included in the minimal installation of MPSS (see Section 1.5.3). The virtual fabric works by creating virtual network interfaces (`mic0`, `mic1`, etc.) in the host OS and in the coprocessor OS. Packets sent through these interfaces travel across the PCIe bus, however, this not visible to user applications.

By default, the IP network is configured in the "static pair" topology, so that coprocessors on the system form a private network, in which the host and each coprocessor have an IPv4 address. CPU to MIC is possible within the compute node (Figure 1.11, left). With packet forwarding enabled, MIC to MIC communication also becomes possible. However, in this configuration, TCP/IP packets cannot leave the host OS.

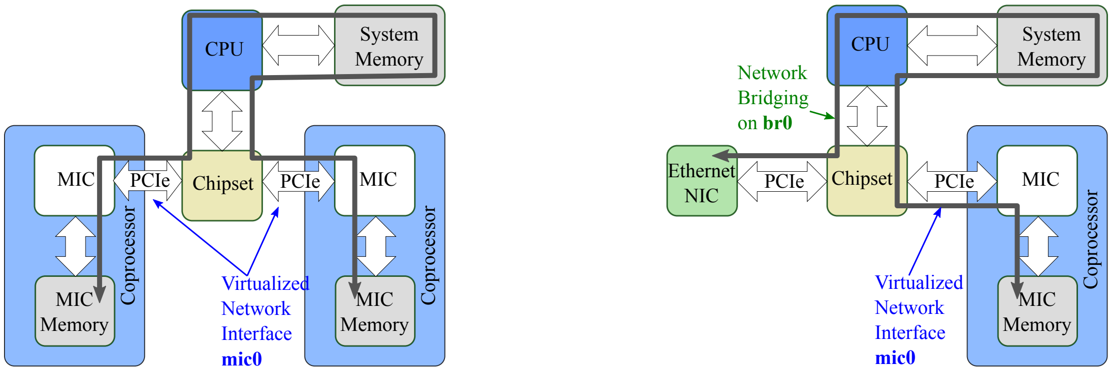


Figure 1.11: Virtualized Ethernet fabric supporting the TCP/IP protocol. Left: communication within the private network of a compute node. Right: network bridging puts coprocessors on the external private network.

It is also possible to configure network bridging in software, and have the coprocessors join the private network of the host. In this setup, messages from/to Intel Xeon Phi coprocessors travel through the PCIe bus, and get routed by their host OS to/from the NIC (Figure 1.11, right). In this way, it is possible to establish peer-to-peer connections between Intel Xeon Phi coprocessors installed in different machines. This powerful functionality makes all Intel Xeon Phi coprocessors act as independent compute nodes. This allows the developer to take a cluster application designed for CPU-based nodes and run it without modification on MIC-based nodes (see Section 2.1.5).

The virtualized TCP/IP fabric provided by MPSS is useful for administrative tasks, and it can also support data traffic in computing applications. However, as of MPSS 3.4.1, the TCP/IP performance is orders of magnitude slower than the physical limitations of the PCIe bus.

A solution to this bottleneck is the InfiniBand functionality in MPSS:

1. Virtual InfiniBand interface `ib-scif` allows the host and the coprocessor to exchange messages via DMA with latencies and bandwidths close to the PCIe limitations (Figure 1.12, left). Peer-to-peer communication between coprocessors in a system over `ib-scif` is also possible.
2. For machines with physical InfiniBand HCAs, Coprocessor Communication Link (CCL) or Performance Scaled Messaging (PSM) allows Remote Direct Memory Access (RDMA) transfers between coprocessors in different nodes (Figure 1.12, right). CCL is used for Mellanox-branded interconnects, and PSM for Intel True Scale interconnects.

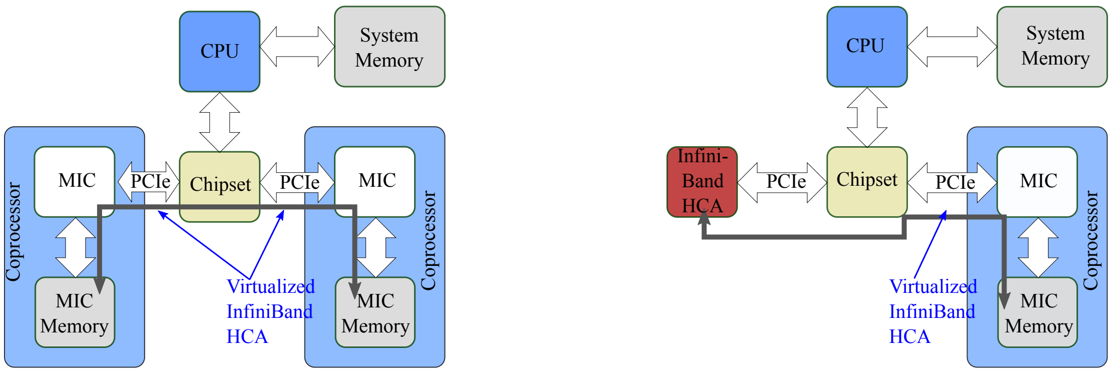


Figure 1.12: Virtualized InfiniBand in MPSS allows DMA transfers between local coprocessors (left) and RDMA (via an InfiniBand HCA) between coprocessors in other compute nodes (right).

CCL and PSM are provided by a special branch of the OpenFabrics Enterprise Distribution (OFED) software suite (Section 1.5.5).

1.2.6. File I/O on Coprocessors

The default configuration of MPSS creates a RAM disk with the `tmpfs` filesystem in the coprocessor's memory (Figure 1.13). This RAM disk stores the operating system files, user applications and third-party libraries. Applications can store scratch data and input/output files in this filesystem. The data stored in the filesystem takes up coprocessor memory, reducing the amount available for applications. The memory-based filesystem is not persistent: all stored data is cleared when the coprocessor reboots.

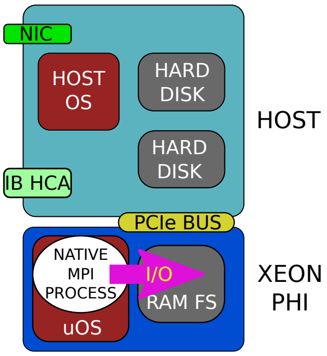


Figure 1.13: RAM disk

Intel Xeon Phi coprocessors can also assume control over the host's physical drives and partitions (Figure 1.14) using the VirtIO block device functionality (see Section 1.5.20 for more detail). VirtIO extends the amount of disk space that the coprocessor can access, and allows read and write access to persistent data storage devices. The performance of VirtIO-shared filesystems is lower than that of the RAM disk.

Finally, thanks to the virtualization of networking on Intel Xeon Phi coprocessors (Section 1.2.5), it is possible to mount network-attached storage inside the coprocessor OS (Figure 1.15). The common NFS protocol is supported over the virtualized TCP/IP fabric, and the scalable Lustre protocol is available over the virtualized InfiniBand fabric.

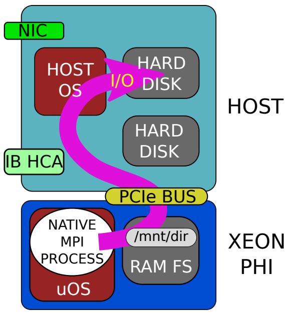


Figure 1.14: VirtIO

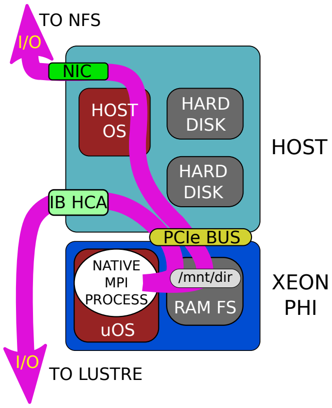


Figure 1.15: Network storage

1.2.7. Common Software Development Tools

Intel supports the cross-compatibility of the multi-core and manycore platforms by maintaining common software development and runtime tools for processors and coprocessors, including

i) Compilers: Intel C, C++ and Fortran compilers;
ii) Optimization tools: Intel VTune Amplifier XE and Intel Intel Trace Analyzer and Collector (ITAC);
iii) Mathematics support: Intel Math Kernel Library (MKL);
iv) Parallelization libraries: Intel MPI and Intel OpenMP,

and others. Figure 1.16 shows some of the tools supporting both platforms.

Figure 1.16: Most Intel software development products support multi-core Intel Xeon CPUs as well as the Intel MIC architecture.

Common development tools for the Intel Xeon and Intel Xeon Phi product families are available as a bundle in Intel Parallel Studio XE (Figure 1.17).

Intel Parallel Studio XE comes in three editions: composer, professional and cluster. Composer is the basic edition, and it includes the Intel compilers and optimized libraries like Intel MKL. Professional edition includes everything for a single node development. In addition to everything in composer, it also includes optimization tools such as Intel VTune Amplifier XE. Cluster is the full edition. It includes everything in professional as well as libraries necessary for cluster development, such as Intel MPI. Table Table 1.4 shows the packaging of the three editions. For a full listing of available features in each edition, refer to the product page for Intel Parallel Studio XE.

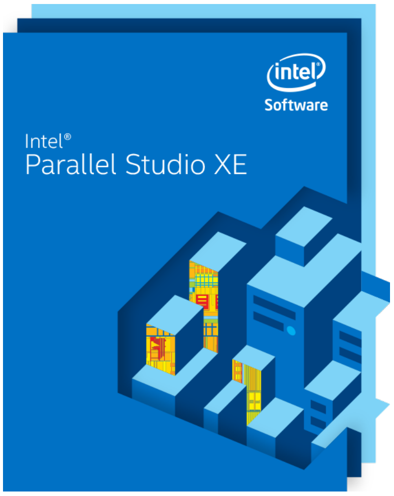


Figure 1.17: Intel software development tool suite for shared memory and distributed memory application design. Intel Xeon processors and Intel Xeon Phi coprocessors are supported by all three suite editions: composer, professional and cluster.

Software \ Edition	**Composer**	**Professional**	**Cluster**
Intel C, C++ and Fortran compilers	x	x	x
Intel Math Kernel Library (MKL)	x	x	x
Intel Threading Building Blocks (TBB)	x	x	x
Intel Performance Primitives (IPP)	x	x	x
Intel VTune Amplifier XE		x	x
Intel Inspector XE		x	x
Intel Advisor XE		x	x
Intel MPI Library			x
Intel Trace Analyzer and Collector (ITAC)			x

Table 1.4: Tools included in the three editions of Intel Parallel Studio XE 2015.

Intel compilers and product suites can be purchased directly from Intel or from one of the authorized resellers. Colfax International is an authorized reseller offering discounts for bundling software licenses with hardware purchases, and academic discounts for eligible customers. For more information, refer to http://www.colfax-intl.com/nd/xeonphi/ or contact sales@colfax-intl.com.

1.2.8. Intel Xeon Processors versus Intel Xeon Phi Coprocessors: Developer Experience

The following is an excerpt from an article "Programming for the Intel Xeon family of products (Intel Xeon processors and Intel Xeon Phi coprocessors)" by James Reinders, Intel's Chief Evangelist and Spokesperson for Software Tools and Parallel Programming [5].

Forgiving Nature: Easier to port, flexible enough for initial base performance
Because an Intel Xeon Phi coprocessor is an x86 SMP-on-a-chip, it is true that a port to an Intel Xeon Phi coprocessor is often trivial. However, the high degree of parallelism of Intel Xeon Phi coprocessors requires applications that are structured to use the parallelism. Almost all applications will benefit from some tuning beyond the initial base performance to achieve maximum performance. This can range from minor work to major restructuring to expose and exploit parallelism through multiple tasks and use of vectors. The experiences of users of Intel Xeon Phi coprocessors and the "forgiving nature" of this approach are generally promising but point out one challenge: the temptation to stop tuning before the best performance is reached. This can be a good thing if the return on investment of further tuning is insufficient and the results are good enough. It can be a bad thing if expectations were that working code would always be high performance. There is no free lunch! The hidden bonus is the "transforming- and-tuning" double advantage of programming investments for Intel Xeon Phi coprocessors that generally applies directly to any general-purpose processor as well. This greatly enhances the preservation of any investment to tune working code by applying to other processors and offering more forward scaling to future systems.

Transformation for Performance
There are a number of possible user-level optimizations that have been found effective for ultimate performance. These advanced techniques are not essential. They are possible ways to extract additional performance for your application. The "forgiving nature" of Intel Xeon Phi coprocessors makes transformations optional but should be kept in mind when looking for the highest performance. It is unlikely that peak performance will be achieved without considering some of these optimizations:

- Memory Access and Loop Transformations (e.g., cache blocking, loop unrolling, prefetching, tiling, loop interchange, alignment, affinity)
- Vectorization works best on unit-stride vectors (the data being consumed is contiguous in memory). Data Structure Transformations can increase the amount of data accessed with unit-strides (such as AoS (Array of Structures) to SoA (Structure of Arrays) transformations or recoding to use packed arrays instead of indirect accesses).
- Use of full (not partial) vectors is best, and data transformations to accomplish this should be considered.
- Vectorization is best with properly aligned data.
- Large Pages Considerations
- Algorithm selection (change) to favor those that are parallelization and vectorization friendly.

Detailed description of Intel Xeon Phi coprocessor programming models can be found in Chapter 2. A thorough exploration of optimization techniques mentioned in this quote is undertaken in Chapter 4.

1.3. Applicability of the MIC Architecture

Not all algorithms and applications are expected to be highly efficient on Intel Xeon Phi coprocessors even with the best optimization effort. For a number of applications, general-purpose processors will remain a better option. This section discusses the properties of applications and algorithms that identify applications as MIC-friendly or not.

1.3.1. Task Parallelism

Intel Xeon Phi coprocessors contain up to 61 cores clocked at approximately 1 GHz, whereas Intel Xeon processors of comparable TDP can have up to 12 cores clocked at around 3 GHz (see Section 1.1.4 and Section 1.1.5). A single lean, low-clock speed KNC cannot deliver better performance than a hardware-rich, high-clock speed CPU core. In fact, the key to good performance on Intel Xeon Phi coprocessors is parallelism.

Multiple cores can be utilized either by running multiple independent processes (for instance, MPI ranks) or a single process with multiple parallel threads. Coprocessor applications should have good scalability up to at least 100 processes or threads, considering the 4 hardware threads per KNC core.

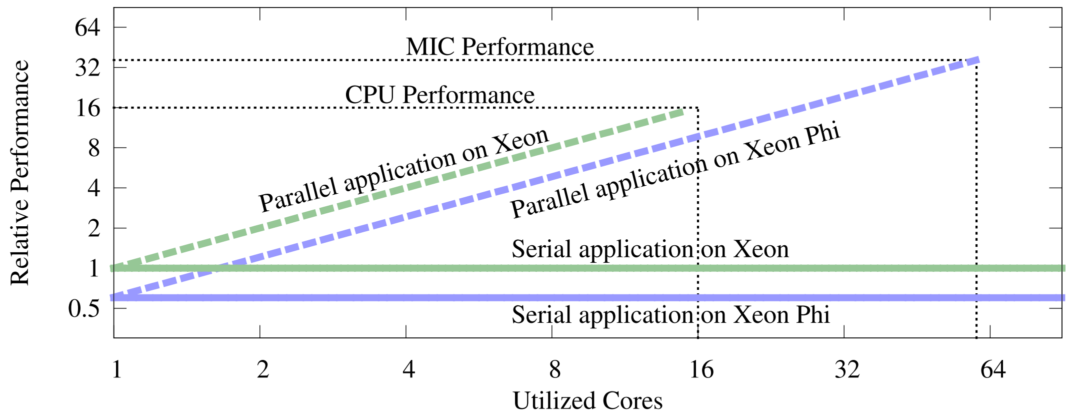


Figure 1.18: Task parallelism (multi-threading or running multiple independent processes) is a prerequisite for good performance with Intel Xeon Phi coprocessors. Serial applications are best executed on the CPU.

Figure 1.18 is an illustration of the necessity for parallelism necessary of Intel Xeon Phi utilization. It shows a synthetic performance of a serial and of a perfectly scalable parallel application as a function of the number of utilized cores on a multi-core CPU and on a MIC architecture coprocessor. While single-core performance on the MIC architecture is lower than on a CPU, having more cores allows the coprocessor to perform better.

Examples

1. Compilation of a programming language is an example of a task more suited for Intel Xeon processors than for Intel Xeon Phi coprocessors, because compilation involves inherently sequential algorithms.

2. Monte Carlo simulations are well-suited for Intel Xeon Phi coprocessors because of their inherent massive parallelism. See, however, a comment in Section 1.3.2.

1.3.2. Data-Parallel Component

Comparing the number of clock cycles per second issued by a 60-core coprocessor (60 cores $\times$ 1 GHz) to the same metric of a two-socket host system with 8-core processors (2 $\times$ 8 $\times$ 3.4 GHz), one can see that the coprocessor is making only some 10% more clock cycles than the host. However, an Intel Xeon Phi coprocessor can provide significantly greater arithmetic performance than the host system. Why?

The answer is vector operations. While Sandy Bridge cores support AVX instructions with support for 256-bit wide SIMD registers, KNC cores support IMCI instructions with 512-bit wide SIMD registers. Neglecting other aspects of performance, such as memory bandwidth, pipelining and transcendental arithmetic performance, an Intel Xeon Phi coprocessor architecture has the capability to perform roughly twice as many arithmetic operations per second as an Intel Xeon CPU.

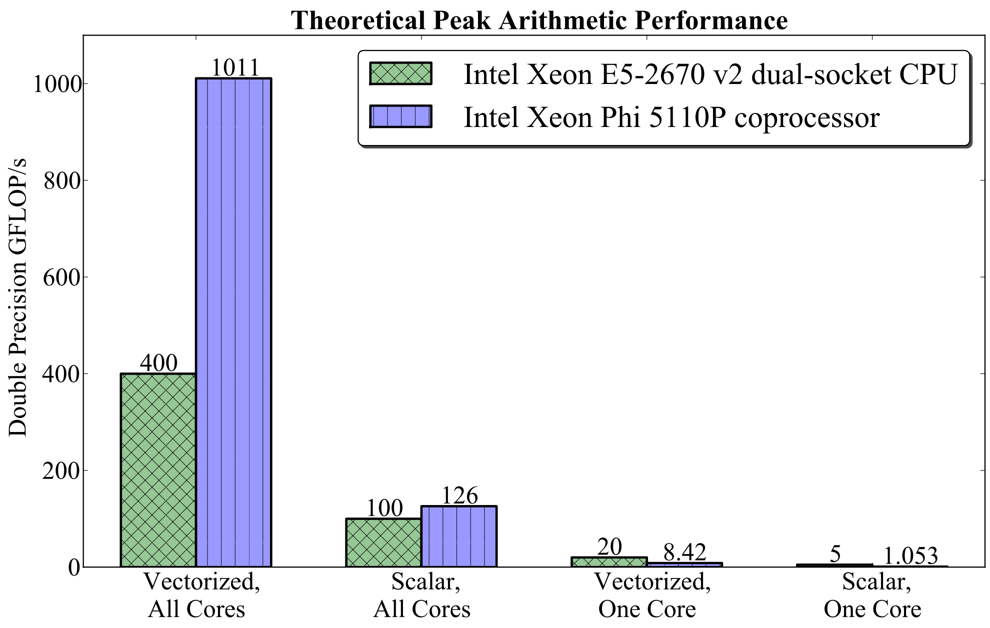


Figure 1.19: Theoretical peak arithmetic performance of an Intel Xeon Phi coprocessor and an Intel Xeon processor of comparable TDP for scalar/vectorized and single/multi-threaded applications.

The flip-side of this architectural difference is also true. Suppose a potentially vectorizable calculation is running in scalar (i.e., non-SIMD) mode on a Sandy Bridge architecture CPU in single precision. This calculation is experiencing up to 256 / (8*`sizeof(float)`)=8-fold performance penalty for failure to employ vector arithmetic. On KNC, the penalty for the lack of vectorization is a factor or 512 / (8*`sizeof(float)`)=16.

Therefore, if a compute-bound application does not employ vectorization, it is unlikely to exhibit better performance on an Intel Xeon Phi coprocessor than on an Intel Xeon processor. See Section 3.1 for more information about vectorization.

Figure 1.19 is a visual pointer to the need for SIMD parallelism (vectorization) in KNC workloads.

Examples

1. Monte Carlo algorithms may be well-suited for execution on Intel Xeon Phi coprocessors if they either use vectors for calculations inside of each Monte Carlo iteration, or perform multiple simultaneous Monte Carlo iterations using multiple SIMD lanes.

2. For common linear algebraic operations, there exist SIMD-friendly algorithms and implementations. These calculations are well-suited for Intel Xeon Phi coprocessors.

1.3.3. Memory Access Pattern

Memory streaming in Intel Xeon Phi coprocessors is fast. The theoretical limit of memory bandwidth is 384 GB/s and practical performance is up to a half of that. This is 2-3x as fast as the memory bandwidth in a two-way system with Intel Xeon processors. Therefore, for applications in which memory access pattern is streamlined, Intel Xeon Phi coprocessors can yield better performance than Intel Xeon processors.

However, streaming memory bandwidth is irrelevant in cases where memory access pattern is complicated. In these cases, performance is limited by memory access latency or cache performance.

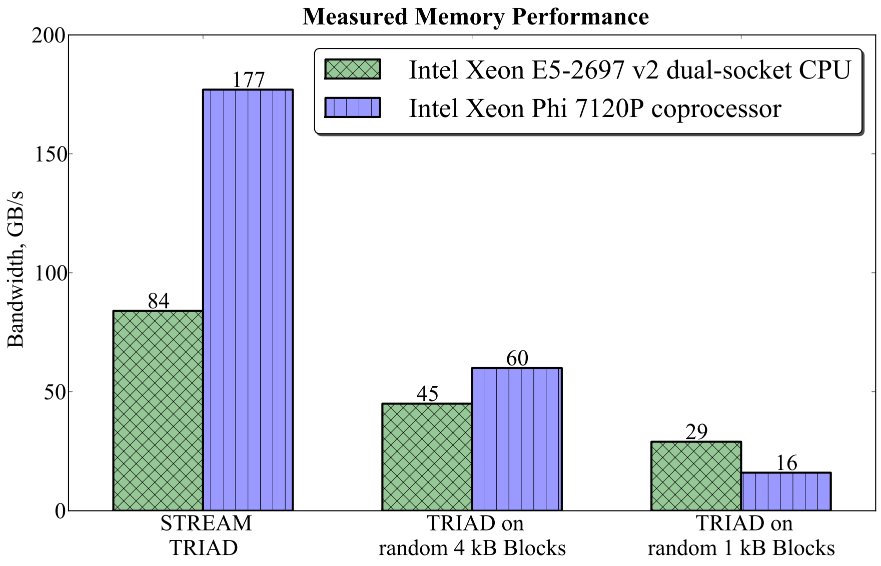


Figure 1.20: Performance with streaming and random memory access.

Figure 1.20 illustrates the difference that randomness in the memory access pattern makes in the performance of an Intel Xeon processor and an Intel Xeon Phi coprocessor. The first set of bars illustrates the performance of the industry-standard STREAM benchmark, the "triad" test. In this benchmark, memory is read and written in a contiguous pattern. The second and third set of bars demonstrate the "triad" operation on randomly scattered blocks of memory (one thread per block). The second set of bars is for

4 kilobyte blocks, and the third is for 1 kilobyte blocks. While the MIC architecture has clear advantage in the streaming case, for random accesses it performs only as good as, or even much worse, than the CPU.

At the same time, the KNC architecture has specialized instructions for strided memory accesses. Fixed-stride write accesses, called "scatters", and fixed-stride read accesses, called "gathers", are optimized to deliver greater bandwidth than just random accesses.

Considering the above facts, one can only expect a better performance from an Intel Xeon Phi coprocessor than from an Intel Xeon processor in two cases:

a) the data set is so small, or the arithmetic intensity (number of operations on every word fetched from memory) is so high that memory performance is irrelevant — the compute-bound case, or
b) memory access pattern is streamlined enough so that the application is limited by memory bandwidth and not memory latency — the bandwidth-bound case.

Multi-threading is as important for bandwidth-bound applications as it is for compute-bound workloads, because all available memory controllers must be utilized. On Intel Xeon Phi coprocessors, special measures on thread binding to cores must be taken in order to optimize for bandwidth, just like with Intel Xeon CPUs. See also Section 4.5 for a discussion of memory and cache traffic tuning.

Examples

1. Monte Carlo calculations with a small size of geometry or physics data may have small memory footprints. Consequently, they are not sensitive to the memory performance;
2. Some stencil operations found in Computational Fluid Dynamics (CFD), image processing, and heat transport simulations have streaming memory access pattern. These algorithms are a good match for the Intel Xeon Phi coprocessor architecture.
3. Applications with complex data structures (e.g., graphs with relatively small amount of data in the vertices) may need to re-arrange or pack data in such a way that data locality is improved in the actual data access pattern.

1.3.4. PCIe Bandwidth Considerations

When data needs to be transferred across the PCIe bus from the host to the coprocessor, or between coprocessors, consideration should be given to the data transfer time. Depending on *how much work* will be done on the coprocessor with the data before the result is returned, and depending on how quickly this work will be done, the usage of the coprocessor may or may not be justified.

Ratio of Arithmetic Operations to Data Size

The following technical characteristics can be used to estimate the benefit of coprocessor usage: a practical PCIe v 2.0 bandwidth of ≈6 GB/s, a practical arithmetic performance of 750 GFLOP/s in double precision, and a practical memory bandwidth on the coprocessor of 150 GB/s. The above numbers yield the following "rules of thumb" for identifying situations when the PCIe overhead is insignificant:

a) for compute-bound calculations, if the coprocessor performs many more than than

$$N_a = \frac{750\ \mathrm{GFLOP/s}}{6\ \mathrm{GB/s}/(8\ \text{bytes per number})} \approx 1000 \qquad (1.2)$$

lightweight floating-point operations (additions and multiplications) per data element, then the data transport overhead is insignificant;

b) for compute-bound calculations with expensive arithmetic operations such as division and transcendental functions, the arithmetic intensity threshold at which the communication overhead is justified is lower than 1000 operations per transferred floating-point number;

c) for bandwidth-bound calculations, if the data in the coprocessor memory is read many more than

$$N_s = \frac{150\ \mathrm{GB/s}}{6\ \mathrm{GB/s}} \approx 25\ \text{times} \qquad (1.3)$$

in streaming fashion, then the time of data transport across the PCIe bus will be insignificant compared to the execution time on the coprocessor.

Arithmetic Complexity

In this context, it is informative to establish a link between the complexity of an algorithm and its ability to benefit from Intel Xeon Phi coprocessors. Namely,

a) if the data size is n, and the arithmetic complexity (i.e., the number of arithmetic operations) of the algorithm scales as $O(n)$, such an algorithm may experience a bottleneck in the data transport. This is because the coprocessor performs a fixed number of arithmetic operations on every number sent across the PCIe bus. If this number is too small, the data transport overhead is not justifiable.
b) for algorithms in which the arithmetic complexity scales faster than $O(n)$ (e.g., $O(n \log n)$ or $O(n^2)$), larger problems are likely to be less limited by PCIe traffic than smaller problems, as the arithmetic intensity in this case increases with n. The stronger the arithmetic complexity scaling, the less important is the communication overhead.

Overlapping Communication with Computation

Overlapping data transfer with work on another piece of the data set can potentially increase overall performance by up to a factor of two. To mask data transfer, the asynchronous transfer and asynchronous execution capabilities of Intel Xeon Phi coprocessors can be used. More details are provided in Section 2.2.5.

Data Persistence and Manycore-Hosted Applications

In some cases, data traffic can be reduced or eliminated if some of the data is retained on the coprocessor between offloads. For instance, read-only physics data, or results of one iteration of a simulation used in the subsequent iteration can be retained. This is discussed in Section 2.2.4.

In other cases, if the host CPU is not used in the application, it is possible to have the data living on the coprocessor from initialization until output and destruction. In this case, it may be possible to eliminate data traffic across the PCIe bus completely, except for scheduling or boundary value exchange traffic. This can be achieved through native programming of coprocessors and manycore-hosted MPI applications, as discussed in Section 2.1.

Examples

1. Computing a vector dot-product, when one or both vectors need to be transferred to the coprocessor, is an inefficient use of Intel Xeon Phi coprocessors, because the complexity of the algorithm is $O(n)$, and only 2 arithmetic operations per transferred floating-point number are performed. The PCIe communication overhead is expected to be too high, and such a calculation can be done more efficiently on the host;

2. Computing a matrix-vector product with a square matrix, when only the vector must be transferred to the coprocessor, is expected to have a small PCIe communication overhead if the vector size is large enough (so that the communication latency is unimportant). The algorithm complexity is $O(n^2)$, and therefore each transferred floating-point number will be used n times.

3. FFTs require $O(n \log n)$ operations on every data element, and it is a bandwidth-bound calculation. Therefore, larger FFTs experience less communication overhead than smaller ones.

1.4. Preparing for Future Parallel Architectures

This section summarizes information on upcoming Intel manycore products and provides advice on preparing computing applications for the future.

1.4.1. Exascale Computing for the Rest of Us

Intel Xeon Phi coprocessors based on the Knights Corner microarchitecture are the first generation of Intel products designed specifically for accelerated HPC. However, future highly parallel processors are already in the development pipeline.

“Exascale” is a milestone in computer science anticipated in 2018, where a computing system with a power consumption under 20 MW is able to achieve a performance of 1 exaflop/s (10^{18} floating-point operations per second). This will require a reduction in the energy cost of arithmetic operations by a factor of $\approx$30 compared to today’s typical performance metrics. It will also necessitate the development of memory subsystems and interconnects capable of supporting such performance. In the arena of supercomputing, the path to Exascale will be paved by specialized processors designed for power efficiency and high degree of parallelism, rather than by general-purpose CPUs designed for flexibility in parallel/sequential processing.

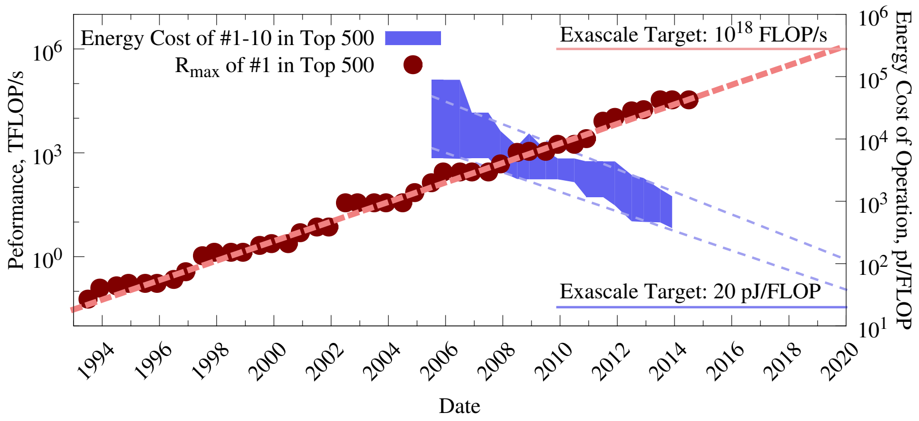


Figure 1.21: By 2018, the trend of CPU-based supercomputing hits the Exascale performance target, but not the energy target. However, highly parallel processors (GPGPUs and MIC architecture) can help to bridge the energy gap.

Figure 1.21 shows historical data of the Top 500 Supercomputer Sites rankings. Two metrics relevant to Exascale are shown in this plot. R_{max}, plotted with red markers, is the maximum performance, in TFLOP/s, achieved in the HPC LINPACK Benchmark on the top-ranked system. The other metric, shown with a blue region, is the range of energy cost of operation in #1 through #10 systems in Top 500. The energy cost was calculated as R_{max}/P, where P is the power consumption reported to the Top 500 list. We also plotted trend lines based on historical data for performance and power efficiency.

The trend lines in Figure 1.21 show that achieving exaflop performance (10^{18} FLOP/s) is a realistic goal. However, fitting an exaflop performance system into the 20 MW power envelope requires better power efficiency than the trend predicts in 2018. It is important that these trend lines are based, predominantly, on CPU-powered systems. A breakthrough technology, such as specialized parallel processors, can break the trend and close the power gap. Consequently, progress in science and technology dependent on big HPC – genomics research, molecular biology, weather modeling, computational fluid dynamics, plasma physics, and other fields – is dependent upon developers continually learning to leverage parallel architectures, including the manycore platform.

At the same time, one does not have to be a prospective Exascale user in order to wonder about the future of high performance platforms. Progress driven by the Exascale challenge will echo on smaller computing resources. Indeed, as of today, vector processors have permeated from big HPC to smaller computing systems, ranging from professional desktop workstations to private computing clusters. This makes computing systems of teraflop per second performance readily available in the form of a GPGPU or an Intel Xeon Phi coprocessor. Likewise, the Exascale era will, in all probability, make petaflop per second computing equally affordable and wide-spread.

Consequently, for software developers, scientists and computing system architects on all scales of computing, it is important to understand how the next generation of parallel processors will affect their workflow and research opportunities.

1.4.2. Second Generation MIC Processor, KNL

The second generation of the Intel manycore architecture will be based on a chip codenamed Knights Landing manufactured with a 14 nm transistor process. The new features expected in KNL are (see disclosures):

1. The second generation chip will be a manycore processor product with more than 60 cores in a 2D mesh architecture, capable of achieving over 3 TFLOP/s in double precision at over 10 GFLOP/J.
2. It will be available in the socket version, i.e., as a standalone bootable processor capable of running an OS.
3. The processor will also be available as a PCIe-connected coprocessor.
4. Second generation MIC processors will have on-package high-bandwidth memory (16 GiB at launch), delivering over 5x the bandwidth of DDR4 for the STREAM benchmark (over 400 GB/s).
5. The on-package memory can be used as a cache for the system memory, or used in a flat memory model.
6. Single-thread performance will be improved by up to 3x over the first generation; multiple hardware threads per core will be used.
7. The processor will support Intel Advanced Vector Extensions 512 (AVX-512) instruction set and feature two VPUs per core.

Figure 1.22: Second generation MIC processor codenamed Knights Landing will be available in the socket version (i.e., as a stand-alone CPU).

Even though at the time of the writing of this book (May 2014), official public information on the second generation is limited, the specification of the

AVX-512 instruction set has been released [6], and Intel has communicated the intent to support existing frameworks (MKL, MPI, OpenMP, OpenCL, as well as Intel Cilk Plus and TBB) with KNL [7]. This information, along with the above mentioned facts and historical trends in Intel products, allow to draw the following two conclusions:

I. Modernization of computing applications for today's Intel Xeon Phi coprocessors is the way to prepare them for Knights Landing.

II. At the same time, developers need to be careful in their choice of programming models. In some cases, incompatibility between first generation (KNC) and second generation (KNL) architectures may impede application porting to the future architecture.

In support of the first statement, note the following facts. 60+ cores with multiple hardware threads will support AVX-512. This is architecturally similar to KNC with up to 61 cores, 4 hardware threads per core, and the IMCI instruction set (see Section 3.1.2). Both instruction sets (IMCI and AVX-512) support 512-bit long vectors and perform operations on single and double precision floating point numbers and integers. Basic arithmetic operations, common transcendental functions, rounding, type conversion and other common vector operations are supported in both instruction sets. Furthermore, on-package high-bandwidth memory is a feature similar to the onboard GDDR5 RAM in first generation Intel Xeon Phi coprocessors (see Section 1.1.1). It is likely that additional usage models of the on-package memory (e.g., the cache model) will expand the range of options available to parallel applications. Still, it will likely be easier to tune a “cache-aware” application optimized for KNC for an additional level of cache in KNL than to optimize a “cache-ignorant” code for data locality in either architecture (see Section 4.5).

The second statement follows from the specification of AVX-512. This instruction set is a superset of IMCI and also backward compatible with the Instruction Set Architecture (ISA) of Intel Xeon processors (SSE, SSE2 and AVX). This means that binaries compiled for Intel Xeon processors can be run on the second generation Intel Xeon Phi processors without recompilation. However, the 256-bit Intel Xeon instructions will not take advantage of the new architecture with 512-bit vectors. That means that at least a recompilation pass will likely be required in order to port applications from Intel Xeon CPUs to the second generation MIC platform. Furthermore, applications compiled for the first generation of Intel Xeon Phi coprocessors will not be binary compatible with the second generation and have to be recompiled. This leads us to a discussion of programming models that simplify code portability to future architectures. See next section for details.

1.4.3. Future-Proof Development Options

Legacy applications that were designed for CPUs in a "future-proof" way have better chances of benefiting from the manycore architecture than applications "hard-wired" to older platforms. Likewise, today's applications developed for Knights Corner architecture will scale to future Knights Landing only if they are designed with portability in mind.

The key to "future-proofing" is using as high level framework as possible for performance-critical parts of the application. For example, an application that relies on BLAS library functions may be accelerated on Intel Xeon Phi coprocessors after linking to the Intel MKL (see Section 5.1). Applications that rely on automatic vectorization by the compiler can be re-compiled with a modern compiler and experience improved performance (see further discussion in Section 4.3). In contrast, applications designed with assembly or intrinsic functions may not be able to run on future architectures, and will have to be re-written.

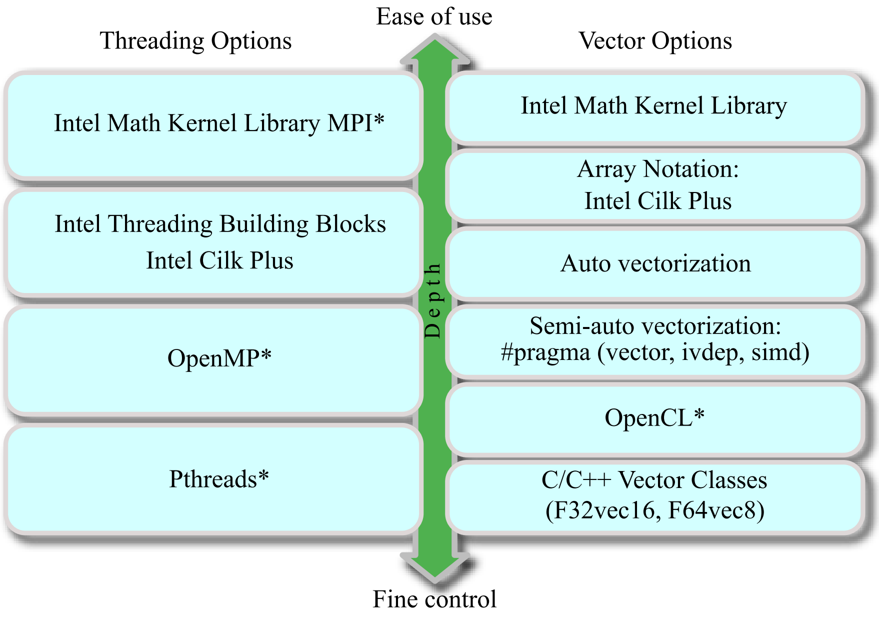


Figure 1.23: Implementation of thread and data parallelism in applications for Intel Xeon processors and Intel Xeon Phi coprocessors designed with Intel software development tools. *Diagram based on materials designed by Intel.*

Figure 1.23 demonstrates the variety of choices for thread and data parallelism implementations in the design of applications for Intel Xeon and Intel Xeon Phi platforms. Depending on the specificity and computing needs of the application, the depth of programmer's control over the code may be chosen from high-level library function calls to low-level threading functionality and Single Instruction Multiple Data (SIMD) instructions. This choice is available in both multi-core and manycore applications.

The focus of this book is on the "future-proof" approach to application programming through the use of the highest-level frameworks. We demonstrate how to design C/C++ codes in such a way that the Intel C Compiler and Intel C++ Compiler can automatically implement thread and data parallelism, with possibility of using recompilation to port the code the from CPU to Intel Xeon Phi coprocessors, and to future manycore architectures. We also discuss using the Intel MKL to rely on a library implementation of common functions.

Even though little has been revealed about planned programming models for Knights Landing, Intel has invested considerable effort into supporting the innovations in OpenMP and MPI specifications and evolving the Intel C, C++ and Fortran compilers as well as Intel MKL. Furthermore, communications from Intel state that automatic vectorization, OpenMP, MPI and MKL, will become if not the leading, but at least reliably supported tools for future platforms, including Knights Landing.

That said, we believe that porting and optimization of today's CPU applications for Knights Corner using the methods that we propose an investment into preparing these applications for Knights Landing and the subsequent hardware innovations leading up to Exascale.

1.5. System Administration with Intel Xeon Phi Coprocessors

This section overviews the process of installing and configuring an Intel Xeon Phi coprocessor and related software. Complete and current procedures can be found in the document "Intel Manycore Platform Software Stack (Intel MPSS) User's Guide". Procedures discussed in this section serve as reference points to the administrator of a system with Intel Xeon Phi coprocessors.

1.5.1. Hardware Compatibility

In general, an Intel Xeon Phi coprocessor requires a system with a free x16 PCIe connector and a BIOS with support for memory-mapped I/O (MMIO) address ranges above 4 GiB. In addition, there must be sufficient cooling, either from the system fans, or from a combination of system fans and the built-in active cooling solution fans on the coprocessor cards. Complete thermal specifications are laid out in Xeon Phi Datasheet.

Ensuring the compatibility of a computing system with Intel Xeon Phi coprocessors is generally the responsibility of the computing system Original Equipment Manufacturer (OEM). Self-installation of Intel Xeon Phi coprocessors into computing systems not validated for usage with these devices is done at the user's risk. If thermal or electrical specifications of the host system are not met, the system or the coprocessor can be irreparably damaged.

Computing systems enabled with Intel Xeon Phi coprocessors provisioned by Colfax International come with Intel Xeon Phi coprocessors and related software and drivers already installed. System configurations validated for use with Intel Xeon Phi coprocessors can be found at http://www.colfax-intl.com/nd/xeonphi/.

1.5.2. Operating Systems

As of MPSS 3.4.1 (Oct 2014), the following operating systems are supported for Intel Xeon Phi coprocessor operation:

1. Red Hat* Enterprise Linux* 64-bit:
   - RHEL 6.3 kernel 2.6.32-279
   - RHEL 6.4 kernel 2.6.32-358
   - RHEL 6.5 kernel 2.6.32-431
   - RHEL 7.0 kernel 3.10.0-123
2. SUSE* Linux* Enterprise Server
   - SLES 11 SP2 kernel 3.0.13-0.27-default
   - SLES 11 SP3 kernel 3.0.76-0.11-default
3. Microsoft Windows:
   - Windows 7 Enterprise SP1
   - Windows 8/8.1 Enterprise
   - Windows Server 2008 R2 SP1
   - Windows Server 2012
   - Windows Server 2012 R2

If an environment is not supported, it means that it has not been validated by Intel. However, it does not mean that Intel MPSS is incompatible with unsupported environments. In fact, we have had success with MPSS installation "out of the box" on all CentOS Linux versions corresponding to the respective supported RHEL versions. Success with other distributions has also been reported (see, e.g., [8]).

Throughout this book, we use MPSS 3.4.1 on CentOS 7.0 Linux, however, most of the system administration procedures discussed here also apply to RHEL 7.0.

Finally, a word of warning on the kernel version. The MIC architecture driver operates as a kernel module. A kernel module must be compiled specifically for the kernel operating on the system, otherwise it will not work. For kernel versions listed above, correctly compiled kernel modules are included in the MPSS distribution. However, if after the installation of MPSS, the user updates the Linux kernel to a newer version, the MPSS will usually stop working. In this case, the MIC module must be recompiled as explained in Section 1.5.6.

1.5.3. Installation and Minimal Configuration of MPSS

Drivers and administrative tools required for Intel Xeon Phi coprocessor operation are included in the MPSS (Intel MIC Platform Software Stack) package, which can be freely downloaded from the Intel MPSS Download Page [9]. The role of MPSS is to boot the Intel Xeon Phi coprocessor, populate its virtual filesystem and start the operating system on the coprocessor, to provide connectivity protocols, and to enable management and monitoring of the coprocessor using specialized tools.

Brief MPSS installation instructions can be found in the file `readme.txt` and detailed instructions in the MPSS User's guide. These files are included in the MPSS archive. For minimal installation of MPSS version 3.4.1, the required configuration steps are illustrated in Listing 1.1. In this and other examples, `lyra` is the hostname of our machine, and `vega` is our Linux username.

```
vega@lyra% su # become root (use 'sudo su' if superuser)
root@lyra% tar xvf mpss-3.4.1-linux.tar # extract the MPSS archive
root@lyra% cd mpss-3.4.1 # go to the MPSS directory
root@lyra% mykernel=`uname -r` # query you kernel version
root@lyra% cp modules/*${mykernel}*.rpm ./ # precompiled modules
root@lyra% yum install --nogpgcheck *.rpm # install MPSS RPMs
root@lyra% modprobe mic # load the MIC module
root@lyra% micctrl --initdefaults # initialize MPSS configuration
root@lyra% systemctl start mpss # start the MPSS service
```

Listing 1.1: Initial configuration of Intel Xeon Phi coprocessor. *If an older version of MPSS is already installed, it must be uninstalled prior to running these instructions.*

The tool `micctrl` creates the system-wide MPSS configuration file `/etc/modprobe.d/mic.conf`, per-card MPSS configuration files in `/etc/mpss/` and per-card directory trees of the Linux OS for the coprocessor as well as images of these trees in `/var/mpss/`. In addition, the hosts file `/etc/hosts` are modified by MPSS. The IP addresses and hostnames of Intel Xeon Phi coprocessors are placed into that file.

To start MPSS and boot coprocessors, system service `mpss` must be started (Listing 1.1). This service can be stopped or restarted in order to disable or re-enable MPSS. For more information on MPSS, refer to Section 1.2.4 and to the MPSS User's Guide [9].

1.5.4. Controlling the MPSS service

To stop, start or restart the MPSS service, the command `service` may be used on RHEL 6.x or `systemctl` on RHEL 7.x as shown in Listing 1.2 and Listing 1.3.

```
root@lyra% service mpss [stop | start | restart | status | unload]
```

Listing 1.2: Controlling the `mpss` service on RHEL 6.x.

```
root@lyra% systemctl [stop | start | restart | status | unload] mpss
```

Listing 1.3: Controlling the `mpss` service on RHEL 7.x.

By default, the MPSS service is not configured to start at boot time. To enable or disable boot time loading of MPSS, use `chkconfig` on RHEL 6.x or `systemctl` on RHEL 7.x (see Listing 1.4 and Listing 1.5).

```
root@lyra% chkconfig mpss [ on | off ]
```

Listing 1.4: Controlling the loading of `mpss` after reboot on RHEL 6.x.

```
root@lyra% systemctl [enable | disable] mpss
```

Listing 1.5: Controlling the loading of `mpss` after reboot on RHEL 7.x.

1.5.5. Integration of MPSS with InfiniBand: OFED

Minimal installation of MPSS, as laid out in Section 1.5.3, will provide a virtualized TCP/IP fabric for communication between the host and the coprocessor. This basic MPSS can be upgraded by installing a special version of the OFED suite. OFED will introduce the following functionality:

1) OFED will provide a driver and management tools for InfiniBand HCAs (colloquially referred to as “InfiniBand cards”) if they are installed in the system. This will enable efficient communication based on RDMA between hosts on the InfiniBand network.

2) The specialized OFED branch with MIC architecture support will provide CCL or PSM. This software layer enables peer-to-peer RDMA communication between Intel Xeon Phi coprocessors on the network. Additional system configuration may be necessary (see, e.g., Section 1.2.5 and [10]) in order to take advantage of CCL or PSM.

3) The special OFED for MIC will also create a virtual InfiniBand interface `ib-scif` for rapid communication between the host and coprocessor(s) within a single compute node or workstation.

Components 1) and 2) are only available in systems with physical InfiniBand HCAs. However, 3) also applies to standalone compute nodes and workstations, even if they do not have an HCA.

Installation of OFED over MPSS depends on the operating system, the brand of InfiniBand cards and MPSS version. Refer to Intel MPSS User’s Guide for details.

1.5.6. Restoring MPSS Functionality after Kernel Updates

Intel MPSS integrates with the operating system by means of a kernel module. This kernel module may become unfunctional after the vendor of the operating systems provides an update to the Linux kernel. In this case, starting MPSS will fail with the following error message:

```
Starting MPSS failed: module MIC not found.
```

and the Intel Xeon Phi coprocessor will be unavailable.

When this happens, there are two solutions:

A) Reboot the system, enter the boot menu and choose to boot the old version of the Linux kernel. Alternatively, the choice of the old kernel may be made permanent by modifying the Grub configuration file `/boot/grub/grub.conf`. This method is a workaround, and should only be used temporarily, until method B can be applied.

B) Rebuild the MIC kernel module.

To rebuild and install the MIC kernel module, superuser access is required. The steps illustrated in Listing 1.6 describe the rebuild process of the kernel module. On RHEL, packages `rpm-build`, `kernel-devel` and `kernel-headers` are required for this procedure. Note that if OFED is installed, in case of kernel updates, the respective kernel modules must also be recompiled. Refer to the Intel MPSS User's guide [9] for details.

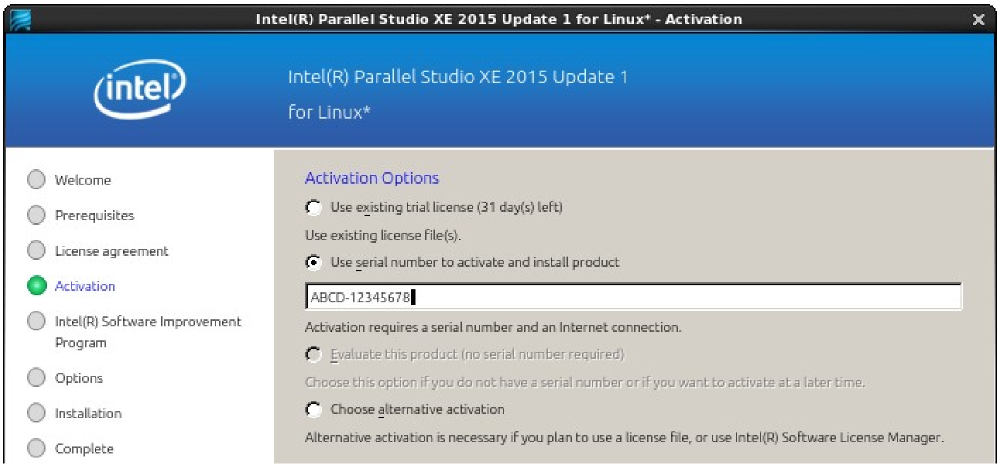
```
root@lyra% systemctl stop mpss
root@lyra% cd mpss-3.4.1/src
root@lyra% # Rebuild the MPSS kernel module:
root@lyra% rpmbuild --rebuild mpss-modules*.rpm
root@lyra% mykernel=`uname -r` # query you kernel version
root@lyra% rpmdir=~/rpmbuild/RPMS/x64_64/mpss-modules
root@lyra% # Fetch the newly rebuilt module
root@lyra% cp ${rpmdir}/mpss-modules*${mykernel}*.rpm ./
root@lyra% yum remove mpss-modules\* # uninstall old modules
root@lyra% yum install mpss-modules*${mykernel}*.rpm # install new
root@lyra% systemctl start mpss # start MPSS
Starting MPSS Stack:                                        [  OK  ]
```

Listing 1.6: Restoring MPSS after Linux kernel update (rebuilding the MPSS kernel module).

1.5.7. Installation of Intel Compilers

Separately from MPSS, a development workstation with an Intel Xeon Phi coprocessor must have Intel software development tools installed. These tools include Intel compilers, parallelization libraries and performance tuning utilities. The products are not a part of the MPSS (see Section 1.2.7) and must be purchased from Intel or from an authorized vendor.

Installation instructions are included with the downloadable software suites. A text-based or GUI-based installer can be used for automatic installation of all the required components. During installation, the user is prompted for a serial number or other license activation option in order to activate the license and support for the tool (Figure 1.24). The serial number or license file is provided to the user at the time of the purchase of the license.


Intel(R) Parallel Studio XE 2015 Update 1 for Linux* - Activation
intel
Intel(R) Parallel Studio XE 2015 Update 1
for Linux*
Welcome
Prerequisites
License agreement
Activation
Intel(R) Software Improvement Program
Options
Installation
Complete

Activation Options

( ) Use existing trial license (31 day(s) left)

Use existing license file(s).

(●) Use serial number to activate and install product

ABCD-12345678

Activation requires a serial number and an Internet connection.

( ) Evaluate this product (no serial number required)

Choose this option if you do not have a serial number or if you want to activate at a later time.

( ) Choose alternative activation

Alternative activation is necessary if you plan to use a license file, or use Intel(R) Software License Manager.

Figure 1.24: Installation and activation of Intel Parallel Studio XE 2015.

To obtain access to the Intel Premier Support service[1], the serial number must be registered at the Intel Registration Center[2] (Figure 1.25). The username and password created during the registration process can then be used for accessing the support service.

[1] http://premier.intel.com/
[2] http://registrationcenter.intel.com/

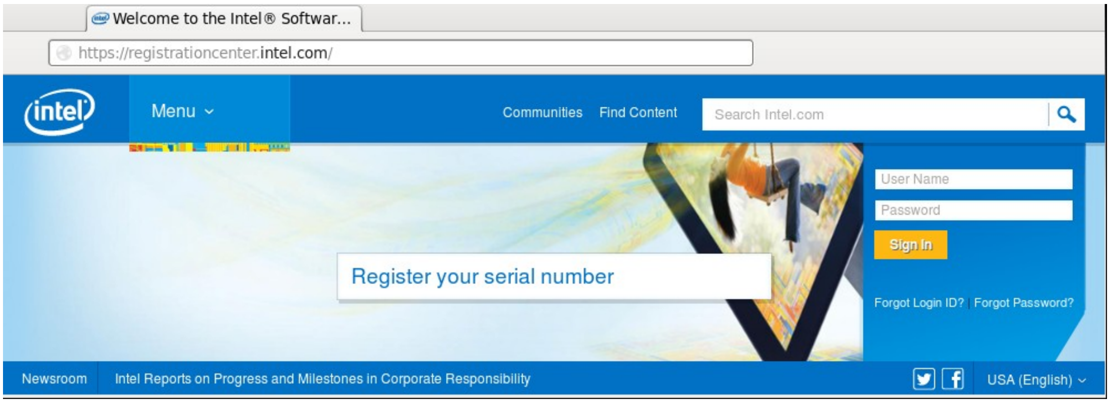

Welcome to the Intel® Softwar...
https://registrationcenter.intel.com/
Menu
Communities
Find Content
Search Intel.com
User Name
Password
Sign In
Register your serial number
Forgot Login ID?
Forgot Password?
Newsroom
Intel Reports on Progress and Milestones in Corporate Responsibility
USA (English)

Figure 1.25: Intel Registration Center: product download and updates.

After installation, it is important to set up the environment variables for Intel Parallel Studio XE as shown in Listing 1.7.

```
vega@lyra% # Option 1: all Intel Cluster Studio tools at once:
vega@lyra% source /opt/intel/parallel_studio_xe_2015/psxevars.sh
vega@lyra%
vega@lyra% # Option 2: individual tools one by one:
vega@lyra% source /opt/intel/composerxe/bin/compilervars.sh intel64
vega@lyra% source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
vega@lyra% source /opt/intel/inspector_xe/inspxe-vars.sh
vega@lyra% source /opt/intel/advisor_xe/advixe-vars.sh
```

Listing 1.7: Enabling environment variables for the `intel64` architecture. The location of the `*-vars.sh` scripts depends on the product suite type and version.

The setup of environment variables using the `compilervars` script can be automated. The automation process depends on the operating system. For example, on RHEL or CentOS, in order to automate loading the script for an individual user, place the command shown in Listing 1.7 into the file `~/.bashrc`. For system-wide enablement, place these commands into `/etc/profile.d/intel.sh`.

1.5.8. Installing the OpenCL Runtime and CodeBuilder

Open Computing Language (OpenCL) is a standard for heterogeneous architecture programming that allows to utilize computing accelerators (GPGPUs and coprocessors) and handle their parallel architecture. Specifically, witn Intel Xeon Phi coprocessors, OpenCL can scale an application across multiple coprocessors and their cores and vector units.

We do not discuss programming in OpenCL in this course, however, for users wishing to work with OpenCL, we provide brief instructions for getting started.

Two pieces of software are necessary to use OpenCL with coprocessors:

1. OpenCL Runtime for Intel CPU and Intel Xeon Phi Coprocessors and
2. Intel Code Builder for OpenCL API

Installation of these products is straightforward. The default installation location of the OpenCL runtime is `/opt/intel/opencl` and the default location of Code Builder is `/opt/intel/opencl-sdk`.

After installation, OpenCL applications may be compiled using either the Intel C++ compiler or the GNU C++ compiler. To link the OpenCL runtime library, pass the compiler argument `-lOpenCL`. To include OpenCL headers, include `<CL/cl.h>`.

Listing 1.8 demonstrates the compilation and runtime output of an example application that queries the OpenCL environment.

Optimization of OpenCL applications for Intel Xeon Phi coprocessors is discussed in the OpenCL Design and Programming Guide for the Intel Xeon Phi Coprocessor.

```
vega@lyra% g++ -lOpenCL -o platformInfo PlatformInfo.cc
vega@lyra% ./platformInfo

Number of platforms: 1
Number of OpenCL devices: 5

Device name:       Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
Device vendor: Intel(R) Corporation
Device type: CPU
OpenCL version supported by the device: OpenCL 1.2 (Build 8)
OpenCL software driver: 1.2.0.8
Parallel compute units: 48
Global memory cache size (bytes): 262144
Global memory cache size (bytes): 134990622720
Local memory cache size (bytes): 32768
Maximum number of work-items in a work-group: 8192
Maximum dimensions that specify the global and local work-item: 3
Maximum number of work-items in each dimension: 8192 8192 8192

Device name: Intel(R) Many Integrated Core Acceleration Card
Device vendor: Intel(R) Corporation
Device type: accelerator
OpenCL version supported by the device: OpenCL 1.2 (Build 8)
OpenCL software driver: 1.2
Parallel compute units: 240
Global memory cache size (bytes): 262144
Global memory cache size (bytes): 12200046592
Local memory cache size (bytes): 32768
Maximum number of work-items in a work-group: 8192
Maximum dimensions that specify the global and local work-item: 3
Maximum number of work-items in each dimension: 8192 8192 8192

// ... (skipped output for 3 more accelerator cards like above)
```

Listing 1.8: Compilation and execution of a sample OpenCL application.

1.5.9. Quick Functionality Check

To verify that an Intel Xeon Phi coprocessor is installed in the system, boot the system and use the Linux tool `lspci`. Depending on the model of the coprocessor and its location on the PCIe bus, the output may be different, but it will look similarly to Listing 1.9.

```
vega@lyra% lspci | grep -i "co-processor"
01:00.0 Co-processor: Intel Corporation Xeon Phi coprocessor 3120
        series (rev 20)
```

Listing 1.9: Using `lspci` to check whether an Intel Xeon Phi coprocessor is installed in the computing system.

When MPSS is running, to verify its functionality and configuration, use the tool `miccheck`:

```
root@lyra% miccheck
MicCheck 3.3-r1
Copyright 2013 Intel Corporation All Rights Reserved

Executing default tests for host
 Test 0: Check number of devices the OS sees in the system ... pass
 Test 1: Check mic driver is loaded ... pass
 Test 2: Check number of devices driver sees in the system ... pass
 Test 3: Check mpssd daemon is running ... pass
Executing default tests for device: 0
 Test 4 (mic0): Check device is in online state and its postcode is
                FF ... pass
 Test 5 (mic0): Check ras daemon is available in device ... pass
 Test 6 (mic0): Check running flash version is correct ... pass
 Test 7 (mic0): Check running SMC firmware version is correct ...
                pass
Status: OK
```

Listing 1.10: Checking the configuration of MPSS.

In case `miccheck` reports that the flash version and/or SMC firmware version is incorrect, a flash operation must be performed using the MPSS tool `micflash` (see documentation for details).

To verify that the compilers have been successfully installed, run the commands show in Listing 1.11. Here, `icc` is the Intel C Compiler, `icpc` is the Intel C++ Compiler, and `ifort` is the Intel Fortran Compiler. If Intel MPI library is installed, the setup of its environment can be checked by probing its wrapper scripts `mpii*`

```
vega@lyra% icc -v
icc version 15.0.1 (gcc version 4.8.2 compatibility)
vega@lyra% icpc -v
icpc version 15.0.1 (gcc version 4.8.2 compatibility)
vega@lyra% ifort -v
ifort version 15.0.1
vega@lyra% mpiicc -v
mpiicc for the Intel(R) MPI Library 5.0 Update 2 for Linux*
Copyright(C) 2003-2014, Intel Corporation.  All rights reserved.
icc version 15.0.1 (gcc version 4.8.2 compatibility)
vega@lyra% mpiicpc -v
mpiicpc for the Intel(R) MPI Library 5.0 Update 2 for Linux*
Copyright(C) 2003-2014, Intel Corporation.  All rights reserved.
icpc version 15.0.1 (gcc version 4.8.2 compatibility)
vega@lyra% mpiifort -v
mpiifort for the Intel(R) MPI Library 5.0 Update 2 for Linux*
Copyright(C) 2003-2014, Intel Corporation.  All rights reserved.
ifort version 15.0.1
```

Listing 1.11: Verifying the installation of the Intel compilers and Intel MPI compiler wrappers.

1.5.10. Overview of Intel MPSS Tools

After installation of MPSS, the system administrator will find a number of utilities for the management and diagnostics of Intel Xeon Phi coprocessors:

`micctrl` is a comprehensive configuration tool for the Intel Xeon Phi coprocessor operating system,

`miccheck` is a set of diagnostic tests for the verification of the Intel Xeon Phi coprocessor configuration,

`micflash` is an Intel Xeon Phi flash memory agent.

`micinfo` is a system information query tool,

`micrasd` is a host daemon logger of hardware errors reported by Intel Xeon Phi coprocessors,

`micsmc` is a utility for monitoring the physical parameters of Intel Xeon Phi coprocessors: model, memory, core rail temperatures, core frequency, power usage, etc.,

`micnativeloadex` a utility for transferring and running native Intel Xeon Phi applications along with their dependencies.

These tools are placed in `/usr/bin` and `/usr/sbin`, and are included in the environment variable `PATH` in most Linux installation. Some of these utilities require superuser privileges.

As usual, the usage and arguments of these tools can be obtained by running the any of the tools with the argument `--help` or by using `man`, as illustrated in Listing 1.12.

```
vega@lyra% micctrl --help
...
vega@lyra% man micsmc
```

Listing 1.12: Obtaining help information on MPSS utilities.

These tools are discussed in the next several sections.

1.5.11. `miccheck`: Basic Troubleshooting

`miccheck` runs a set of diagnostic tests in order to verify the configuration of an Intel Xeon Phi coprocessor system. Its functionality was illustrated in Section 1.5.9.

The following list summarizes the aspects of configuration that `miccheck` can probe, and possible solutions to failures.

1. **Are Intel Xeon Phi coprocessors detected on the PCIe bus?** If no coprocessors are detected, either the system does not have any coprocessors, or it may indicate a hardware problem. You can try reseating the coprocessor(s) in their slots, or moving them to a different available PCIe slot. If that does not help, contact the OEM of the workstation or compute node.

2. **Is the driver (i.e., `module mic`) loaded?** If not, it may indicate a kernel version conflict (see Section 1.5.6), or `modprobe mic` has not been executed.

3. **Has the driver detected all coprocessors that show up on the PCIe bus?** If not, it may indicate hardware/software incompatibility. Contact Intel Premier Support.

4. **Is `service mpss` running?** If not, it may be stopped or unfunctional. Try `service mpss start` or refer to Section 1.5.9.

5. **Does the loaded driver version correspond to the installed MPSS version?** Driver mismatch may occur after an upgrade of MPSS. In this case, an easy solution to load the correct driver is a system reboot.

6. **Are all coprocessors online and reporting post code FF?** If not, the boot process may have stalled. Check `/var/log/messages` and `/var/log/mpssd` for details.

7. **Is the RAS daemon available in devices?** If not, it may indicate misconfigured MPSS. If restarting MPSS does not help, a clean configuration may solve the problem (`micctrl --cleanconfig` and `micctrl --initdefaults`).

8. **If the running flash version of the device(s) correct for the installed MPSS?** If not (which may occur after MPSS upgrade), the coprocessors may need to be flashed (Section 1.5.13).

9. **Is the running SMC firmware version of the coprocessor(s) correct for the installed MPSS?** If not (which may occur after MPSS upgrade), the coprocessors may need to be flashed with correct SMC firmware (Section 1.5.13).

10. **Can coprocessors be pinged?** If not, either the coprocessor OS has not booted (try `service mpss restart`), or there is a problem with the network configuration of the coprocessor (see Section 1.5.21).

11. **Can current user establish SSH connections with coprocessors?** If not, it may indicate a problem with user accounts and/or SSH keys in the coprocessor filesystem (Section 1.5.17).

In MPSS 3.4.1, by default, only a subset of tests is run on all Intel Xeon Phi coprocessors. However, additional tests and subsets of tests and devices can be selected. Run `miccheck --help` for details.

For more advanced troubleshooting, refer to the Intel MPSS for Linux: Troubleshooting Flow Chart.

1.5.12. `micctrl`: Coprocessor OS Configuration

The `micctrl` command is the power state and system configuration administration tool. Functions of `micctrl` include power state commands, configuration file creation and parsing, and modification of individual configuration parameters, such as user accounts, networking setup, authentication, and other. This section provides an overview and explanation of the `micctrl` functionality. Run `micctrl --help` for more details. Starting from MPSS 3.5, Appendix B in MPSS User's Guide also covers command line arguments of `micctrl`.

Booting and Rebooting the coprocessor

The coprocessor can be put in one of the following states using `micctrl`:

"ready" state means that the coprocessor has passed initial power on process, the bootstrap μOS has been loaded from the SMC, and is listening for commands to download and boot the Linux operating system.

"online" state means that the coprocessor has booted the Linux operating system and is ready for user applications.

There are also transitional situations ("booting" or "shutdown") between the above states. The following arguments of `micctrl` allow to query and modify the state of the coprocessor.

`-b` or `--boot` boot (i.e., place in the "online" state) one or more Intel Xeon Phi coprocessors in the "ready" state. The MPSS service must be running;

`-S` or `--shutdown` shutdown (i.e., place in the "ready" state) of one or more coprocessors currently in the "online" state;

`-R` or `--reboot` reboot one or more coprocessors currently in the "online" state (gentle method);

`-r` or `--reset` reset one or more Intel Xeon Phi coprocessors currently in the any state (rough method) and place them in the "ready" state;

`-s` or `--status` show the boot state of Intel Xeon Phi coprocessors in the system;

`-w` or `--wait` wait for one or more Intel Xeon Phi coprocessors to not be in either the *booting* or *shutdown* states.

Global Configuration Manipulation

Configuration of Linux on Intel Xeon Phi coprocessors is controlled by a set of configuration files in `/etc/mpss/`. These files determine the configuration of the boot image in `/var/mpss`. The following arguments of `micctrl` allow global manipulation of configuration and images:

`--initdefaults` Initialize the defaults in Intel Xeon Phi coprocessor configuration files. Used once after MPSS software installation. Creates configuration files in `/etc/mpss` unique to each coprocessor and the directory tree in `/var/mpss`. The MPSS service must not be running.

`--resetconfig` Propagates configuration changes to the image files. Used after changes are made to configuration files. The MPSS service must not be running.

`--resetdefaults` Reset the configuration files back to default. Used if hand editing of files has created undesirable effects. The MPSS service must not be running.

`--cleanconfig` Completely remove all configuration information. Must be followed by `micctrl --initdefaults`.

`--config` Display a human-readable overview of the current configuration of Intel Xeon Phi coprocessors.

User Account Configuration

Specific parameters of configuration can be modified by `micctrl`. A very important subset of this functionality is user account management.

When `micctrl --initdefaults` is initially run, all user accounts present on the host system are replicated in the coprocessor OS image. However, if subsequently the administrator adds or deletes new users on the host, these changes do not automatically make it to the coprocessor. The following arguments of `micctrl` help to propagate these changes.

--useradd | --userdel Add or delete a user in coprocessor OS image.

--groupadd | --groupdel Add or delete a Linux group in OS image.

--passwd Set a user's password on the Intel Xeon Phi coprocessor.

--sshkeys Update a user's SSH keys for passwordless login.

--ldap | --nis Setup or disable LDAP or NIS support for coprocessors.

Network Configuration

The assignment of IP addresses and, possibly, network bridging on Intel Xeon Phi coprocessors can be controlled via `micctrl`:

--addbridge | --delbridge | --modbridge Add, delete or modify a Linux network bridge to establish virtualized connection of Intel Xeon Phi coprocessors with the host NICs.

--network Set the network topology (static pair, static bridged or DHCP bridged) and global configuration parameters (IP addresses, and netmasks, MTU values, gateway).

--addnfs | --remnfs Add or remove an NFS import directive to the file(s) `/etc/fstab` in the Intel Xeon Phi filesystem.

--mac Set the MAC address for a coprocessor. By default, MAC addresses are randomly generated and assigned to coprocessors when `micctrl --initdefaults` is executed. However, in large clusters, MAC address collisions may occur. This argument allows the administrator to have explicit control over the coprocessors' MAC addresses.

Miscellaneous Configuration Parameters

Other arguments of `micctrl` allow the administrator to control various parameters, including:

- Location of configuration files other than default;
- Verbosity level during `micctrl` operation;
- Non-default root device (specific initramfs disk, NFS root)
- Automatic booting;
- Power management options;
- System log location;

and others.

Examples of `micctrl` usage are available in Sections 1.5.17, 1.5.19, 1.5.20 and 1.5.21. Complete list of arguments is displayed by `micctrl --help`; details on any specific argument `--arg` can be obtained by running `micctrl --arg --help`.

1.5.13. `micflash`: Coprocessor Firmware Updates

The primary purpose of the `micflash` tool is to update the firmware in Intel Xeon Phi coprocessor's flash memory. This typically needs to be done after MPSS is updated.

Prior to using the `micflash` utility, the Intel Xeon Phi coprocessor must be put in the *ready* state. After flashing, the host system must be restarted. `micflash` can automatically detect how many coprocessors are installed and what their silicon stepping is. Then it locates the corresponding flash image file and writes it in the coprocessors' flash memory. The basic procedure for flashing the coprocessor is shown in Listing 1.13.

```
root@lyra% service mpss stop
Shutting down Intel(R) MPSS:                                  [OK]
root@lyra% micctrl -r
mic0: resetting
mic1: resetting
root@lyra% micctrl -w
mic0: ready
mic1: ready
root@lyra% micflash -update -device all
No image path specified - Searching: /usr/share/mpss/flash
mic0: Flash image: /usr/share/mpss/flash/EXT_HP2_C0_0390-02.rom.smc
mic1: Flash image: /usr/share/mpss/flash/EXT_HP2_C0_0390-02.rom.smc
mic0: Flash update started
mic1: Flash update started
...
Please restart host for flash changes to take effect
root@lyra% reboot
```

Listing 1.13: `micflash` updates aIntel Xeon Phi coprocessor's current status.

In addition, `micflash` can save and retrieve the current flash image version. For a complete list of `micflash` functions, run `micflash -help`.

WARNING: Multiple instance of `micflash` should never be allowed to access the same Intel Xeon Phi coprocessor simultaneously!

Be careful when manually selecting the flash image file. An incorrect flash image may completely incapacitate ("brick") the device!

1.5.14. `micinfo`: Coprocesssor, Firmware, Driver Info

The `micinfo` tool can be used in order to obtain detailed information about the Intel Xeon Phi coprocessor, installed system and the driver version. To obtain complete information, make sure that `service mpss` is running, and run `micinfo` as the root user.

```
vega@lyra% sudo micinfo
MicInfo Utility Log
Copyright 2011-2013 Intel Corporation All Rights Reserved.
...
System Info
    HOST OS                    : Linux
    OS Version                 : 2.6.32-431.el6.x86_64
    Driver Version             : 3.3-0.1.rc1
    MPSS Version               : 3.3
    Host Physical Memory       : 132052 MB

Device No: 0, Device Name: mic0
  Version
    Flash Version                  : 2.1.02.0390
    SMC Firmware Version           : 1.16.5078
    SMC Boot Loader Version        : 1.8.4326
    uOS Version                    : 2.6.38.8+mpss3.3
...
```

Listing 1.14: Example of `micinfo` tool output.

Using `-listdevices` option provides a list of the Intel Xeon Phi coprocessors present in the system.

```
vega@lyra% micinfo -listdevices
...
deviceId |  domain   | bus# | pciDev# | hardwareId
---------|-----------|------|---------|-----------
       0 |         0 |   83 |       0 |   225D8086
       1 |         0 |   84 |       0 |   225D8086
-------------------------------------------------
```

Listing 1.15: Listing available Intel Xeon Phi coprocessors with `micinfo` utility.

To request detailed information about a specific device, the command line argument `-deviceinfo <number>` should be used. Additionally, the information displayed by this command can be narrowed down by including the option `-group <group name>`. Where valid group names are:

- Version
- Board
- Core
- Thermal
- GDDR

For instance, the following shell command returns the information about the total number of cores on the first Intel Xeon Phi coprocessor, current voltage and frequency:

```
vega@lyra% sudo micinfo -deviceinfo 0 -group Thermal
MicInfo Utility Log
Copyright 2011-2013 Intel Corporation All Rights Reserved.
...
  System Info
    HOST OS                   : Linux
    OS Version                : 2.6.32-431.el6.x86_64
    Driver Version            : 3.3-0.1.rc1
    MPSS Version              : 3.3
    Host Physical Memory      : 132052 MB

Device No: 0, Device Name: mic0

  Thermal
    Fan Speed Control        : On
    Fan RPM                  : 2700
    Fan PWM                  : 50
    Die Temp                 : 57 C
```

Listing 1.16: Printing out detailed information about the thermal status of the first Intel Xeon Phi coprocessor.

1.5.15. `micrasd`: Reliability Monitor, Error Logging

`micrasd` is an application running on the host system to handle and log the hardware errors reported by Intel Xeon Phi coprocessors. It can be run as a service daemon. This tool requires administrative privileges.

The following command starts `micrasd`:

```
root@lyra% service micras start
Enable daemon mode.
Enable Maintenance mode test and repair.
Finish parsing options.
Fri May 30 12:06:18 2014 MICRAS INFO
                        : /usr/bin/micrasd: Running in Daemon Mode.
                                                              [   OK   ]
```

Listing 1.17: `micras` log Intel Xeon Phi coprocessor errors handler.

The errors will be logged into the file `/var/log/micras.log` with the tag "`MICRAS`".

1.5.16. `micsmc`: Real-Time Monitoring Tool

The `micsmc` tool returns information about the physical parameters of the Intel Xeon Phi coprocessor: processor, memory, core rail temperatures; core frequency and power usage. `micsmc` can also be used for viewing error logs, monitoring and connection to Intel Xeon Phi coprocessors, viewing and managing log files; root/admin users can manage per-coprocessor or per-node settings, such as ECC, Turbo Mode, and power states.

The `micsmc` tool operates in two modes: Graphical User Interface (GUI) mode and Command Line Interface (CLI) mode. To invoke the GUI mode, `micsmc` should be executed without any additional parameters (see Figure 1.26).

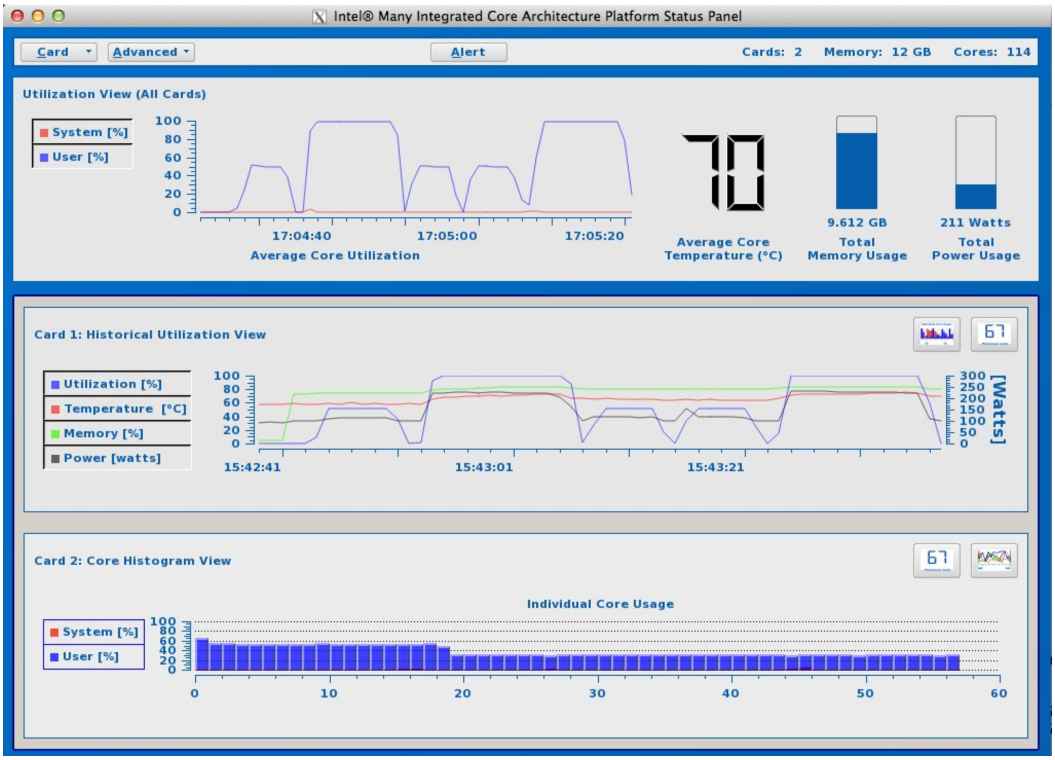


Figure 1.26: The GUI mode of the `micsmc` tool illustrating the execution of a workload on a system with two Intel Xeon Phi coprocessors.

The CLI mode is activated with command-line arguments. This mode produces similar information, but in a one-shot operation, which allows usage in a script environment. `micsmc` invoked in the CLI mode accepts the following arguments:

`-c` or `--cores` returns the average and per core utilization levels for each available board in the system.

`-f` or `--freq` returns the clock frequency and power levels for each available board in the system.

`-i` or `--info` returns general system info.

`-m` or `--mem` returns memory utilization data.

`-t` or `--temp` returns temperature levels for each available board in the system.

`--pwrenable [cpufreq|corec6|pc3|pc6|all]` enables specified respective power management features, disables unspecified

`--pwrstatus` Returns and the status of power management features for each coprocessor,

`--turbo [status|enable|disable]` returns or modifies the Turbo Mode status on all coprocessors

`--ecc [status|enable|disable]` returns or modifies the ECC status on all coprocessors

`-a` or `--all` results in the processing of all valid options, excluding these: `--help`, `--turbo`, and `--ecc`.

`-h` or `--help` displays command specific help information.

Example output in the CLI mode is shown in Listing 1.18.

```
vega@lyra% micsmc -a mic0
mic0 (info):
   Device Series: ... Intel(R) Xeon Phi(TM) coprocessor x100 family
   Device ID: ....... 0x225d
   Number of Cores: . 57
   OS Version: ...... 2.6.38.8+mpss3.3
   Flash Version: ... 2.1.02.0390
   Driver Version: .. 3.3-0.1.rc1 (root@yocto-182-65)
   Stepping: ........ 0x2
   Substepping: ..... 0x0

mic0 (temp):
   Cpu Temp: ................ 55.00 C
   Memory Temp: ............. 44.00 C
   Fan-In Temp: ............. 35.00 C
   Fan-Out Temp: ............ 44.00 C
   Core Rail Temp: .......... 43.00 C
   Uncore Rail Temp: ........ 42.00 C
   Memory Rail Temp: ........ 42.00 C

mic0 (freq):
   Core Frequency: .......... 1.10 GHz
   Total Power: ............. 93.00 Watts
   Low Power Limit: ......... 315.00 Watts
   High Power Limit: ........ 375.00 Watts
   Physical Power Limit: .... 395.00 Watts

mic0 (mem):
   Free Memory: ............. 5454.86 MB
   Total Memory: ............ 5740.88 MB
   Memory Usage: ............ 286.02 MB

mic0 (cores):
   Device Utilization: User: 0.00%, System:  0.09%,  Idle:  99.91%
   Per Core Utilization (57 cores in use)
      Core #1:   User:   0.00%,   System:   0.25%,   Idle:  99.75%
      Core #2:   User:   0.00%,   System:   0.25%,   Idle:  99.75%
...
```

Listing 1.18: `micsmc` output: system information, health, utilization.

1.5.17. User Management on Intel Xeon Phi Coprocessors

About Linux SSH Leys

The Linux OS on the Intel Xeon Phi coprocessor supports SSH access for all users, including `root`, using public key authentication. For each user, the public SSH key files found in the user's `/home/user/.ssh` directory (or for `/root/.ssh` for the `root` user). Listing 1.19 demonstrates the creation of a new user `john` and generation of his SSH key pair.

The SSH key pair consists of two files: the public key stored in the file `/home/john/.ssh/id_rsa.pub` and the private key stored in `/home/john/.ssh/id_rsa`. The private key must be securely stored in `john`'s home directory. The public key, on the contrary, may be freely shared with anybody. When an administrator wants to give `john` access to a server (in our case, to an Intel Xeon Phi coprocessor), the administrator adds the public key to the file `/home/john/.ssh/authorized_keys` *on the server* (i.e., on the coprocessor). During authentication, the SSH server will verify, using encrypted traffic, that `john`'s private key (stored on his machine) matches the authorized public key (stored on the server or coprocessor).

An additional, optional security feature for an SSH key is the passphrase. The user must enter the passphrase to use the key pair. This protects the user's security in case their private key file is stolen. However, it is safe to leave the passphrase blank if the theft of the private key file is not a likely security threat in the system.

Public keys work as a secure replacement for passwords and allow convenient passwordless logins. For Intel Xeon Phi coprocessors, SSH keys serve as more than a convenience feature. They open doors to MPI clients trying to access coprocessors in native applications, and therefore, are required for this use case.

```
root@lyra% adduser john
root@lyra% passwd john
Changing password for user john.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
root@lyra% su john
john@lyra% ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/john/.ssh/id_rsa):
Created directory '/home/john/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/john/.ssh/id_rsa.
Your public key has been saved in /home/john/.ssh/id_rsa.pub.
The key fingerprint is:
e8:5f:9d:d1:07:7e:a8:42:9a:df:f0:7b:d2:fb:bd:6e john@lyra
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|              .  |
|       .     o o |
|      . S . . + o|
|     .   + . + o |
|      . o + +.   |
|       . o =. oE.|
|        . . +++=+|
+-----------------+
john@lyra% cat /home/john/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA1OA7RRyNZYu2KKjMekXioCUS+3sQmjp
OFl606n45F8C2shzaZajqda1NduVnugiyInm/Z96wt5g2UJbd7g7Fb7Vf+7Vc/BBtO7
OiefK+slX+mmwkSyfvmslK+loJHlsK45ih9jrFWgMHa/TwZ1m4UNn2lAzNWS0KxbVA5
ifql7G7uptoJnGzypW8SoZFokEXTg3sOSNn4FouxRCR7jLwSUEch+vwx1vDEYtbPF9Z
YuRWgxBEJJ+fo7CMzIPOhRV9PaEzogADkU+y1DEAGuNON5//bhVU92hiRRKBhbVvjVx
QCM0YGb/VpVjBNwgHhzF96sTV4sGnyoVfEieVXOhjAQ== john@lyra
```

Listing 1.19: Creating a new user generating their SSH keys.

Initial Configuration of User Accounts in MPSS

If MPSS has not been configured yet, then adding the user `john` to the MIC filesystem will be performed automatically when the administrator executes `micctrl --initdefaults`. Moreover, the `micctrl` utility will automatically add `john`'s public key to his `authorized_keys` file, so that `john` can log in to the coprocessor without a password. See Listing 1.20 for an illustration.

```
root@lyra% # Check whether john's home directory is present in the
root@lyra% # MIC filesystem. It is not there at this stage.
root@lyra% ls /var/mpss/mic0/home/
micuser
root@lyra% micctrl --initdefaults
...
root@lyra% # What about now? Yes.
root@lyra% ls /var/mpss/mic0/home/
john  micuser
root@lyra% ls /var/mpss/mic0/home/john/.ssh/
authorized_keys  id_rsa  id_rsa.pub
root@lyra% cat /var/mpss/mic0/home/john/.ssh/authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA1OA7RRyNZYu2KKjMekXioCUS+3sQmjp
...
QCM0YGb/VpVjBNwgHhzF96sTV4sGnyoVfEieVXOhjAQ== john@lyra
root@lyra% # john's public SSH key has made it to authorized_keys
root@lyra% # Can john log in to the coprocessor?
root@lyra% service mpss start
Starting Intel(R) MPSS:                                     [  OK  ]
mic0: online (mode: linux image: /usr/share/mpss/boot/bzImage-kni...
root@lyra% su john
john@lyra% ssh mic0
Warning: Permanently added 'mic0,172.31.1.1' (RSA) to the list of
known hosts.
john@mic0% # We are on the coprocessor!
```

Listing 1.20: Creating a new user `john` and configuring MPSS from scratch to add this account and its SSH keys to the MIC filesystem.

If home directory on the coprocessor is NFS-shared (see Section 1.5.19), then `micctrl` will only create the user. To log in, the user's public SSH key must be in the `authorized_keys` file on the host.

Adding Users to Already Configured MPSS

Suppose that now the administrator wants to add a new user account `mary`. Listing 1.21 shows that adding the account to the host does not propagate it to the coprocessor.

```
root@lyra% adduser mary
root@lyra% passwd mary
...
root@lyra% su mary
mary@lyra% ssh-keygen
...
root@lyra% # Is mary on the coprocessor, too? No.
root@lyra% ls /var/mpss/mic0/home/
john  micuser
```

Listing 1.21: Creating a new user `mary`. The new account does not automatically propagate to the MIC filesystem.

There are two ways to add `mary` to the coprocessor filesystem and allow her to access it:

1. A drastic, but reliable method is to run `micctrl --cleanconfig` and `micctrl --initdefaults` (Listing 1.22). This will add all host users, including `mary`, to the coprocessor. However, any additional customization of MPSS will be gone after `--cleanconfig`.

```
root@lyra% service mpss stop
Shutting down Intel(R) MPSS:                                  [  OK  ]
root@lyra% micctrl --cleanconfig
root@lyra% micctrl --initdefaults
...
root@lyra% service mpss start
Starting Intel(R) MPSS:                                       [  OK  ]
mic0: online (mode: linux image: /usr/share/mpss/boot/bzImage...
```

Listing 1.22: Reconfiguring MPSS from scratch to add a new user `mary` to the coprocessor operating system.

The second method is to use `micctrl --adduser` to just add the new user, as shown in Listing 1.23. This will preserve the MPSS configuration. Still, a restart of the MPSS service is required in this case.

```
root@lyra% service mpss stop
Shutting down Intel(R) MPSS:                                   [  OK  ]
root@lyra# id -u mary # Find mary's userid
502
root@lyra# id -g mary # Find mary's groupid
503
root@lyra% micctrl --adduser=mary --uid=502 --gid=503
root@lyra% # Is mary on the coprocessor, now? Yes!
root@lyra% ls /var/mpss/mic0/home/
john  mary  micuser
root@lyra% service mpss start
Starting Intel(R) MPSS:                                        [  OK  ]
mic0: online (mode: linux image: /usr/share/mpss/boot/bzImage-kni...
root@lyra% su mary
mary@lyra% ssh mic0
Warning: Permanently added 'mic0,172.31.1.1' (RSA) to the list of
known hosts.
mary@mic0% # We are on the coprocessor!
```

Listing 1.23: Propagating a new user `mary` to already configured MPSS.

Note that we had to explicitly specify the userid and groupid of the new user. Consistency between host and coprocessor usernames is required for correct Linux permissions when the administrator NFS-shares directories, including `/home`. See Section 1.5.19 for more information about NFS.

1.5.18. SSH Client Configuration

In Listing 1.20 and Listing 1.23, the SSH client printed out a warning: “Permanently added ‘mic0,172.31.1.1’ (RSA) to the list of known hosts”. This message indicates that the host has not had SSH sessions with `mic0` before, and therefore it added the server key from `mic0` to the list of known hosts in the file `/home/john/.ssh/known_hosts` (or `/home/mary/.ssh/known_hosts`). In subsequent connections, this message will not appear.

Note that this is not the default behavior. Normally, the SSH client asks whether the user wishes to trust this new host. The user must answer “Y” in the interactive prompt, and only then will the SSH session proceed. This default behavior is undesirable in systems with many Intel Xeon Phi coprocessors, because MPI jobs on coprocessors will not run until all coprocessors are present in the known hosts file.

To change the default behavior and enable mute adding of hosts to the users’ known hosts file, we modified the global SSH client configuration file and set “`StrictHostKeyChecking no`” in it. Note that this may be a security threat in some systems.

An alternative way of dealing with the server key checking is a fine-grained approach that disables strict host key checking only for certain trusted hosts. See SSH documentation for more details.

1.5.19. NFS Mounting a Host Export

Because the operating system on Intel Xeon Phi coprocessors is a regular Linux operating system, it supports the NFS protocol. NFS-sharing a directory between a host (or a storage server) and coprocessors is a convenient way to enable file I/O from applications running on the MIC architecture.

NFS export from the host to the coprocessor is configured in the same way as NFS exports between regular Linux hosts. The `root` user on host must allow export of the shared folder by including the respective line in the `/etc/exports` file. In addition, the host firewall needs to be configured to allow traffic from coprocessors on the following ports:

1) tcp/udp port 111 - RPC 4.0 portmapper
2) tcp/udp port 2049 - NFS server

Alternatively, if all traffic to and from Intel Xeon Phi coprocessors is trusted, the administrator can configure the firewall to enable all traffic from interfaces `mic0`, `mic1`, etc. For instance, on RHEL, with the firewall `iptables`, it can be done by adding the following rules:

```
-A INPUT -s mic0 -j ACCEPT
-A INPUT -s mic1 -j ACCEPT
-A FORWARD -s mic0 -j ACCEPT
-A FORWARD -s mic1 -j ACCEPT
```

Listing 1.24: `iptables` rules to enable all TCP/IP traffic packets from coprocessors.

On the client, an NFS share can be mounted either manually using the command `mount`, or automatically by including the corresponding line in the file `/etc/fstab`. The latter task can be performed using `micctrl`.

Example

As an example, let us illustrate how to share the Linux home folder `/home` with all Intel Xeon Phi coprocessors. The mount point on coprocessors will be the same as on the host, i.e, `/home`.

First of all, let us ensure that all the necessary services are started (Listing 1.25). You will need to install the package `nfs-utils` if some of these services are missing.

```
root@lyra% service rpcbind status
rpcbind (pid  3495) is running...
root@lyra% service nfslock status
rpc.statd (pid  3624) is running...
root@lyra% service nfs status
rpc.svcgssd is stopped
rpc.mountd (pid 3864) is running...
nfsd (pid 3927 3926 3925 3924 3923 3922 3921 3920) is running...
rpc.rquotad (pid 3860) is running...
root@lyra%
```

Listing 1.25: Verifying the NFS server status on the host.

To enable sharing /home, in the host file /etc/exports the line shown in Listing 1.26 must be present. This line should be added in a text editor. Note that spaces (and absence of spaces) are important in the syntax of this line, so it should be typed exactly as shown, except for the hostnames. The hostnames should be chosen according to the system configuration. After updating the file /etc/exports, the command exportfs -ra should be executed in order to pass the modifications to the NFS server.

```
root@lyra% cat /etc/exports
/home mic0(rw,no_root_squash) mic1(rw,no_root_squash)
root@lyra% exportfs -ra
```

Listing 1.26: Text to append to /etc/exports on host.

Now we can attempt to log in and mount the home directory from the coprocessor as shown in Figure 1.27. Note that in the arguments of mount, the hostname host is a predefined hostname in /etc/hosts on each coprocessor, pointing to the physical machine hosting the coprocessors.

Instead of mounting the share from the system hosting the coprocessor, we could have mounted a directory from a remote host. This can be done by replacing host with the IP address of the remote NFS server. Bridged networking would need to be configured for that (Section 1.5.21).

If an error occurs at this stage, return to the host and verify the file

/etc/exports and the firewall configuration.

```
root@lyra% ssh mic0
root@mic0% mount -t nfs host:/home/ /home
root@mic0% mount | grep home
host:/home/ on /home type nfs (rw,relatime,vers=3,rsize=1048576,
wsize=1048576,namlen=255,hard,proto=tcp,port=65535,timeo=70,
retrans=3,sec=sys,local_lock=none,addr=172.31.1.254)
```

Listing 1.27: Manually mounting /home on the coprocessor `mic0`.

The directory mounted as in Listing 1.27 will no longer be mounted after the coprocessor reboots. To automatically mount `/home` when the coprocessor boots, we can include the corresponding entry to `/etc/fstab` in the coprocessor filesystem. This can be done using the tool `micctrl`:

```
root@lyra% service mpss stop
Shutting down Intel(R) MPSS:                                [  OK  ]
root@lyra% sudo micctrl --addnfs=host:/home --dir=/home # Add entry
[Warning] mic0: Server host may not be reachable if the interface
is not routed out of the host
[Warning] Modified existing NFS entry for MIC card path '/home'
[Warning] mic1: Server host may not be reachable if the interface
is not routed out of the host
[Warning] Modified existing NFS entry for MIC card path '/home'
root@lyra% cat /var/mpss/mic0/etc/fstab # Verify fstab modification
rootfs      /         auto     defaults         1 1
proc        /proc     proc     defaults         0 0
devpts      /dev/pts  devpts   mode=0620,gid=5  0 0
host:/home  /home     nfs      defaults                1 1
root@lyra% service mpss start
Starting Intel(R) MPSS:                                     [  OK  ]
mic0: online (mode: linux image: /usr/share/mpss/boot/bzImage-kni...
root@lyra% ssh mic0 mount | grep home # Verify mount success
host:/home/ on /home type nfs (rw,relatime,vers=3,rsize=1048576,
wsize=1048576,namlen=255,hard,proto=tcp,port=65535,timeo=70,
retrans=3,sec=sys,local_lock=none,addr=172.31.1.254)
```

Listing 1.28: Automated procedure for creating a persistent NFS mount on a coprocessor.

For more information regarding the NFS service, refer to [11].

1.5.20. Sharing a Local Disk with VirtIO Block Device

An alternative procedure for sharing files between the host and coprocessors is the use of the VirtIO Block Device support (see Figure 1.14). VirtIO, as the name suggests, originates from platform virtualization tasks, where it is used to export data storage devices to virtual machines. This technology is similar to NFS in that it allows to read and write files in a directory on a coprocessor, while these files are stored on physical media (a hard disk drive) in the CPU-based host system. However, unlike NFS, VirtIO cannot be used to export a filesystem on a remote host. Also, with VirtIO, the storage device is not shared with the host, but completely given to the coprocessor; unmounting is required for the host to access new files on the virtual disk. On the positive side, as of MPSS 3.3, VirtIO can provide better file reading performance than `NFS`. Also, VirtIO can be used to mount a swap partition, which is not possible with NFS.

Host-side configuration of VirtIO is shown in Listing 1.29.

```
root@lyra% blk=/dev/mapper/vg_storage-lv_share
root@lyra% echo $blk >/sys/class/mic/mic0/virtblk_file
```

Listing 1.29: Host-side configuration for the export of an LVM volume to an Intel Xeon Phi coprocessor using VirtIO.

Here `/dev/mapper/vg_storage-lv_share` is an example filename, indicating an LVM volume that we are sharing with the coprocessor `mic0`. Besides an LVM volume, it is possible to share a regular file or a physical device.

Coprocessor-side configuration involves a procedure shown in Listing 1.30.

```
root@mic0% modprobe mic_virtblk # Load the kernel module on MIC
root@mic0% mkdir /mnt/vda # Create the mount point
root@mic0% mkfs.ext2 /dev/vda # Format the exported virtual drive
root@mic0% mount -t ext2 /dev/vda /mnt/vda # Mount
```

Listing 1.30: Coprocessor-side configuration for mounting the virtual drive.

Note that while it is possible to format the virtual drive on the host (by formatting the device listed in `virtblk_file`), formatting it on the coprocessor yields better performance.

Some of these configuration steps are persistent across coprocessor reboots. Namely, formatting the drive is necessary only once in the life of the filesystem. Also, the kernel module `mic_virtblk` will be automatically loaded after an MPSS reboot as long as `virtblk_file` on the host exists. However, the creation of the mount point and mounting the virtual drive (Listing 1.30) will not survive coprocessor reboot. Like with NFS, it is possible to automatically mount the device on the coprocessor. The procedure for doing so is shown in Listing 1.31.

```
root@lyra% mkdir -p /var/mpss/mic0/mnt/vda
root@lyra% echo '/dev/vda  /mnt/vda  ext2  defaults  0  0' >> \
>          /var/mpss/mic0/etc/fstab
root@lyra% service mpss restart
root@lyra% ssh mic0 mount | grep vda
/dev/vda on /mnt/vda type ext2 (rw,relatime,errors=continue)
```

Listing 1.31: Creating a persistent mount point and a mounting rule for the virtual drive on the coprocessor.

To access from the host the files written on the coprocessor, unmount the VirtIO device on the coprocessor and mount it on the host. Do the same to copy files from the host onto the virtual drive for subsequent reading on the coprocessor.

```
root@lyra% ssh mic0 umount /mnt/vda
root@lyra% mkdir -p /mnt/share
root@lyra% mount /dev/mapper/vg_storage-lv_share /mnt/share
root@lyra% ls /mnt/share/*
```

Listing 1.32: Unmounting the virtual drive on the host and mounting it on the coprocessor is required for host-side manipulations with the drive filesystem.

Naturally, `/dev/mapper/vg_storage-lv_share` is again an example of the device exported to the coprocessor; this name on your filesystem may be different.

1.5.21. Bridged Networking in Clusters with Coprocessors

As Section 1.2.5 explains, it is possible to configure the virtual network interfaces (`mic0`, `mic1`, etc.) and the coprocessor OS in such a way that the coprocessors join the private network of the hosts. This allows peer-to-peer communication between coprocessors in different chassis, including access to a remote NFS server and collective MPI communication.

We will assume that the private network of the cluster has a netmask `255.255.255.0`, and compute nodes have IP addresses `10.33.0.1`, `10.33.0.11`, `10.33.0.21`, etc. We will configure the coprocessor network in such a way that coprocessors in the first compute node have IP addresses `10.33.0.2` and `10.33.0.3`; in the second compute node — `10.33.0.12` and `10.33.0.13`, etc.

To set up this configuration, first, a network bridge must be created on each compute node. This is done by creating a NIC configuration file shown in Listing 1.33.

```
root@lyra% cat /etc/sysconfig/network-scripts/ifcfg-eno1
DEVICE=eno1
TYPE=Ethernet
NM_CONTROLLED=no
ONBOOT=yes
BRIDGE=br1
BOOTPROTO=none
HWADDR=00:1e:67:56:b5:a6 # System-specific value
root@lyra% cat /etc/sysconfig/network-scripts/ifcfg-br1
DEVICE=br1
TYPE=Bridge
ONBOOT=yes
DELAY=0
NM_CONTROLLED=no
BOOTPROTO=static
IPADDR=10.33.0.1 # IP on the private network
NETMASK=255.255.255.0 # Netmask of the private network
root@lyra% # In case of incorrect configuration, you may
root@lyra% # lose connection to the system at this step.
root@lyra% service network restart
```

Listing 1.33: Configuration files for a virtual bridge on a compute host.

In this configuration, `ifcfg-br1` is a new file that we created. With this file, we are configuring the host system to use the virtual interface `br1` to connect to the network, and self-assign the IP address `10.33.0.1`. If the cluster has a DHCP server, it is acceptable to connect `br1` using DHCP by setting `BOOTPROTO=dhcp`.

The file `ifcfg-eno1` was created during the OS installation, and we modified it by adding the line `BRIDGE=br1` and removing the lines that assign the IP address to this device. This procedure must be repeated on each compute node, either manually, or using the cluster management software.

The second step in creating bridged networking for Intel Xeon Phi coprocessors is shown in Listing 1.34.

```
root@lyra% service mpss stop
root@lyra% micctrl --addbridge=br1 --type=external \
>          --ip=10.33.0.1 --netbits=24
root@lyra% micctrl --network --bridge=br1 \
>          --ip=10.33.0.2:10.33.0.3
root@lyra% service mpss start
```

Listing 1.34: Configuring coprocessors on compute nodes to connect to an external network bridge. This makes coprocessors on remote machines IP-addressable.

The command `micctrl --network ...` has changed the IP addresses of the two coprocessors present in this system. This change will be reflected in `/etc/hosts`.

Now that the coprocessors of this machine have IP addresses on the same subnet as the hosts, it is possible to ping, SSH into them and send MPI messages to them from remote machines on this subnet.

Further details on network configuration with Intel Xeon Phi coprocessors can be found in Chapter 5 of [12].

1.5.22. Peer to Peer Communication between Coprocessors

By default (i.e., if OFED is not installed), coprocessors will use the TCP/IP protocol for communication with each other. As the communication goes through the host OS, in order for two or more coprocessors in a system to exchange TCP/IP packets, it is necessary to enable TCP/IP packet forwarding in the host OS or create an internal bridge with `micctrl`.

It is a good idea to enable packet forwarding or set up an internal bridge even if OFED is installed, because the user may force MPI applications to use TCP/IP instead of InfiniBand by setting `I_MPI_FABRICS=tcp` (see Section 2.4). This can be done for debugging or benchmarking purposes.

Packet Forwarding in Static Pair Topology

When using the static pair topology (default MPSS network configuration), in order to check whether IP packets can travel from one coprocessor to another, log in to the coprocessor and try to `ping` another coprocessor. If this test fails, packet forwarding is likely disabled on the host.

To enable packet forwarding, edit the file `/etc/sysctl.conf` and ensure that the following line is present (or change `0` to `1` in it):

```
net.ipv4.ip_forward = 1
```

Listing 1.35: Enabling packet forwarding in the host file `/etc/sysctl.conf` to allow peer to peer communication between coprocessors.

Edits of file `/etc/sysctl.conf` will become effective after a system reboot. To enable enable packet forwarding for the current session (i.e., without reboot), use the command shown in Listing 1.36.

```
vega@lyra% sudo /sbin/sysctl -w net.ipv4.ip_forward=1
```

Listing 1.36: Enabling packet forwarding on the host.

If a firewall is used, make sure that it is not blocking the forwarding of packets from coprocessors (see Listing 1.24).

Internal Bridge Topology

Instead of exposing the system to potentially harmful traffic, the administrator may enable peer to peer communication between coprocessors in a single system by setting up a network bridge. An example procedure for this is shown in Listing 1.37.

```
vega@lyra% sudo su
root@lyra% systemctl stop mpss
root@lyra% micctrl --addbridge=br1 --type=internal \
>                  --ip=192.168.100.1
root@lyra% micctrl --network=static --bridge=br1 \
>                  --ip=192.168.100.2:192.168.100.3
root@lyra% service mpss start
root@lyra% service mpss start
root@lyra% ifconfig br1 | head -2
br1       Link encap:Ethernet  HWaddr 4C:79:BA:26:08:F9
          inet addr:192.168.100.1  Bcast:192.168.100.255
                                    Mask:255.255.255.0
root@lyra% cat /etc/hosts | grep lyra
192.168.100.2        lyra-mic0 mic0 #Generated-by-micctrl
192.168.100.3        lyra-mic1 mic1 #Generated-by-micctrl
```

Listing 1.37: Setting up an internal bridge.

In the above procedure, a private subnet `192.168.100.255/24` is created within the host. The host and both coprocessors can communicate with each other in this subnet.

1.5.23. Manual Customization of the coprocessor OS

The tool `micctrl` has a large number of functions to modify the default configuration of MPSS. However, for fine-grained control, or just for reliability reasons, some administrators may prefer to manually manipulate configuration files instead of using `micctrl`.

The configuration files controlling MPSS are:

1. Static configuration files in `/etc/mpss`. These include:

   - Global configuration file `/etc/mpss/default.conf` containing the location of the MIC filesystem image, network configuration and crash/timeout options;
   - Per-coprocessor configuration files `/etc/mpss/mic*.conf` containing boot and network options for Intel Xeon Phi coprocessors;
   - Additional configuration files `/etc/mpss/conf.d/*` containing overlays for additional packages on coprocessors.

2. OS file trees and images in `/var/mpss`. These include:

   - Per-coprocessor filelists `/var/mpss/mic*.filelist` and trees `/var/mpss/mic*`
   - Per-coprocessor filesystem images `/var/mpss/mic*.image.gz`, which are copied to coprocessors and used during the MPSS boot process.

The relationship between static configuration files and OS file trees and images is that the former are used to generate the latter. If the administrator changes anything in `/etc/mpss`, these changes must be propagated to `/var/mpss` by running `micctrl --resetconfig`.

Any changes in `/var/mpss` (either performed manually, or using the `micctrl` tool) are reflected in the image files automatically and become effective upon coprocessor reboot.

CHAPTER 2
Programming Models

In Chapter 1, we introduced the MIC architecture without going into the details of how to program Intel Xeon Phi coprocessors. This chapter demonstrates the utilization of the Intel Xeon Phi coprocessor from user applications written in C or C++. It focuses on transferring data and executable code to the coprocessor. Parallelism will be discussed in Chapter 3, and performance optimization in Chapter 4.

There are two classes of programming models for Intel Xeon Phi coprocessors: offload and native models. Offload applications are those in which the process is launched on the host CPU and later communicates with the coprocessor on the local machine (Figure 2.1, left). Native applications, in contrast, start on the coprocessor directly (Figure 2.1).

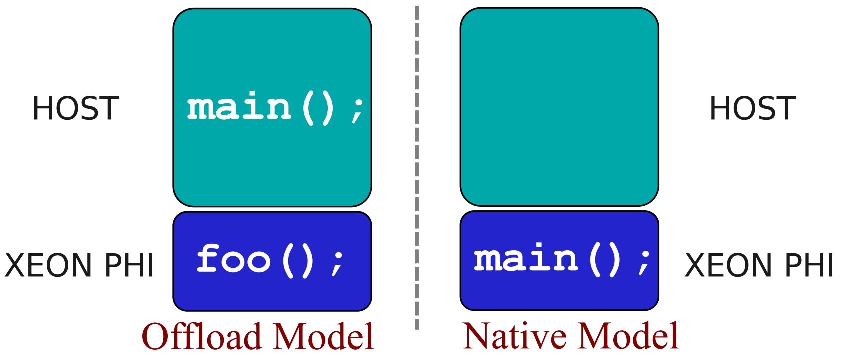


Figure 2.1: Classes of programming models for Intel Xeon Phi coprocessors.

Within each of these two classes, applications differ in the way they share resources and scale across a cluster, and in programming language extensions used to perform offload. Section 2.1 discusses native applications, and Sections 2.2, 2.3 and 2.5 introduce three different offload approaches. Scaling across heterogeneous clusters is discussed in Section 2.4.

2.1. Native Applications and MPI

Intel Xeon Phi coprocessors run a Linux operating system, with a virtual filesystem, a multi-user environment, and support for traditional Linux services, including SSH and NFS. These services allow the programmer to run applications directly on an Intel Xeon Phi coprocessor, without the involvement of the host. This does not mean that an application compiled for an Intel Xeon CPU will run on an Intel Xeon Phi coprocessor. Rather, it means that it is possible to compile an application for an Intel Xeon Phi coprocessor from the same source code as the CPU application. Then the executable and its dependent libraries must be transferred to, or shared with, the coprocessor's filesystem. This approach to utilizing coprocessors is called native programming.

2.1.1. Using Compiler Argument `-mmic` to Compile Native Applications for Intel® Xeon Phi™ Coprocessors

To compile a C, C++ or Fortran code as an executable for the Intel Xeon Phi architecture, Intel compilers must be invoked with the argument `-mmic`. A "Hello World" code for the coprocessor is shown in Listing 2.1.

Note that for consistency, throughout this book we treat all code as C++, however, a lot of of our examples do not use object-oriented programming and can be compiled, with minimal modifications, as C.

```
#include <cstdio>
#include <unistd.h>
int main(){
  printf("Hello world! I have %ld logical processors.\n",
         sysconf(_SC_NPROCESSORS_ONLN ));
}
```

Listing 2.1: This C++ language code ("`Native-Hello.cc`") can be compiled for execution on the host as well as on an Intel Xeon Phi coprocessor.

First, let us run this code on the host CPU. The compilation procedure and runtime output are shown in Listing 2.2. Here, `icpc` is the executable for the Intel C++ Compiler. The name of the executable is not specified, so the compiler sets it to the default name `a.out`.

```
vega@lyra% icpc Native-Hello.cc # Compile file for the CPU
vega@lyra% ./a.out
Hello world! I have 48 logical processors.
```

Listing 2.2: Compiling and running the “Hello World” code on the host.

In a similar way, the same code can be compiled to run on the Intel Xeon Phi architecture. Listing 2.3 demonstrates that. In this case, an additional argument `-mmic` is passed to the compiler. Note that the code fails to run on the host, because it is not compiled for the Intel Xeon architecture.

```
vega@lyra% icpc Native-Hello.cc -mmic # Compile for the MIC
vega@lyra% ./a.out
-bash: ./a.out: cannot execute binary file
```

Listing 2.3: Compiling native application for Intel Xeon Phi coprocessors.

The next step is to run the native application on an Intel Xeon Phi coprocessor. We will demonstrate this in the next section, after discussing the options for establishing secure shell sessions with coprocessors.

2.1.2. Running Native Applications on Using SSH

Intel Xeon Phi coprocessors run a Linux operating system with an SSH server (see also Section 1.5.3). When the MPSS is configured, the list of Linux users and their SSH keys on the host are transferred to the Intel Xeon Phi coprocessor filesystem. By default, the first Intel Xeon Phi coprocessor in the system is resolved to the hostname `mic0`, as specified in the file `/etc/hosts`.

That said, we can transfer the executable `a.out` to the coprocessor using the secure copy tool `scp`, as shown in Listing 2.4. After that, we can log into the coprocessor using `ssh` and use the shell to run the application on the coprocessor. Running this executable produces the expected "Hello world" output, and the number of logical processors is correctly detected as 244.

```
vega@lyra% scp a.out mic0:~/
a.out                     100%    10KB  10.4KB/s   00:00
vega@lyra% ssh mic0
user@mic0% pwd
/home/user
user@mic0% ls
a.out
user@mic0% ./a.out
Hello world! I have 244 logical processors.
```

Listing 2.4: Transferring and running a native application on an Intel Xeon Phi coprocessor.

An alternative way to transfer the compiled application to the coprocessor is to put the executable in a directory shared with the coprocessor. The common protocol NFS can be used to share a directory with the coprocessor (see Section 1.5.19).

Note that the application must still be compiled on the host. It is not possible to log in to `mic0` and compile the code from the coprocessor, because the Intel C++ Compiler executable itself is compiled for the CPU architecture.

2.1.3. Running Native Applications with `micnativeloadex`

A part of the MPSS tool suite, the `micnativeloadex` utility, is a tool for running native applications on Intel Xeon Phi coprocessors. It uses the SSH protocol to copy a native binary to a specified Intel Xeon Phi coprocessor and execute it. In addition, `micnativeloadex` automatically checks library dependencies for the application, and, if they can be located, these libraries are also copied to the device prior to execution. By default, the output from the application running remotely on the Intel Xeon Phi coprocessor is redirected back to the local host console. This redirection can be enabled or disabled using the environment variable `MIC_PROXY_IO` (see Section 2.2.10).

The default search for path dependent libraries is set using the environment variable `SINK_LD_LIBRARY_PATH`. This environment variable works just like the variable `LD_LIBRARY_PATH` for normal Linux applications. To only display the list of dependencies, `micnativeloadex` should be run with the command line argument `-l`.

To demonstrate how this tool works, consider a native application that uses the Intel OpenMP library (Listing 2.5).

```
#include <omp.h>
#include <cstdio>
int main(){
  printf("Hello world! I have %ld logical processors.\n",
         omp_get_max_threads()); // function from the OpenMP library
}
```

Listing 2.5: Sample application `Native-Hello2.cc` with a dependency on the OpenMP library.

```
vega@lyra% icpc -mmic -qopenmp -o Native-Hello2 Native-Hello2.cc
vega@lyra% micnativeloadex ./Native-Hello2
The remote process indicated that the following libraries could not
be loaded: libiomp5.so
...
```

Listing 2.6: Using `micnativeloadex` to run a native application on a coprocessor.

This time, we compile this application with the argument `-qopenmp` in order to enable dynamic linking of the Intel OpenMP library[1]. We also specify the name of the executable file by supplying the argument `-o`. Listing 2.6 shows the compilation procedure and the result of this operation. Here, `micnativeloadex` reports that it was unable to locate the library file `libiomp5.so` for the coprocessor architecture. We must add the file location to `SINK_LD_LIBRARY_PATH`, as shown in Listing 2.7.

```
vega@lyra% locate libiomp5.so
/opt/intel/composer_xe_.../compiler/lib/ia32/libiomp5.so
/opt/intel/composer_xe.../compiler/lib/intel64/libiomp5.so
/opt/intel/composer_xe.../compiler/lib/mic/libiomp5.so
vega@lyra% SINK_LD_LIBRARY_PATH=\
>/opt/intel/composer_xe.../compiler/lib/mic
vega@lyra% micnativeloadex ./Native-Hello2
Hello world! I have 240 logical processors.
```

Listing 2.7: Using `SINK_LD_LIBRARY_PATH` to help `micnativeloadex` to run a native application on an Intel Xeon Phi coprocessor.

We used the Linux tool `locate` to find the file `libiomp5.so`. This file was found in the file tree of the Intel Composer XE tool. The directory `compiler/lib` contains subdirectories `ia32`, `intel64` and `mic` for 32-bit and 64-bit Intel architecture and for the Intel MIC architecture, respectively. We included the MIC path into `SINK_LD_LIBRARY_PATH`, which enabled `micnativeloadex` to locate the corresponding file to the coprocessor. The application was executed successfully.

Note that if we were using the SSH approach of Section 2.1.2, then we would have to manually locate and transfer `libiomp5.so` to the coprocessor. An alternative to `micnativeloadex` and manual copying of libraries is NFS-sharing directories containing Intel libraries and user libraries for the MIC architecture as discussed in Section 1.5.19.

[1]Older versions of Intel compilers used `-openmp`. Starting with Intel compilers version 15.0, that argument was deprecated along with all other arguments begining with `-o`. The argument `-o` is now reserved for specifying the output file name. The new correct spelling of the OpenMP linkage argument is `-qopenmp`

2.1.4. Monitoring the Coprocessor Activity with `micsmc`

To demonstrate how the Intel Xeon Phi coprocessor activity can be monitored, and at the same time to show how POSIX threads (Pthreads) can be used to execute parallel codes on coprocessors, below we construct a Pthreads-based workload. The source code shown in Listing 2.8 spawns as many threads as there are logical processors in the system, and each thread executes an infinite loop in function `Spin(void*)`. This application does not perform any useful calculations, but it keeps all cores occupied to produce activity that we will monitor. Naturally, this code is suitable for both the host (an Intel Xeon processor) and the target (an Intel Xeon Phi coprocessor).

```
#include <cstdio>
#include <unistd.h>
#include <pthread.h>

void *Spin(void *arg) {
  while(1); // Go into an infinite loop
  pthread_exit(NULL);
}

int main (int argc, char *argv[]){
  const int n = sysconf(_SC_NPROCESSORS_ONLN);
  printf("Spawning %d threads that do nothing. \
Use ^C to terminate.\n", n);
  fflush(0); // Flush stdout for proxy console I/O
  for (int i = 1; i < n; i++) {
    pthread_t thr; // Create (n-1) threads
    pthread_create(&thr, NULL, Spin, NULL);
  }
  Spin(NULL); // Hang the main thread
  pthread_exit(NULL);
}
```

Listing 2.8: This C code (“`Native-Spin.cc`”) illustrates how the Pthreads library can be used to produce parallel applications on Intel Xeon Phi coprocessors.

The output in Listing 2.9 illustrates the compilation and running of the code `Native-Spin.cc` on a coprocessor. The code enters an infinite loop and never terminates, so the execution must be terminated by pressing

`Ctrl+C`. However, while the program is running, we can monitor the Intel Xeon Phi coprocessor load.

```
vega@lyra% icpc -lpthread -o Native-Spin -mmic Native-Spin.cc
vega@lyra% micnativeloadex ./Native-Spin
Spawning 240 threads that do nothing. Use ^C to terminate.
^C
```

Listing 2.9: Running `Native-Spin.cc` as a native workload for Intel Xeon Phi coprocessors.

Prior to running `spin`, in a separate terminal, we can launch the monitoring utility by executing "`micsmc`". This action initiates the graphical graphical user interface for monitoring the load on the coprocessor, temperature, reading logs and error messages, and controlling some of the coprocessor's settings.

Figure 2.2 illustrates how the load on the coprocessor increases for the duration of the execution of the workload code, and drops afterwards. Memory usage, power consumption and processor temperature are also displayed.

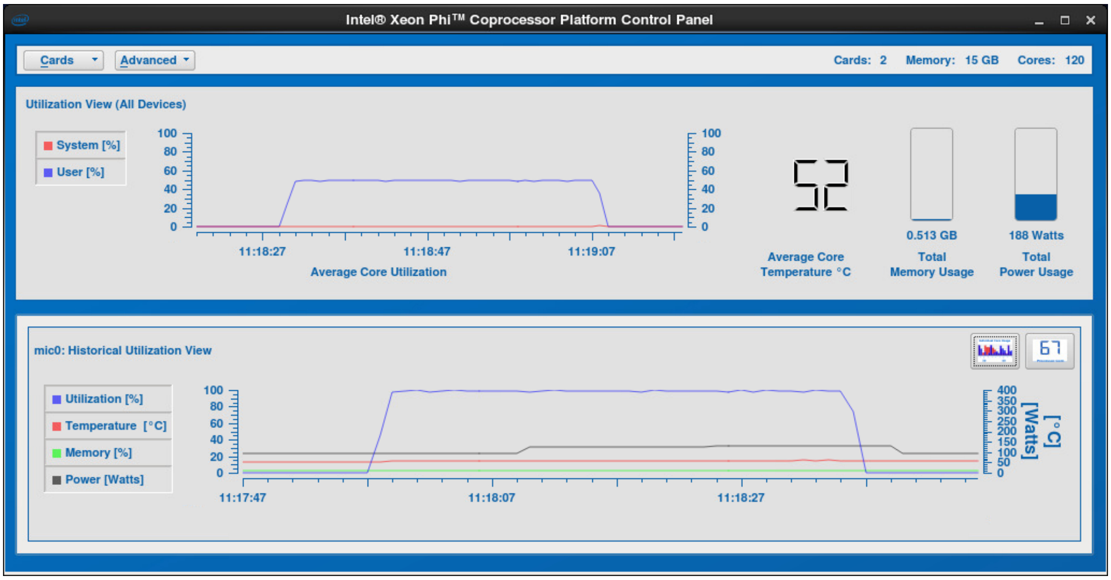
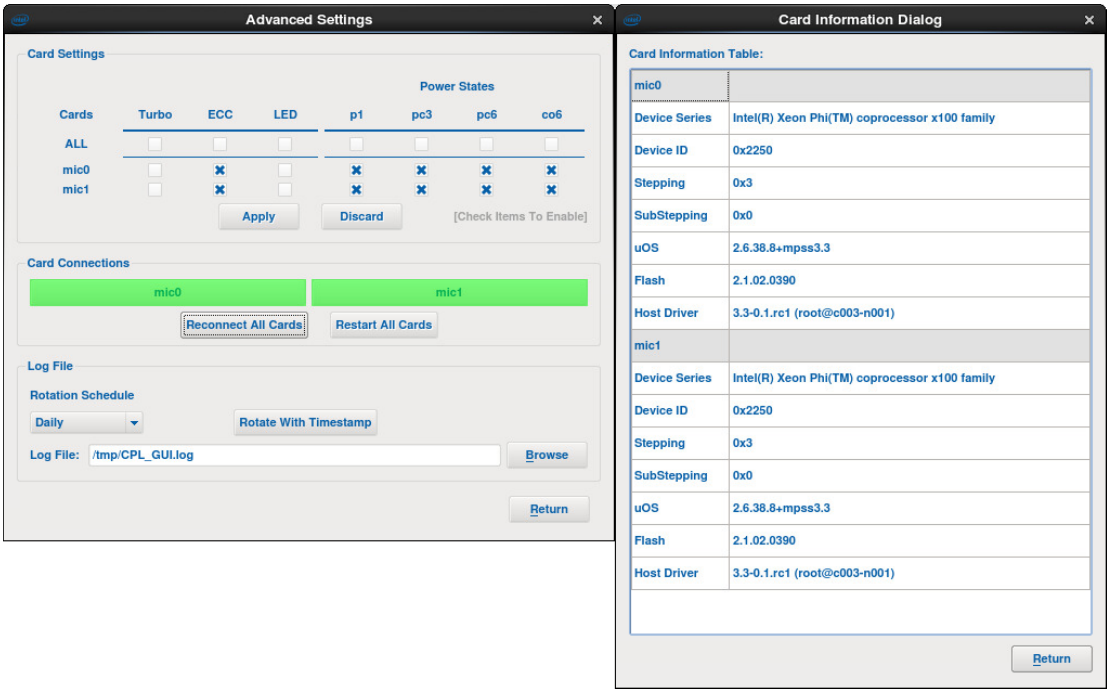


Figure 2.2: The interface of the `micsmc` utility: the load on the Intel Xeon Phi coprocessor.

Figure 2.3 shows additional functionality of `micsmc`: power settings control and information panel. Much of this information and controls are also available via the MPSS command line tools.



Card Settings

				Power States			
Cards	Turbo	ECC	LED	p1	pc3	pc6	co6
ALL	☐	☐	☐	☐	☐	☐	☐
mic0	☐	✖	☐	✖	✖	✖	✖
mic1	☐	✖	☐	✖	✖	✖	✖

Card Connections

Log File

Rotation Schedule: Daily

Log File: /tmp/CPL_GUI.log

Card Information Table:

mic0	
Device Series	Intel(R) Xeon Phi(TM) coprocessor x100 family
Device ID	0x2250
Stepping	0x3
SubStepping	0x0
uOS	2.6.38.8+mpss3.3
Flash	2.1.02.0390
Host Driver	3.3-0.1.rc1 (root@c003-n001)
mic1	
Device Series	Intel(R) Xeon Phi(TM) coprocessor x100 family
Device ID	0x2250
Stepping	0x3
SubStepping	0x0
uOS	2.6.38.8+mpss3.3
Flash	2.1.02.0390
Host Driver	3.3-0.1.rc1 (root@c003-n001)

Figure 2.3: The interface of the `micsmc` utility. Power control and information.

See Section 1.5.10 for more information on `micsmc`.

2.1.5. MPI Applications on Intel Xeon Phi Coprocessors

The ability of Intel Xeon Phi coprocessors to run native applications and the IP-addressability feature of coprocessors make it possible to seamlessly integrate Intel Xeon Phi coprocessors into applications that employ the traditional solution for cluster computing, MPI. This section describes how to compile a native application for Intel Xeon Phi coprocessors with MPI and run it on the coprocessor.

About the MPI Protocol

MPI, or Message Passing Interface, is a high-level communication protocol for high performance computing applications in distributed memory systems. This protocol provides an API for exchanging data (messages) between processes participating in a collective job. All aspects of communication on the hardware and OS level are handled by the implementation of MPI, allowing the programmer to focus on the parallel algorithm and data traffic patterns.

For information about using MPI to express parallel algorithms in distributed memory, refer to Section 3.4.

Intel MPI

Intel's proprietary implementation of MPI is available as the Intel MPI Library. Intel MPI version 5 implements version 3.0 of the MPI protocol.

The Intel MPI Reference Guide [13] contains more detailed information about using Intel MPI.

Setting Up

Intel MPI library is available as a stand-alone Intel software product, or as a part of the Intel Parallel Studio XE. In this section, we assume that Intel MPI is used on a single compute node (or workstation) with one or more Intel Xeon Phi coprocessor.

After installing MPI, environment variables should be set by calling a script included in the Intel MPI distribution (Listing 2.10):

```
vega@lyra% source /opt/intel/impi/5.0.1.035/intel64/bin/mpivars.sh
```

Listing 2.10: Setting Intel MPI environment variables on the host. The location of the initialization script changes with Intel MPI library version.

The above procedure enables the running of MPI processes on the CPU, but not on coprocessors. For native coprocessor applications with MPI, the Intel MPI binaries and libraries need to be made available to the Intel Xeon Phi coprocessor. There are two ways to achieve this:

1) A straightforward, but not recommended method, is to copy certain files from `/opt/intel/impi` to the coprocessor.

2) A better method is to NFS-share the required files with the coprocessor or coprocessors. The administrator may share `/opt/intel/impi` or even `/opt/intel` with all coprocessors in the system. The procedure for NFS-sharing directories with a coprocessor is described in Section 1.5.19.

We will assume that the latter method is used, and that all the required files are available to the coprocessor.

Usage

MPI applications must be compiled with special wrapper applications: `mpiicc` for C, `mpiicpc` for C++ or `mpiifort` for Fortran codes. To launch the resulting executable as a parallel MPI application it should be run using a wrapper script called `mpirun`. MPI executables can also be executed as usual applications, but parallelization does not occur in this case.

“Hello World” with MPI on the Host

Listing 3.64 shows a “Hello World” example of MPI usage. When this application is run in the MPI environment, multiple processes execute this code. Each of these processes is assigned an identification number, known as rank, which can be used by the programmer to determine the role of each process in the application. MPI processes exchange information by passing messages in a variety of ways. Message passing is discussed in more detail in Section 3.4.

```
#include <mpi.h>

int main (int argc, char *argv[]) {
  int rank, size, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name (name, &namelen);
  printf ("Hello World from rank %d running on %s!\n", rank, name);
  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) printf("MPI World size = %d processes\n", size);
  MPI_Finalize ();
}
```

Listing 2.11: Source code `MPI-Hello.cc` of a “Hello world” program with MPI.

To compile and run the source file from Listing 2.11, we use the procedure demonstrated in Listing 2.12; `node1` is the local hostname.

```
user@node1% mpiicpc -o MPI-Hello MPI-Hello.cc # MPI compiler
user@node1% mpirun -host node1 -np 2 ./MPI-Hello
Hello World from rank 1 running on node1!
Hello World from rank 0 running on node1!
MPI World size = 2 processes
```

Listing 2.12: Compiling the “Hello World!” code with Intel MPI for the host system and running it using two processes.

Hello World with MPI on the Coprocessor

To compile the "Hello World" code for MPI on Intel Xeon Phi coprocessors, the `-mmic` compiler flag must be used, just like with non-MPI native applications. No modification of the source code is required to use MPI on coprocessors in this case. To run the application, the resulting executable file must be copied to (or NFS-shared with) the Intel Xeon Phi coprocessor, and then `mpirun` must be invoked on the host as shown in Listing 2.13.

```
user@node1% mpiicpc -mmic -o MPI-Hello.MIC MPI-Hello.cc
user@node1% sudo scp MPI-Hello.MIC mic0:~/
user@node1% export I_MPI_MIC=1
user@node1% mpirun -host mic0 -np 2 ~/MPI-Hello.MIC
Hello World from rank 1 running on node1-mic0!
Hello World from rank 0 running on node1-mic0!
MPI World size = 2 processes
```

Listing 2.13: Compiling and running a Hello World code with Intel MPI on an Intel Xeon Phi coprocessor.

The difference between this case and the case shown in Listing 2.12 is that we included the argument `-host mic0` instead of `-host node1`. According to `/etc/hosts`, the hostnames `mic0` and `node1-mic0` resolve to the first coprocessor in the system. This assignment is made by the MPSS initialization script. In addition, we had to set the environment variable `I_MPI_MIC=1` in order to enable Intel MPI processes on the MIC architecture.

Even though the code `MPI-Hello.cc` does not contain any explicit message passing, it contains a call to `MPI_Barrier()`, which involves some implicit communication. This function serves to ensure that the last line of output ("MPI World size=...) is always printed after all introductions from MPI ranks.

Additional Information

This concludes a brief introduction into using Intel MPI to compile and run MPI applications on Intel Xeon Phi coprocessors. The discussion of MPI will continue later in this book:

- In Section 2.4.3 we will demonstrate how to run MPI calculations on multiple coprocessors or on the host and coprocessors simultaneously, and how to execute applications on a cluster with a longer list of nodes;
- Subsequently, in Section 3.4, we will introduce message passing API, which allows to effect cooperation between processes.
- Optimization in MPI is discussed in Section 4.7.

2.2. Explicit Offload Model

Section 2.1 demonstrated how native codes for the MIC architecture may be run directly on the coprocessor without the involvement of the host. This is not the only way to use Intel Xeon Phi coprocessors. An alternative method is the so-called "offload approach", where an application begins execution on the host, and at some point it employs the MIC architecture by transferring only some of the data and functions to run on the coprocessor. This process of data and code transfer to the coprocessor is generally called *offload*, and applications using this procedure are known as offload applications. This section describes a set of C/C++ language extensions for the explicit offload model, and Section 2.3 introduces alternative extensions, the virtual-shared memory model. Finally, Section 2.5 discusses the features of the OpenMP 4.0 specification designed specifically for offload applications.

2.2.1. "Hello World" Example in the Explicit Offload Model

The source code in the C++ language in Listing 2.14 demonstrates offloading a section of the host application to an Intel Xeon Phi coprocessor using the statement `#pragma offload`.

```
1  #include <cstdio>
2  #include <unistd.h>
3  
4  int main(int argc, char * argv[] ) {
5   printf("Hello World from main()! I see %d logical processors.\n",
6           sysconf(_SC_NPROCESSORS_ONLN ));
7  #pragma offload target(mic)
8  {
9   printf("Hello World from offload! I see %d logical processors.\n",
10          sysconf(_SC_NPROCESSORS_ONLN ));
11 }
12  printf("Bye\n");
13 }
```

Listing 2.14: Source code of `Offload-Hello.cc` example with an offload segment to be executed on an Intel Xeon Phi coprocessor.

Line 6 in Listing 2.14 — `#pragma offload target(mic)` — indicates that the following segment of the code should be executed on an Intel Xeon Phi coprocessor (i.e., "offloaded").

This application must be compiled as a usual host application: no additional compiler arguments are necessary in order to compile offload applications. This code produces the following output:

```
vega@lyra% icpc -o Offload-Hello Offload-Hello.cc
vega@lyra% ./Offload-Hello
Hello World from main()! I see 48 logical processors.
Bye
Hello World from offload! I see 244 logical processors.
```

Listing 2.15: Output of the execution of `Offload-Hello.cc`.

In the default version implemented in `Offload-Hello.cc`, offload is blocking, i.e., control returns to `main()` only after the offloaded code returns.

In this context, it is worth mentioning that it is somewhat surprising that the line "Bye" is the second printed line at runtime, even though in the original code it is the third line. At the same time, it should be surprising to the beginner reader that "Hello from offload" was printed at all, because it was output into `stdout` of the OS. The explanation for the presence of this line, as well as for the altered order of output, is the proxy console I/O functionality discussed in Section 2.2.10. The output to the coprocessor's `stdout` is buffered and mirrored in the host console, and the consistency of the order of output is therefore not guaranteed.

In this example, initiating offload with `#pragma offload` is trivial, because all code and all data in the offload region exist only in the scope of `#pragma offload`. However, when functions or data need to be offloaded, offload programming will require more customization of the offload directive.

2.2.2. Offloading Functions

When user-defined functions are called in an offload region, they must be declared with the qualifier `__attribute__((target(mic)))` (see Listing 2.16). This qualifier tells the compiler to generate the MIC architecture executable code for the function.

```
__attribute__((target(mic))) void MyFunction() {
    printf("Hello from offload!\n");
}

// ...
#pragma offload target(mic)
  {
    MyFunction();
  }
```

Listing 2.16: Offloading a function to an Intel Xeon Phi coprocessor.

If multiple functions must be declared with this qualifier, there is a shorthand way to set and unset this qualifier inside a source file (see Listing 2.17). This also useful when using `#include` to inline header files.

```
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() { // This function has target(mic) set
        printf("Hello World from coprocessor!\n");
}
void MyFunctionTwo() { // The target(mic) attribute is still active
        fflush(0);
}
#pragma offload_attribute(pop)

//...
#pragma offload target(mic)
{
  MyFunctionOne();
  MyFunctionTwo();
}
```

Listing 2.17: Declaring multiple functions with the target attribute qualifier.

2.2.3. Offloading Bitwise-Copyable Data

In this section, we will discuss the transfer of data to the coprocessor in the explicit offload model.

Offloading Scope-Local Data

Local scalar variables and arrays of known size are automatically transferred to and from the coprocessor at the start and the end of the offload, respectively. However this behavior can be modified by including them into one of the clauses of the offload pragma: `in`, `out`, `inout`, or `nocopy`. In the code in Listing 2.18, the value of `N` will be copied "in", i.e., from host to the coprocessor at the start of the offload. After offload, the contents of `data` will be copied "out", i.e., from the coprocessor back to the host.

```
void MyFunction() {
  int N = 1000; // Local scalar
  int data[N]; // Local array of known size
#pragma offload target(mic) in(N) out(data)
  {
    for (int i = 0; i < N; i++)
      data[i] = i;
  }
}
```

Listing 2.18: Offload of local scalars and arrays of known size using `#pragma offload`.

When data is stored in an array referenced by a pointer, the array size is unknown at compile time. In this case, the programmer must indicate the array length in a clause of `#pragma offload`, as shown in Listing 2.19. The length is indicated in array elements and not bytes.

```
void MyFunction(const int N, int* data) {
#pragma offload target(mic) in(N) out(data: length(N))
  {
    for (int i = 0; i < N; i++)
      data[i] = 0;
  }
}
```

Listing 2.19: Offload of pointer-based arrays of unknown size

Offloading Global and Static Variables

When an offloaded variable is used in the global scope or with the `static` attribute, it must be declared with the same qualifier as an offloadable function, `__attribute__((target(mic)))`:

```
int* __attribute__((target(mic))) data;

void MyFunction() {
  static __attribute__((target(mic))) int N = 1000;
  data = new int[N];
#pragma offload target(mic) in(N) out(data: length(N))
  {
    for (int i = 0; i < N; i++)
      data[i] = 0;
  }
}
```

Listing 2.20: Offload of global and static variables

Data Transfer without Computation

If it is necessary to send data to the coprocessor without launching any processing of this data, either the body of the offloaded code can be left blank (i.e., use "`{}`" after `#pragma offload`), or a special `#pragma offload_transfer` can be used as shown in Listing 2.21.

```
#pragma offload_transfer target(mic) in(N) out(data: length(N))
  // The above pragma does not have a body.
  // Continuing on the host...
```

Listing 2.21: Transferring data to the coprocessor without computation.

This pragma is especially useful when combined with the clause `signal`. This initiates an asynchronous data transfer, which can be used to overlap communication with computation (on the host or on the coprocessor). See Section 2.2.5 for a discussion of asynchronous data transfer.

2.2.4. Data and Memory Persistence Between Offloads

By default, when an array is offloaded to the coprocessor, the following operations are be performed:

(1) Allocate a memory buffer for the array on the coprocessor;
(2) Copy data from host array to the buffer on the coprocessor (for clauses `in` and `inout`);
(3) Perform offloaded calculations;
(4) Copy data from coprocessor buffer to the host array (for clauses `inout` and `out`);
(5) Free the memory buffer on the coprocessor.

In some cases, an offload region is called multiple times with the same shape and size of some or all data structures. In this case, step (1) is necessary only in the first offload instance, and step (5) – only in the last one. To preserve a memory buffer allocated on a coprocessor, clauses `alloc_if` and `free_if` may be used. These clauses are given arguments which, if evaluated to `1`, enforce the allocation or freeing of memory, respectively. Listing 2.22 illustrates the usage of these clauses. Note how the character ‘`\`’ is used in order to make the specification of the pragma continue onto the next line.

```
  double *p=(double*)malloc(sizeof(double)*N);

// Allocate, but not free memory for array p
#pragma offload target(mic) in(N) \
inout(p : length(N) alloc_if(1) free_if(0))
  {
    // ... perform work
  }

// Do not allocate, but free memory for array p
#pragma offload target(mic) in(N) \
inout(p : length(N) alloc_if(0) free_if(1))
  {
    // ... perform work
  }
```

Listing 2.22: Illustration of memory buffer retention on coprocessor between offloads.

Furthermore, for some data structures, the programmer may wish to skip steps (2) or (4) or both.

- To skip the copy-in stage (2), but perform copy-out (4), use the `out` clause.
- To skip the copy-out stage (4), but perform copy-in (2), use `in`.
- To skip both the copy-in and copy-out, use either the `nocopy` clause, or `in` with a length of 0.

The latter data persistence instruction must always be combined with the memory-retaining clause `free_if(0)` in the previous offload and `alloc_if(0)` in the current offload. Listing 2.23 illustrates data persistence between offloads.

```
// Allocate, but not free memory for array p
#pragma offload target(mic) in(N) in(p : length(N) free_if(0))
  {
    // ... perform work
  }

// Do not allocate memory for p, and re-use the data in it
#pragma offload target(mic) in(N) nocopy(p : length(N) alloc_if(0))
  {
    // ... perform work - same values in p as in first offload
  }
```

Listing 2.23: Illustration data persistence on coprocessor between offloads.

Memory buffer retention and data persistence may be crucial in applications where data traffic takes a significant fraction of execution time. See Section 2.2.4 for more information.

2.2.5. Asynchronous Offload

By default, offload pragmas are synchronous, i.e., they block execution on the host until the offloaded function returns. It is also possible to initiate asynchronous (i.e., non-blocking) offload and data transfer in the explicit offload model. Asynchronous data transfer opens additional possibilities for optimization:

a) data transfer time can be masked;
b) the host processor and coprocessor can be employed simultaneously;
c) it provides a way to distribute work across multiple coprocessors.

Asynchronous data transfer is initiated by adding the specifier `signal` to the offload pragma. After that, another offload pragma with the `wait` clause, or `#pragma offload_wait` are used to catch the signal of the end of the offload. Example in Listing 2.24 illustrates the use of asynchronous transfer pragmas.

```
#pragma offload_transfer target(mic:0) signal(data) \
                         in(N) in(data: length(N))

// Execution will not block until transfer is completer.
// The function below will be run concurrently with data transfer.
SomeOtherFunction(otherData);

#pragma offload target(mic:0) wait(data) \
                in(N) nocopy(data: length(N)) out(result: length(N))
{
  //...this offload will be launched after the data is transferred
}
```

Listing 2.24: Illustration of asynchronous data transfer and `wait` clause.

In this code, `#pragma offload_transfer` initiates the transfer, and specifier `signal` indicates that it should be asynchronous. With asynchronous offload, `SomeOtherFunction()` will be executed concurrently with data transport. In the second pragma statement, the specifier `wait(data)` indicates that the offloaded calculation should not start until the data transport signaled by `data` has been completed. Any pointer variable can serve as the signal, not just the pointer to the array being transferred.

Besides including the `wait` clause in an offload pragma, the compiler supports the `offload_wait` pragma, which is illustrated in Listing 2.25.

```
#pragma offload_wait target(mic:0) wait(data)
```

Listing 2.25: Illustration of `#pragma offload_wait`.

Here, the host code execution will wait at this pragma until the transport signaled by `data` has finished. This pragma is useful when it is not necessary to initiate another offload or data transfer at the synchronization point.

Similarly to asynchronous data transfer, function offload can be done asynchronously, as shown in Figure 2.26.

```
char* offload0;
char* offload1;

#pragma offload target(mic:0) signal(offload0) \
    in(N) in(data0 : length(N)) out(result0 : length(N))
{ // Offload will not begin until data is transferred
  Calculate(data0, result0);
}

#pragma offload target(mic:1) signal(offload1) \
    in(N) in(data1 : length(N)) out(result1 : length(N))
{ // Offload will not begin until data is transferred
  Calculate(data1, result1);
}

#pragma offload_wait target(mic:0) wait(offload0)
#pragma offload_wait target(mic:1) wait(offload1)
```

Listing 2.26: Illustration of asynchronous offload to different coprocessors.

In this code, two coprocessors are employed simultaneously using asynchronous offloads. More information on managing multiple coprocessors in a system with the explicit offload model can be found in Section 2.4.1.

Complete information about asynchronous transfer can be found in the Intel C++ Compiler reference.

2.2.6. Target-Specific Code

When the Intel compiler is building executable code for the MIC architecture, it defines the preprocessor macro `__MIC__`. For CPU code, this macro is undefined. That applies to the native applications (with and without the argument `-mmic`), as well as to offload codes (for the coprocessor and the host versions of functions marked with the offload attribute). This macro allows the programmer to "check" where the code is executed, and to write different versions of code for the host and for the coprocessor.

The macro `__MIC__` can be used to write multi-versioned codes in a number of practical cases:

a) The values of tuning parameters in HPC algorithms may depend on the amount of memory, cache size, and other parameters, which are different between the host and the coprocessor platforms. These tuning parameters can be multi-versioned using `__MIC__`:

```
#ifdef __MIC__
const int tileSize = 32; // Use the value 32 for MIC
#else
const int tileSize = 64; // Use the value 64 for CPU
#endif
```

b) When using intrinsics or assembly programming, `__MIC__` can protect functions unavailable on either the host, or the target platform.

```
#ifdef __MIC__
for (int i = 0; i < n; i += 16) { // Intrinsics on MIC
  __m512 Avec = _mm512_load_ps(A+i);
  __m512 Bvec = _mm512_load_ps(B+i);
  Avec = _mm512_add_ps(Avec, Bvec);
  _mm512_store_ps(A+i, Avec);
}
#else
for (int i = 0; i < n; i++) { // Same code with automatic
  A[i] = A[i] + B[i];         // vectorization on the CPU
}
#endif
```

c) It is often convenient to have diagnostic output reflect what platform the code is running on.

2.2.7. Optional and Conditional Offload, Fall-Back to Host

If no coprocessors are found in the system, the offload code can be executed anyway, using the host processor instead of the coprocessor. This can be achieved by adding the clause `optional` to the offload pragma.

```
#include <cstdio>
#include <unistd.h>
int main(int argc, char * argv[] ) {
  printf("Hello World from main()! I see %d logical processors.\n",
          sysconf(_SC_NPROCESSORS_ONLN ));
#pragma offload target(mic) optional
{
#ifdef __MIC__
  printf("Hello from offload on MIC with %d logical processors.\n",
#else
  printf("Hello from offload on CPU with %d logical processors.\n",
#endif
         sysconf(_SC_NPROCESSORS_ONLN )); fflush(0);
}}
```

Listing 2.27: `Offload-Fallback.cc`: handling fall-back to host when offload fails.

In Listing 2.28, the code `Offload-Fallback.cc` is complied and executed. In the first execution attempt, the coprocessor is available, and offload occurs. In the second attempt, the MIC driver is intentionally disabled, and offload fails. Execution proceeds nevertheless, only the code is run on the host.

```
vega@lyra% icpc Offload-Fallback.cc -o Offload-Fallback
vega@lyra% ./Offload-Fallback
Hello World from main()! I see 48 logical processors.
Hello from offload on MIC with 244 logical processors.
vega@lyra% sudo systemctl stop mpss # Disabling coprocessors
vega@lyra% ./Offload-Fallback
Hello World from main()! I see 48 logical processors.
Hello from offload on CPU with 48 logical processors.
```

Listing 2.28: Compiling and running `Offload-Fallback.cc`.

A related clause of the offload pragma allows to perform conditional offload, i.e., to choose at runtime whether to perform offload or execute the code on the CPU. This clause is `if`, and it takes one argument. If the argument evaluates to a non-zero value or boolean “true”, the offload will be sent to a coprocessor. If it evaluates to 0 or boolean “false”, the calculation in the scope of the offload pragma will fall back to host CPU execution.

```
#pragma offload target(mic) if(N>1000)
```

Conditional offload can be used, for example, to prevent offload of a problem that is too small to pay off for the offload overhead (see Section 1.3.4), or to distribute work between coprocessors and the CPU (see Section 2.4.1).

2.2.8. Offload Diagnostics

It is possible to generate diagnostic output for offload applications. This can be done using the Linux environment variable `OFFLOAD_REPORT` or the function `_Offload_report`.

a) If `OFFLOAD_REPORT` is unset, no diagnostic output is produced (this is the default behavior).
b) `OFFLOAD_REPORT=1` prints information on the offload locations (lines of code) and times.
c) `OFFLOAD_REPORT=2`, in addition, produces information regarding the amount of data traffic.
d) `OFFLOAD_REPORT=3` gives additional details: device initialization and individual variable transfers.

Listing 2.29 demonstrates the report produced by setting the environment variable `OFFLOAD_REPORT=2`.

```
vega@lyra% ./Offload-Fallback
Hello World from main()! I see 48 logical processors.
Hello from offload on MIC with 244 logical processors.
vega@lyra%
vega@lyra% export OFFLOAD_REPORT=2
vega@lyra% ./Offload-Fallback
Hello World from main()! I see 48 logical processors.
[Offload] [MIC 0] [File]                    Offload-Fallback.cc
[Offload] [MIC 0] [Line]                    6
[Offload] [MIC 0] [Tag]                     Tag 0
Hello from offload on MIC with 244 logical processors.
[Offload] [HOST]  [Tag 0] [CPU Time]        0.495117(seconds)
[Offload] [MIC 0] [Tag 0] [CPU->MIC Data]   0 (bytes)
[Offload] [MIC 0] [Tag 0] [MIC Time]        0.000366(seconds)
[Offload] [MIC 0] [Tag 0] [MIC->CPU Data]   0 (bytes)
```

Listing 2.29: Using the environment variable `OFFLOAD_REPORT` to monitor the execution of an application performing offload to an Intel Xeon Phi coprocessor.

2.2.9. Environment Variables and `MIC_ENV_PREFIX`

Default Behavior

Environment variables defined on the host are automatically forwarded to the coprocessor when an offload application is launched.

Example: suppose the user sets on the host the environment variable `MYPARAMETER=myval`. In the offloaded program, environment variable `MYPARAMETER` on the coprocessor will have the value `myval`.

Avoiding Collisions with Host

To avoid environment variable name collisions on the host and the coprocessor, the the environment variable `MIC_ENV_PREFIX` can be used. When this variable is set, only environment variables with names beginning with the prefix are forwarded, and the prefix is stripped on the coprocessor.

Example: suppose the user sets `MIC_ENV_PREFIX=PHI`, and on the host `PHI_MYPARAMETER=val2`. In the offloaded part of the application running on the coprocessor, the environment variable `MYPARAMETER` will have the value `val2` (i.e., the variable name is stripped of “`PHI_`”, and the variable value is passed to the coprocessor).

Alternative Syntax

It is also possible to set multiple environment variables on all coprocessor by setting `*_VAR=variable1=value1|variable2=value2|...`, where `*` is the value of `MIC_ENV_PREFIX`.

Example: suppose the user sets on host `MIC_ENV_PREFIX=PHI` and `PHI_VAR=MYPARAMETER=val3|MYOTHERPARAMETER=val4`. In the offload region, the variable `MYPARAMETER` will have the value `val3`, and `MYOTHERPARAMETER` will have the value `val4`.

Specific Values for Specific Coprocessors

Finally, it is possible to set specific variables for pecific coprocessors by using the prefix `*_0`, `*_1`, etc., where `*` is the value of `MIC_ENV_PREFIX`.

Example: with `MIC_ENV_PREFIX=PHI`, and `PHI_0_MYPAR=val4`, and `PHI_1_VAR=MYNEWPAR=val5|MYOTHERPAR=val6`. In the offload region on coprocessor `mic0`, the variable `MYPAR` will have the value `val4`, and on `mic1`, `MYNEWPAR=val5`, and `MYOTHERPAR=val6`.

Example

Listing 2.30 and Listing 2.31 demonstrate environment forwarding and the effect of `MIC_ENV_PREFIX`.

```
#include <cstdio>
#include <cstdlib>

int main(){
#pragma offload target (mic)
  {
    char* mypar = getenv("MYPARAMETER");
    if (mypar)
      printf("MYPARAMETER=%s on the coprocessor.\n", mypar);
    else
      printf("MYPARAMETER is not defined on the coprocessor.\n");
} }
```

Listing 2.30: This code, `Offload-Env.cc`, prints the value of the environment variable `MYPARAMETER` on the coprocessor.

```
vega@lyra% icpc Offload-Env.cc -o Offload-Env
vega@lyra% ./Offload-Env
MYPARAMETER is not defined on the coprocessor.
vega@lyra%
vega@lyra% export MYPARAMETER=val
vega@lyra% ./Offload-Env
MYPARAMETER=val on the coprocessor.
vega@lyra%
vega@lyra% export MIC_ENV_PREFIX=PHI
vega@lyra% ./Offload-Env
MYPARAMETER is not defined on the coprocessor.
vega@lyra%
vega@lyra% export PHI_MYPARAMETER=val2
vega@lyra% ./Offload-Env
MYPARAMETER=val2 on the coprocessor.
```

Listing 2.31: Effect of `MIC_ENV_PREFIX` on environment variable passing to the coprocessor.

2.2.10. Proxy Console I/O

The example in Section 2.2.1 demonstrates that the code executing on the coprocessor can output data to the standard output stream, and this data appears on the host console. How does that happen?

When console output operations are called on an Intel Xeon Phi coprocessor, *e.g.* with `printf()`, they are buffered in the coprocessor OS and later passed on (proxied) to the host console by the `COI` (Coprocessor Offload Infrastructure), a daemon running on the coprocessor. The communication scheme of the console proxy is shown in Figure 2.4.

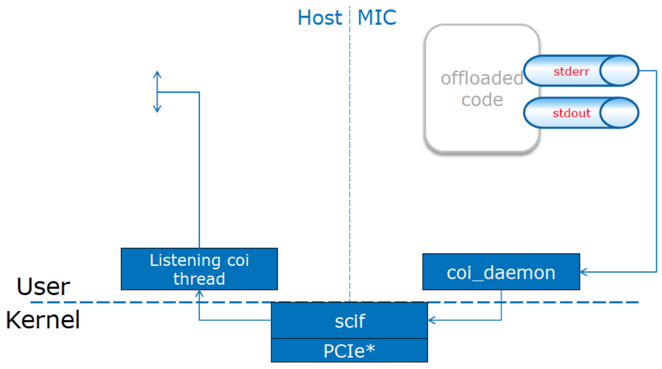


Figure 2.4: Proxy console I/O diagram. Output to standard output and standard error streams on the coprocessor is buffered and passed on to the host terminal. *Image credit: Intel Corporation.*

In the case of the “Hello World” code (Listing 2.14 and Listing 2.15), buffering delays caused the stream from the coprocessor to be printed out *after* the host had finished the last `printf()` function call (line 11 in Listing 2.14).

The output buffer must be flushed using the `fflush(0)` function of the `stdio` library in order to ensure consistent operation of the console proxy. Without `fflush(0)` in the coprocessor code, the output of the `printf` function might be lost if the program is terminated prematurely.

The proxy console I/O service is enabled by default. It can be disabled by setting the environment variable `MIC_PROXY_IO=0`. Despite the name "Proxy console I/O", the `coi` service proxies only the standard output and standard error streams. Proxy console input is not supported.

2.2.11. Review: Explicit Offload Model

The key language constructs used for the offload model programming are listed below. For complete description, refer to the Intel C++ compiler reference.

1. `__attribute__((target(mic)))` is a declaration qualifier that indicates that the declared object (global/static variable or function) must be compiled into the target code. Examples in Listing 2.32 show how the non-scalar variable `data` can be made visible in the scope of the target code and the function `CountNonzero()` can be compiled for the coprocessor.

```
__attribute__((target(mic))) int data[1000];
__attribute__((target(mic))) int MyFunction(void* arg);
```

Listing 2.32: Illustration of `__attribute__((target(mic)))` usage. Variables and functions marked with this qualifier may be used in offloaded code.

2. `#pragma offload_attribute(push, target(mic))` and `#pragma offload_attribute(pop)` can be used instead of the qualifier `__attribute__((target(mic)))` when multiple consecutive elements in a source file need to be included in the offload code. Example in Listing 2.33 specifies that several arrays and all of the variables and functions declared in the header file `myvariables.h` should be accessible to the coprocessor code.

```
#pragma offload_attribute(push, target(mic))

int* data; // Apply the offload qualifier to a pointer-based array,
int MyFunction(void* arg); // a function
#include "myvariables.h" // or even a whole file

#pragma offload_attribute(pop)
```

Listing 2.33: Illustration of `offload_attribute(push)` usage.

3. `#pragma offload_transfer target(mic)` requests that certain non-scalar data must be copied to the coprocessor. This pragma

takes a number of clauses to specify data traffic. These clauses are described in Section 2.2.4.

```
#pragma offload_transfer target (mic:0) \
    in(ptrdata : length(N) alloc_if(1) free_if(0))
```

Listing 2.34: Illustration of `#pragma offload_transfer` usage.

The code in Listing 2.34 requests that array `data` of `N` elements must be transferred `in`, i.e., from the host to the coprocessor number 0, and that the memory on the coprocessor must be allocated before the offload, but not freed afterwards. The symbol "\" is used to break the pragma code into several lines. This is a blocking operation, which means that code execution will stop until the transfer is complete. It is also possible to request a non-blocking (asynchronous) offload, using the `signal` clause, as described in Section 2.2.4.

In this example, no operations will be applied to the transferred data. To request some processing along with data transfer, `#pragma offload` should be used, as described below.

4. `#pragma offload target(mic)` specifies that the code following this pragma must be executed on the coprocessor if possible. This pragma takes a number of clauses to specify data traffic. These clauses are described in Section 2.2.4. The function `MyFunction()` in Listing 2.35 will be offloaded to the coprocessor if one is available, or run on the host otherwise.

```
#pragma offload target(mic) in(N) in(ptrdata : length(N)) optional
  {
    int ct = MyFunction(ptrdata);
  }
```

Listing 2.35: Illustration of `#pragma offload` usage.

2.3. Shared Virtual Memory Model

In the explicit offload model discussed in Section 2.2, the programmer must explicitly set up data marshalling in order to send data to, or from, the coprocessor. An alternative to this approach is the shared virtual memory programming model, where data marshalling is neither necessary nor possible. In the shared virtual memory model, the programmer uses special memory allocators and offload calls, which allow the runtime system to automatically transfer the data from host to the coprocessor and back. Furthermore, in this model the memory addresses of virtual-shared heap objects are the same on the host and on the coprocessor. The shared virtual memory model is only available in C and C++; it is not available in Fortran.

The shared virtual memory approach has several advantages over the explicit offload model:

1. When a large number of objects are shared with the coprocessor, the shared virtual memory model may require less programming and debugging effort than the data marshalling approach;

2. Shared virtual memory enables the sharing of complex (i.e., not bitwise-copyable) objects, which the offload pragmas do not allow. For example, C++ classes cannot be transferred using offload pragmas, but can be shared with the coprocessor in the shared virtual memory model.

3. The runtime system monitors the shared virtual memory for changes. When a part of an array in shared virtual memory is modified on the host, this changes is propagated to the coprocessor at the start of the next offload. When the coprocessor modifies shared virtual memory, changes are propagated back to the host at the end of offload. Modified arrays are not copied entirely, only the parts that were written to are synchronized. The granularity of synchronization is the size of virtual memory pages.

4. Memory buffer retention and data persistence on the coprocessor are maintained automatically, without the programmer's involvement.

To use the shared virtual memory model, the programmer has to specify what data and how it should be accessed by the target:

- Programmer marks variables that need to be *shared* between the host system and the target.
- The same shared variable can then be used in both host and coprocessor code.
- Runtime system *automatically maintains coherence* at the beginning and at the end of offload statements. Upon entry to the offload code, data modified on the host are automatically copied to the target, and upon exit from the offload call, data modified on the target are copied to the host.

The shared virtual memory model syntax is based on two keywords: `_Cilk_shared` and `_Cilk_offload`. Note that, despite `_Cilk` being a part of these keywords, the programmer is not limited to using Intel Cilk Plus to parallelize the offloaded code. OpenMP, Pthreads, and other frameworks can be used within the offloaded segment. See Section 3.2 for more information about Intel Cilk Plus and OpenMP.

2.3.1. Offloading Functions

Functions that may be offloaded to the coprocessor must be marked with the keyword `_Cilk_shared`. This keyword is similar to the qualifier `__attribute__((target(mic)))` in the explicit offload model. To offload a function to an Intel Xeon Phi coprocessor, place the keyword `_Cilk_offload` before the offload call. This is the counterpart of `#pragma offload` in the explicit offload model.

Below is an illustration of a "Hello World" program with one function call in the shared virtual memory model.

```
#include <cstdio>

_Cilk_shared void MyFunction() {
#ifdef __MIC__
  printf("Hello from Offload\n");
#else
  printf("Offload failed, running on CPU\n");
#endif
}

int main() {
  printf("Hello from Host\n");
  _Cilk_offload MyFunction();
}
```

Listing 2.36: File `Shared-Hello.cc`, a "Hello World" program with the shared virtual memory model of Intel Xeon Phi coprocessor programming.

```
vega@lyra% icpc -o Shared-Hello Shared-Hello.cc
vega@lyra% ./Shared-Hello
Hello from Host
Hello from Offload
```

Listing 2.37: Compiling and running `Shared-Hello.cc`.

Note that for all shared virtual memory model examples, we will use the C++ language and the Intel C++ Compiler `icpc`.

2.3.2. Sharing and Offloading Objects

The keyword `_Cilk_shared` must also be used to mark variables that are meant to be shared with the coprocessor in the shared virtual memory model. A function marked with `_Cilk_shared` and offloaded with `_Cilk_offload` may read and modify shared data on the coprocessor. The modified values will be automatically synchronized between the host and the coprocessor. Listing 2.38 illustrates the usage of this model.

```
#include <cstdio>
#define N 1000
_Cilk_shared int A[N];

void Initialize() { // Runs on host
  for (int i = 0; i < N; i++)
    A[i] = i;
}

_Cilk_shared void Modify() { // Runs on coprocessor
#ifdef __MIC__
  for (int i = 0; i < N; i++)
    A[i] = -A[i];
#else
  printf("Offload to coprocessor failed!\n");
#endif
}

void Verify() { // Runs on host
  bool errors = false;
  for (int i = 0; i < N; i++) errors |= (A[i] != -i);
  printf("%s\n", (errors ? "ERROR" : "CORRECT"));
}

int main(int argc, char *argv[]) {
  Initialize();
  _Cilk_offload Modify(); // Function call on coprocessor
  Verify();
}
```

Listing 2.38: Example of using the shared virtual memory and offloading calculations with `_Cilk_shared` and `_Cilk_offload` of the function call.

2.3.3. Dynamic Allocation in Shared Virtual Memory

Dynamic memory shared between the host and the target must be allocated and deallocated with the special functions listed below. Details are available in the Intel C++ Compiler Reference Manual.

Allocators:

```
void *_Offload_shared_malloc(size_t size)
void *_Offload_shared_aligned_malloc(size_t size, size_t alignment)
```

Deallocators:

```
void *_Offload_shared_free(void* ptr)
void *_Offload_shared_aligned_free(void* ptr)
```

These allocators are similar to `malloc` and `_mm_malloc` with the corresponding `free` and `_mm_free`. However, the pointers returned by these special allocators are pointing to a special shared virtual address space. In this space, the values of pointers on host are exactly the same as on the coprocessor in the offload region.

Memory allocated using these special allocators is automatically monitored for changes and synchronized between the host and the coprocessor at the beginning and end of each offload instantiated with `_Cilk_offload`.

The aligned versions of allocators return memory addresses aligned on a user-specified boundary. See Section 3.1.4 for more information on data alignment.

There is a requirement is that the pointer variable that stores the pointer must be shared itself. This is achieved by declaring, for example, a pointer to an array of integers, as `int* _Cilk_shared ptr`.

Listing 2.39 demonstrates how pointer-based data can be dynamically allocated and used on the target code, and Listing 2.40 demonstrates the result.

```
#include <cstdio>
#define N 10000
int* _Cilk_shared data; // Shared pointer to shared data
int _Cilk_shared sum; // Shared scalar

_Cilk_shared void ComputeSum() {
#ifdef __MIC__
  printf("Address of data[0] on coprocessor: %p\n", &data[0]);
  fflush(0);
  sum = 0;
  for (int i = 0; i < N; ++i)
    sum += data[i]; // Compute sum on coprocessor
#else
  printf("Offload to coprocessor failed!\n");
#endif
}

int main() {
  data = (_Cilk_shared int*)_Offload_shared_malloc(N*sizeof(float));
  for (int i = 0; i < N; i++) // Initialize data on host
    data[i] = i;
  printf("Address of data[0] on host: %p\n", &data[0]);
  _Cilk_offload ComputeSum(); // Process data on coprocessor
  printf("%s\n", (sum==N*(N-1)/2 ? "CORRECT" : "ERROR"));
  _Offload_shared_free(data);
}
```

Listing 2.39: `Shared-Pointers.cc` demonstrates using `_Offload_shared_malloc` for dynamic shared virtual memory allocation in C/C++.

```
vega@lyra% icpc -o Shared-Pointers Shared-Pointers.cc
vega@lyra% ./Shared-Pointers
Address of data[0] on host: 0x820000030
Address of data[0] on coprocessor: 0x820000030
CORRECT
```

Listing 2.40: Output of the code in Listing 2.39

2.3.4. Classes in Shared Virtual Memory

Note that transferring bitwise-copyable objects, such as arrays, is possible both with the explicit offload model, and with shared virtual memory. However, sharing structures with pointer elements and C++ classes is only possible using shared virtual memory, as these objects are not bitwise-copyable. Listing 2.41 — Listing 2.43 illustrate how complex objects, such as structures and classes, can be shared between the host and the target.

```
#include <cstdio>
#include <cstring>

typedef struct {
  int  i;
  char c[10];
} person;

_Cilk_shared void SetPerson(_Cilk_shared person & p,
                            const char _Cilk_shared *name, const int i) {
#ifdef __MIC__
  p.i = i;
  strcpy(p.c, name);
  printf("On coprocessor: %d %s\n", p.i, p.c);
#else
  printf("Offload to coprocessor failed.\n");
#endif
  fflush(0);
}

person _Cilk_shared someone;
char _Cilk_shared who[10];

int main(){
  strcpy(who, "John"); // Store data in a _Cilk_shared variable
  _Cilk_offload SetPerson(someone, who, 1); // Initialize on MIC
  printf("On host: %d %s\n", someone.i, someone.c); // Use on host
}
```

Listing 2.41: `Shared-Struct.cc` demonstrates how a C++ structure can be synchronized between the host and the coprocessor using shared virtual memory.

Listing 2.42 shows the result:

```
vega@lyra % icpc -o Shared-Struct Shared-Struct.cc
vega@lyra % ./Shared-Struct
On coprocessor: 1 John
On host: 1 John
```

Listing 2.42: Output of the code in Listing 2.41

In the above example, the function `SetPerson` accepts an argument initialized on the host. However, the function itself is executed on the coprocessor. It produces an object (`someone`), which is later used on the host.

The example in Listing 2.41 and Listing 2.42 contains a C structure. Because all members of this structure are contiguous in memory, and no pointers are stored in it, this structure is, in fact, bitwise-copyable. It could have been offloaded to the coprocessor using `#pragma offload` – for example, by camouflaging it as an array of type `char`.

However, for a more complex object, such as the class shown in Listing 2.43 and Listing 2.44, `#pragma offload` would not work. `class Person` contains a member of type `char*`, which is a pointer. Even if we copy the pointer value using `#pragma offload`, the memory referred to by this pointer will not be copied. In this case, the shared virtual memory model can help to perform offload.

Note that in the code, we apply the qualifier `_Cilk_shared` to the class declaration. This makes all members of the class, including pointers, also `_Cilk_shared`.

Another point worth mentioning is that shared memory allocation in this case occurs on the coprocessor in function `Set(...)`. The result is still as expected: data in the shared buffer are copied over to the host at the end of offload.

```
#include <cstdio>
#include <cstring>

class _Cilk_shared Person {
 public:
  char* c; // Pointer member - not bitwise-copyable

  Person() { c=NULL; } // Construct without memory allocation

  void Set(const char _Cilk_shared * name) {
#ifdef __MIC__
    c=(char*)_Offload_shared_malloc(strlen(name)); // Memory alloc
    strcpy(c, name);
    printf("On coprocessor: %s\n", c);
#else
    printf("Offload to coprocessor failed.\n");
#endif
    fflush(0);
  }
};

Person _Cilk_shared someone;
char _Cilk_shared who[10];

int main(){
  strcpy(who, "Mary");
  _Cilk_offload someone.Set(who);
  printf("On host: %s\n", someone.c);
}
```

Listing 2.43: `Shared-Class.cc` demonstrates a shared virtual C++ with pointer members.

```
vega@lyra% icpc -o Shared-Class Shared-Class.cc
vega@lyra% ./Shared-Class.cc
On host: Mary
On coprocessor: Mary
```

Listing 2.44: Output of the code in Listing 2.43

2.3.5. Placement Operator `new` for Shared Classes

To allocate buffers in shared virtual memory, the special allocator function `_Offload_shared_malloc` must be used. However, what if a shared class needs to be allocated using operator `new`? Regular usage of operator `new` to allocate memory and call the class constructor is not applicable in this case, because it allocates local, and not shared, memory. However, the placement version of operator `new` can be used in tandem with `_Offload_shared_malloc` in order to create a shared class.

The placement version of operator `new` calls the class constructor without allocating memory for the class. In this version, `new` takes one argument of type `void*` pointing to the pre-allocated memory. This operator is made available by including the header file `<new>`.

```
class MyClass { /* ... */ };
// ...

  char* buf = (char*) malloc(sizeof(MyClass));
  MyClass* obj = new(buf) MyClass();
```

Listing 2.45: Placement version of operator `new`

In the above code snippet, allocator `malloc` performs allocation of a sufficient amount of memory. Then operator `new` places the object of type `MyClass` into that memory and calls the class constructor. Because there is argument `(buf)` passed to `new`, no memory allocation occurs in `new`. The creation of the object `obj` in the above code is equivalent to, simply,

```
MyClass* obj = new MyClass();
```

Listing 2.46: Regular version of operator `new`

The placement version of `new` allows the programmer to choose the memory allocator, or implement a custom one. Specifically, for the shared virtual memory model, we can call `_Offload_shared_malloc` to allocate memory and then use the placement version of `new` to construct the class in that memory. This method is shown in Listing 2.47 and Listing 2.48.

```
#include <cstdio>
#include <cstdlib>
#include <new>

class _Cilk_shared MyClass {
  int i;
 public:

  MyClass(){ i = 1000; };

  void Print(){
#ifdef __MIC__
    printf("On coprocessor: ");
#else
    printf("On host: ");
#endif
    printf("%d\n", i); fflush(0);
  }
};

MyClass* _Cilk_shared sharedData;

int main() {
  sharedData = new(_Offload_shared_malloc(sizeof(MyClass)))MyClass;
  _Cilk_offload sharedData->Print(); // Check value on coprocessor
  sharedData->Print(); // Check value on host
}
```

Listing 2.47: `Shared-PlacementNew.cc` illustrates using the placement version of operator `new` to allocate a C++ class in shared virtual memory.

```
vega@lyra% icpc -o Shared-PlacementNew Shared-PlacementNew.cc
vega@lyra% ./Shared-PlacementNew
On coprocessor: 1000
On host: 1000
```

Listing 2.48: Compilation and execution of `Shared-PlacementNew.cc`.

2.3.6. Asynchronous Offload

To perform asynchronous offload in the shared virtual memory model, the keyword `_Cilk_offload` must be prepended by `_Cilk_spawn`. The latter is a part of the Intel Cilk Plus API, also applicable to asynchronous execution of regular functions on host. See Section 3.3 for more detail.

After spawning one or more offloads, the code may need to issue a synchronization instruction `_Cilk_sync` in order to wait for the spawned functions and avoid unpredictable program behavior. If offloads are spawned from a function, synchronization is implicitly performed upon exit from the function.

Just like with the explicit offload model, it may be necessary to specify the target of offload. This is done by changing the offload keyword to `_Cilk_offload_to(i)`, where `i` is the zero-based number of the target coprocessor.

Listing 2.49 illustrates using asynchronous offloads to overlap offloaded calculations with a workload on the host.

```
// Offload to mic0, but do not want for completion
_Cilk_Spawn _Cilk_offload_to(0) MyFunction(myObject[0]);

// Offload to mic1, but do not want for completion
_Cilk_spawn _Cilk_offload_to(1) MyFunction(myObject[1]);

// Run on host (normal blocking function call)
MyFunction(myObject[2]); // Compute on host

// Wait for completion of offloads
_Cilk_sync;
```

Listing 2.49: Asynchronous offload in the shared virtual memory model.

2.3.7. Summary for Shared Virtual Memory Model

Table 2.1 summarizes the usage of the shared virtual memory model API.

Entity	Syntax	Effect
Function	`int _Cilk_shared func();`	Executable code for both host and target; may be called from either side
Global variable	`_Cilk_shared int x = 0;`	Visible on both sides
File/Function static	`static _Cilk_shared int x;`	Visible on both sides, only to code within the file/function
Class	`class _Cilk_shared x {...};`	Class methods, members, and operators are available on both sides
Pointer to shared data	`int _Cilk_shared *p;`	*p* is local (not shared), can point to shared data
A shared pointer	`int *_Cilk_shared p;`	*p* is shared; should only point at shared data
Offloading a function call	`_Cilk_offload func(y);`	*func* executes on coprocessor if possible
	`_Cilk_offload_to(n) func(y);`	*func* **must** be executed on the specified (n-th) coprocessor
Offloading asynchronously	`_Cilk_spawn _Cilk_offload func(y)`	Non-blocking offload
Offload a parallel for-loop	`_Cilk_offload _Cilk_for(i=0; i<N; i++) {}`	Loop executes in parallel on target. The loop is implicitly outlined as a function call

Table 2.1: Shared virtual memory programming model API.

2.4. Using Multiple Coprocessors

We have discussed in Sections 2.4.1, 2.4.2 and 2.4.3 how to use a single Intel Xeon Phi coprocessor using native (or MPI) applications, in the explicit offload model or in the shared virtual memory model. This section describes how multiple coprocessors can be used in these programming models.

There are several ways to employ several Intel Xeon Phi coprocessors simultaneously. The best method depends on the structure and parallel algorithm of the application.

In distributed memory applications using MPI, there exists a multitude of methods for utilizing multiple hosts and multiple devices (see Section 3.4.1). However, all of these methods can be placed into one of the following two categories:

(1) MPI processes run only on CPUs and offload to coprocessors, and
(2) MPI processes run as native applications on coprocessors without using the CPUs (or run natively on coprocessors as well as on CPUs).

For applications utilizing MPI in mode (1), and for offload applications using only a single host, multiple coprocessors per host can be utilized using a combination of approaches described in Section 2.4.1 and Section 2.4.2:

(1a) spawning multiple threads on the host, each performing offload to the respective coprocessor, and

(1b) performing asynchronous offloads from one host thread.

For MPI applications in mode (2), scaling across multiple coprocessors occurs naturally, however, bridged networking is required for peer-to-peer communication (see Section 1.2.5).

We will start with the discussion of the offload model, in its explicit implementation and in the shared virtual memory variation, and then proceed to discussing the usage of MPI for heterogeneous applications with multiple Intel Xeon Phi coprocessors. Note that this section is not a tutorial on OpenMP, Intel Cilk Plus or MPI. Refer to Chapter 3 for information expressing parallelism using these frameworks.

2.4.1. Multiple Coprocessors with Explicit Offload

Querying the Number of Devices

In the host code, the number of available Intel Xeon Phi coprocessors can be queried with a call to function `_Offload_number_of_devices()`:

```
const int numDevices = _Offload_number_of_devices();
```

Specifying an Explicit Offload Target

With several Intel Xeon Phi coprocessors installed in a system, it is possible to request offload to a specific coprocessor. This has been demonstrated in Listing 2.25, where `mic:0` indicates that the offload must be performed to the first coprocessor in the system. Another example is shown in Listing 2.50.

```
#pragma offload target(mic: 0)
{
  foo();
}
```

Listing 2.50: Directing explicit offload to the first Intel Xeon Phi coprocessor in the system.

Specifying a target number of 0 or greater indicates that the call applies to the coprocessor with the corresponding zero-based number. For a target number greater than or equal to the coprocessor count, the offload will be directed to the coprocessor equal to the target number modulo device count. For example, with 4 coprocessors in the system, `mic:1`, `mic:5`, `mic:9`, etc., direct offload to the second coprocessor.

Specifying `mic:-1` instead will invite the runtime system to choose a coprocessor or fail if none are found.

In applications using asynchronous offloads, specifying target numbers is critical, as waiting for a signal from the wrong coprocessor can result in the code hanging. The same applies to applications that use data persistence on the coprocessor. If a persistent array is allocated on a specific coprocessor, but an offload pragma tries to re-use that array on a different coprocessor, a runtime error will occur.

Explicit Offload from Multiple Host Threads

One of the methods to employ multiple coprocessors is spawning several host tasks/threads and offloading to the respective device from every thread. Listing 2.51 illustrates how to do this with OpenMP. Here, `#pragma omp parallel` makes all offloads launch concurrently in different threads.

```
#include <cstdio>
#include <unistd.h>
int main(){
  int numDevices = _Offload_number_of_devices();
  printf("Running %d parallel threads\n", numDevices);
#pragma omp parallel num_threads(numDevices)
  { const int i = omp_get_thread_num();
    printf("Starting blocking offload to mic%d...\n", i);
#pragma offload target(mic:i)
    { system("hostname");  sleep(1); }
    printf("Done with offload to mic%d.\n", i);
} }
```

```
vega@lyra% icpc -qopenmp -o Multiple-Threads Multiple-Threads.cc
vega@lyra% ./Multiple-Threads
Running 4 parallel threads
Starting blocking offload to mic0...
Starting blocking offload to mic1...
Starting blocking offload to mic3...
Starting blocking offload to mic2...
host-mic0
host-mic3
host-mic2
host-mic1
Done with offload to mic0.
Done with offload to mic3.
Done with offload to mic2.
Done with offload to mic1.
```

Listing 2.51: Top: code `Multiple-Threads.cc` illustrates of employing several coprocessors simultaneously using multiple host threads. Bottom: compiling with `-qopenmp` in order to enable `#pragma omp`.

Load Balancing with Offload from Threads

The method presented above can be used to automatically distribute a set of work-items across all available coprocessors as illustrated in Listing 2.52.

```
#include <omp.h>
#include <cstdio>
#include <unistd.h>
int main(){
  srand(0);
  const int nWorkItems=8;
  int workItem[nWorkItems];
  for (int i = 0; i < nWorkItems; i++)
    workItem[i] = 1 + rand()%5;
  int numDevices = _Offload_number_of_devices();
  printf("Running %d parallel threads\n", numDevices);
#pragma omp parallel for schedule(dynamic,1) \
        num_threads(numDevices)
  for (int i = 0; i < nWorkItems; i++) {
    const int iMIC = omp_get_thread_num();
    printf("Offload   work item %d to mic%d...\n", i, iMIC);
#pragma offload target(mic:iMIC) in(i) in(workItem[i:1])
    { sleep(workItem[i]); }
} }
```

```
vega@lyra% icpc -qopenmp -o Multiple-Distribute \
> Multiple-Distribute.cc
vega@lyra% ./Multiple-Distribute
Running 4 parallel threads
Offload   work item 0 to mic1...
Offload   work item 1 to mic0...
Offload   work item 3 to mic3...
Offload   work item 2 to mic2...
Offload   work item 4 to mic3...
Offload   work item 5 to mic0...
Offload   work item 6 to mic2...
Offload   work item 7 to mic0...
```

Listing 2.52: Code `Multiple-Distribute.cc` is an illustration of distributing a large number of work items between offloads to multiple coprocessors.

In this case, only 4 threads will operate concurrently, because the value of `numDevices` is 4. Each of the threads will offload to the respective coprocessor: thread 0 to `mic0`, thread 1 to `mic1` and so on. This is the purpose of the assignment of `iMIC` in line 15.

Every time an offload in a thread finishes, the thread will pick up the next iteration in `i` and perform offload to the idle coprocessor. This behavior is enforced by the clause `schedule(dynamic,1)`. Note that the code uses the `sleep()` call to somewhat randomize the duration of offloads. Despite that, the load balancing behavior enforced by the dynamic scheduling clause maintains load on all devices. See Section 3.2.3 for more information about load scheduling in OpenMP.

Finally, Listing 2.52 illustrates how to share data between coprocessor. This is applicable to numerical algorithms where parallelism is achieved by partitioning the dataset between processors. In `#pragma offload`, the clause `in(workItem[i:1])` sends a segment array `workItem` at position `i` of length `1` into the coprocessor. This method of specifying the range of offloaded array is a part of the Intel Cilk Plus notation, further discussed in Section 3.1.7.

This approach to load balancing can be extended to heterogeneous applications which use coprocessors simultaneously with the host CPU. To do that, the number of threads must be increased by 1. The extra thread will perform fall-back to host execution instead of offload. This can be achieved with the clause `if` of `#pragma offload`:

```
#pragma offload target(mic:iMIC) if (iMIC<=numDevices)
```

We will not demonstrate this method and leave it to the reader to implement as an exercise.

Multiple Coprocessors with Asynchronous Explicit Offload

Another way to employ several coprocessors does not involve spawning multiple host threads. Instead, a single host threads spawns multiple asynchronous (i.e., non-blocking) offloads. This approach is illustrated in Listing 2.53.

```
 1 #include <cstdlib>
 2 #include <cstdio>
 3 int main(){
 4   int nDevices = _Offload_number_of_devices();
 5   int i, resp[nDevices];   resp[0:nDevices] = 0;
 6   for (i = 0; i < nDevices; i++) {
 7 #pragma offload target(mic:i) inout(resp[i:1]) signal(&resp[i])
 8     {
 9       // The offloaded job does not block execution on the host
10       resp[i] = 1;
11     }
12   }
13   for (i = 0; i < nDevices; i++) {
14     // This loop waits for all asynchronous offloads to finish
15 #pragma offload_wait target(mic:i) wait(&resp[i])
16   }
17   for (i = 0; i < nDevices; i++)
18     if (resp[i] == 1)
19       printf("OK: device %d responded\n", i);
20     else
21       printf("Error: device %d did not respond\n", i);
22 }
```

Listing 2.53: `Multiple-Async.cc` illustrates employing several Intel Xeon Phi coprocessors simultaneously using asynchronous offloads

In `Multiple-Async.cc`, the code uses only one host thread, but spawns multiple offloads in the for-loop in line 9. The asynchronous nature of offload is requested by the clause `signal`. For simplicity, the signal is chosen as a pointer to the array sent to the respective coprocessor. Loop in line 13 waits for signals. The arrival of each signal indicates the end of the offload.

2.4.2. Multiple Coprocessors in the Shared Virtual Memory Model

With the shared virtual memory model, multiple coprocessors can be employed similarly to the explicit offload model.

Querying the Number of Coprocessors

Querying the number of coprocessors in the shared virtual memory model is done in the same way as in the explicit offload model:

```
const int numDevices = _Offload_number_of_devices();
```

Specifying Offload Target

If there are several Intel Xeon Phi coprocessors present, the programmer can choose which one to use with the `_Cilk_offload_to(number)` keyword, as shown in Listing 2.54.

```
_Cilk_offload_to(i) func();
```

Listing 2.54: `_Cilk_offload_to(i)` will use Intel Xeon Phi coprocessor number `i` (counted from zero) for offloading. See also Section 2.4.1 for information about the rules of coprocessor specification.

Multiple Blocking Offloads in Threads

Listing 2.55 illustrates the same approach in the shared virtual memory model as Listing 2.51 in the explicit offload model. In this case, the loop in Line 11 is executed in parallel with the help of the Intel Cilk Plus library. It is expected that the number of available Cilk Plus workers is greater than the number of coprocessors in the system, and therefore, all offloads will start simultaneously. Section 3.2.3 explains parallel loops in Intel Cilk Plus.

```
1  #include <cstdio>
2  #include <cstdlib>
3  #include <unistd.h>
4
5  _Cilk_shared void MyWorkload() {  system("hostname");  sleep(1); }
6
7  int main(){
8    int numDevices = _Offload_number_of_devices();
9    _Cilk_for (int i = 0; i < numDevices; i++) {
10     printf("Starting blocking offload to mic%d...\n", i);
11     _Cilk_offload_to(i) MyWorkload();
12     printf("Done with offload to mic%d.\n", i);
13 } }
```

```
vega@lyra% icpc -o Multiple-Cilk Multiple-Cilk.cc
vega@lyra% ./Multiple-Cilk
Starting blocking offload to mic0...
Starting blocking offload to mic1...
Starting blocking offload to mic2...
Starting blocking offload to mic3...
host-mic0
host-mic1
host-mic2
host-mic3
Done with offload to mic0.
Done with offload to mic1.
Done with offload to mic3.
Done with offload to mic2.
```

Listing 2.55: `Multiple-Cilk.cc`, an illustration of employing several Intel Xeon Phi coprocessors simultaneously using thread parallelism on host.

Multiple Asynchronous Offloads

Instead of using parallel threads, it is possible to use a single-threaded application with asynchronous offloads to different coprocessors. In the shared virtual memory model, asynchronous offload is instantiated with the keyword `_Cilk_spawn`, and the instruction to wait for spawned offloads is `_Cilk_sync`.

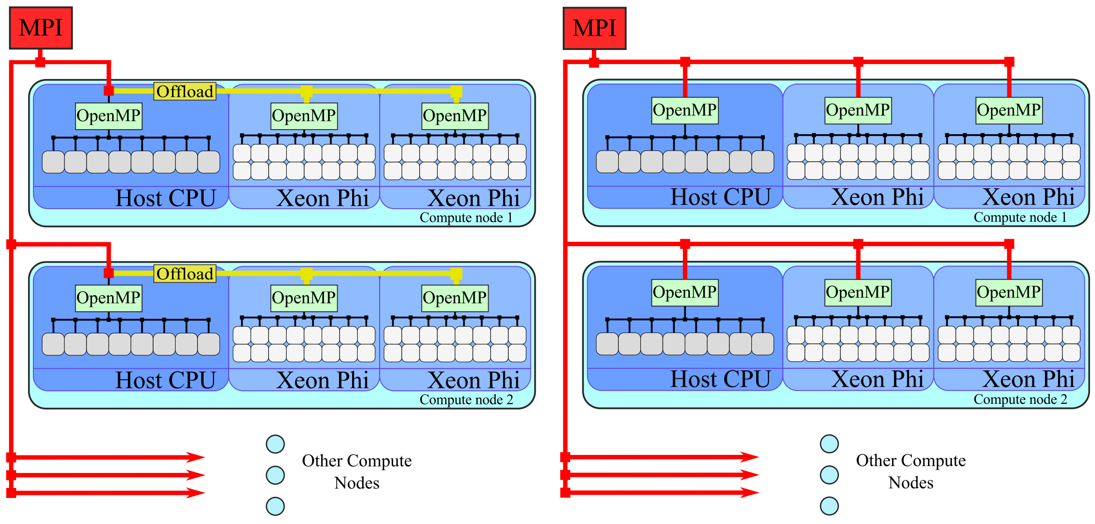
Listing 2.56 illustrates this approach. This code is analogous to the explicit offload code with asynchronous offload in Listing 2.53.

```
#include <cstdlib>
#include <cstdio>

void _Cilk_shared Respond(int _Cilk_shared & a) {
  a = 1;
}

const int nDevices = _Offload_number_of_devices();
_Cilk_shared int response[nDevices];

int main(){
  response[0:nDevices] = 0;
  for (int i = 0; i < nDevices; i++) {
    _Cilk_spawn _Cilk_offload_to(i)
      Respond(response[i]);
  }
  _Cilk_sync; // Wait for end of offloads
  for (int i = 0; i < nDevices; i++)
    if (response[i] == 1)
      printf("OK: device %d responded\n", i);
    else
      printf("Error: device %d did not respond\n", i);
}
```

Listing 2.56: `Multiple-Spawn.cc` illustrates employing several Intel Xeon Phi coprocessors simultaneously using asynchronous offload in the shared virtual memory model.

2.4.3. Multiple Coprocessors with MPI

In MIC-enabled computing clusters, there are two fundamental approaches to running MPI jobs that employ Intel Xeon Phi coprocessors.



(1) (left) MPI processes run only on processors and perform offload to coprocessors attached to their respective host. In this case, MPI jobs are submitted as if Intel Xeon Phi coprocessors are not installed, but one or several coprocessors per system can be used as described in Section 2.4.1 and Section 2.4.2. It is possible to employ this method with either the bridge, or the static pair network topology of coprocessors (see Section 1.5.21).

(2) (right) MPI processes run as native applications on coprocessors (or on coprocessors as well as processors). The procedure employing multiple coprocessors with this approach is presented in this section. Note that in order to use this approach with more than one host (i.e., on a cluster), the network connections of Intel Xeon Phi coprocessors must be configured in the bridge topology, so that all coprocessors are directly IP-addressable on the same private network as the hosts. In this section, we restrict the examples to a single host with multiple coprocessors, and therefore the network configuration is unimportant.

Code

Examples in this Section re-use the “Hello World” code from Listing 2.11. For convenience, this code is reproduced below. In the rest of this section, we assume that the MPI library has been NFS-shared with coprocessors, and environment variables initialized, as discussed in Section 2.1.5.

```
#include <mpi.h>
int main (int argc, char *argv[]) {
  int rank, size, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name (name, &namelen);
  printf ("Hello World from rank %d running on %s!\n", rank, name);
  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) printf("MPI World size = %d processes\n", size);
  MPI_Finalize ();
}
```

Fabrics

Depending on the Intel MPI version, the library may try to use the InfiniBand fabric for communication by default. Even with one compute node or workstation, virtual InfiniBand may be configured using OFED (see Section 1.2.5). However, if it is not done, `mpirun` may return error messages about problems with protocol `dapl`. If that happens, the owner of the system may either install OFED, or force MPI to use the virtual Ethernet fabric as shown below. The latter solution will result in slower MPI traffic, but it may be sufficient for some applications, especially orientation examples.

```
vega@lyra% export I_MPI_FABRICS=tcp
```

See Section 4.7.4 for more information about fabrics.

Launching MPI Applications on Coprocessor from Host

First, we demonstrate launching an MPI job on one coprocessor from the host (see Listing 2.57). In this case, an additional environment variable, `I_MPI_MIC`, must be set on the host. The argument `-host mic0` passed to `mpirun` sends the job to coprocessor 0.

```
vega@lyra% mpiicpc -o MPI-Hello.MIC MPI-Hello.cc -mmic
vega@lyra% scp MPI-Hello.MIC mic0:~/
MPI-Hello.MIC         100%   12KB  12.4KB/s   00:00
vega@lyra% export I_MPI_MIC=1
vega@lyra% mpirun -host mic0 -np 2 ~/MPI-Hello.MIC
Hello World from rank 1 running on mic0!
Hello World from rank 0 running on mic0!
MPI World size = 2 processes
```

Listing 2.57: Launching an Intel MPI application on one coprocessor from the host.

MPI Applications on Multiple Coprocessors

To start the application on two coprocessors, we can specify a list of hosts using the separator ':', as shown in Listing 2.58. If bridged network topology is configured, this command can also accept remote coprocessors and remote hosts.

```
vega@lyra% scp MPI-Hello.MIC mic1:~/
MPI-Hello.MIC         100%   12KB  12.4KB/s   00:00
vega@lyra% mpirun \
> -host mic0 -np 2 ~/MPI-Hello.MIC : \
> -host mic1 -np 2 ~/MPI-Hello.MIC
Hello World from rank 2 running on mic1!
Hello World from rank 3 running on mic1!
Hello World from rank 1 running on mic0!
Hello World from rank 0 running on mic0!
MPI World size = 4 processes
```

Listing 2.58: Launching an Intel MPI application on two coprocessors from the host.

MPI Machine File

In clusters, in order to run jobs on hundreds of hosts and coprocessors, using a long command line with multiple ‘:’ separators is impractical. In this situation, it is better to use a machine file with the list of machines instead of individual hosts (or coprocessors), as demonstrated in Listing 2.59.

```
vega@lyra% cat mymachines.txt
mic0:2
mic1:2
vega@lyra% mpirun -machinefile mymachines.txt ~/MPI-Hello.MIC
Hello World from rank 3 running on lyra-mic1!
Hello World from rank 2 running on lyra-mic1!
Hello World from rank 1 running on lyra-mic0!
Hello World from rank 0 running on lyra-mic0!
MPI World size = 4 processes
```

Listing 2.59: Launching an Intel MPI application using a machine file.

There is a different way to use the machine file, as shown in Listing 2.60 Note that in the first case, ranks 0 and 1 were placed on `mic0` and ranks 2 and 3 on `mic1`, while in the second case, rank 0 went to `mic0`, rank 1 to `mic1`, rank 2 again to `mic0`, rank 3 to `mic1`. Rank ordering may be important when data partition and message traffic is considered.

```
vega@lyra% cat mymachines.txt
mic0
mic1
vega@lyra% mpirun -np 4 -machinefile mymachines.txt ~/MPI-Hello.MIC
Hello World from rank 3 running on lyra-mic1!
Hello World from rank 1 running on lyra-mic1!
Hello World from rank 2 running on lyra-mic0!
Hello World from rank 0 running on lyra-mic0!
MPI World size = 4 processes
```

Listing 2.60: Launching an Intel MPI application using a machine file. Permuted order of ranks.

Heterogeneous MPI Applications: Host and Coprocessor(s)

It is possible to have some processes executing on coprocessors and some on the host, as shown in Listing 2.61.

```
vega@lyra% mpiicpc -mmic -o ~/MPI-Hello MPI-Hello.cc
vega@lyra% mpiicpc -o ~/MPI-Hello MPI-Hello.cc
vega@lyra% cp MPI-Hello ~/
vega@lyra% mpirun \
> -host mic0 -np 2 ~/MPI-Hello.MIC : \
> -host mic1 -np 2 ~/MPI-Hello.MIC : \
> -host lyra -np 2 ~/MPI-Hello
Hello World from rank 5 running on lyra!
Hello World from rank 4 running on lyra!
Hello World from rank 2 running on lyra-mic1!
Hello World from rank 3 running on lyra-mic1!
Hello World from rank 1 running on lyra-mic0!
Hello World from rank 0 running on lyra-mic0!
MPI World size = 6 processes
```

Listing 2.61: Launching an Intel MPI application on two coprocessors and the host itself. The symbol '\' in the second line indicates the continuation of the shell command onto the next line.

Naturally, because the CPU and the coprocessor have different technical specifications and yield different performance, the programmer must ensure load balancing between the CPU and the MIC subsystems. One of the ways to do this at the machine file level is to specify different numbers of MPI processes for CPU hosts and MIC hosts (coprocessors). Whether to specify more or fewer ranks per node on the MIC to achieve load balance, depends on the workload partitioning algorithm and on the relative performance of the code on the CPU and MIC.

Machine File for Heterogeneous Applications

Instead of the long command line, a machine file can be used to launch heterogeneous MPI jobs. In this case, the obvious difficulty is only one executable name is passed to `mpirun`, while the executable for the CPU architecture must be stored in a different file from the executable for the MIC architecture. This difficulty can be overcome with one of the two methods.

Method 1 The programmer may compile the MIC executable and copy it to a directory in the coprocessor(s) filesystem, then compile the CPU executable with the same name, and copy it to the same path as the MIC executable, but in the host filesystem.

```
vega@lyra% # Compiling the MIC executable
vega@lyra% mpiicpc -mmic -o ~/MPI-Hello MPI-Hello.cc
vega@lyra% scp MPI-Hello mic0:~/
MPI-Hello              100%   12KB  12.4KB/s   00:00
vega@lyra% scp MPI-Hello mic1:~/
MPI-Hello              100%   12KB  12.4KB/s   00:00
vega@lyra% # Compiling the CPU executable with same name:
vega@lyra% mpiicpc -o ~/MPI-Hello MPI-Hello.cc
vega@lyra% cat mymachines.txt # machine file
mic0:2
mic1:2
lyra:2
vega@lyra% mpirun -machinefile mymachines.txt ~/MPI-Hello
Hello World from rank 5 running on lyra!
Hello World from rank 4 running on lyra!
Hello World from rank 2 running on lyra-mic1!
Hello World from rank 3 running on lyra-mic1!
Hello World from rank 1 running on lyra-mic0!
Hello World from rank 0 running on lyra-mic0!
MPI World size = 6 processes
```

Listing 2.62: Launching an Intel MPI application on two coprocessors and the host itself using a machine file and an executable with the same name and path.

It is impossible to use Method 1 if the executable is stored in a shared filesystem, because upon copying one of the executables, the other will be overwritten. In case of shared filesystem, the second method is more appropriate.

Method 2 Another way is to use the special Intel MPI environment variable `I_MPI_MIC_POSTFIX`. This variable instructs the Intel MPI library that the name and path of the MIC executable is the same as the name and path of the CPU executable with the addition of the postfix. This method is illustrated below.

```
vega@lyra% # Compiling MIC executable with postfix .MIC
vega@lyra% mpiicpc -o ~/MPI-Hello.MIC MPI-Hello.cc
vega@lyra% # Compiling CPU executable with no postfix:
vega@lyra% mpiicpc -o ~/MPI-Hello MPI-Hello.cc
vega@lyra% scp MPI-Hello.MIC mic0:~/
MPI-Hello.MIC        100%   12KB  12.4KB/s   00:00
vega@lyra% scp MPI-Hello.MIC mic1:~/
MPI-Hello.MIC        100%   12KB  12.4KB/s   00:00
vega@lyra% cp MPI-Hello ~/
vega@lyra% cat mymachines.txt # machine file
mic0:2
mic1:2
lyra:2
vega@lyra% export I_MPI_MIC_POSTFIX=.MIC
vega@lyra% mpirun -machinefile mymachines.txt ~/MPI-Hello
Hello World from rank 5 running on lyra!
Hello World from rank 4 running on lyra!
Hello World from rank 2 running on lyra-mic1!
Hello World from rank 3 running on lyra-mic1!
Hello World from rank 1 running on lyra-mic0!
Hello World from rank 0 running on lyra-mic0!
MPI World size = 6 processes
```

Listing 2.63: Launching an Intel MPI application on two coprocessors and the host itself using a machine file and and a MIC name postfix.

Note that an alternative variable, `I_MPI_MIC_PREFIX`, allows the programmer to place MIC and CPU executables at different paths.

2.5. Offload Programming with OpenMP 4.0

The new OpenMP 4.0 standard added support for offload to devices, providing an alternative method for offloading to Intel Xeon Phi coprocessors. Like the explicit offload, the offloading API in OpenMP 4.0 is based on pragmas. These pragmas are used to explicitly control data transfer and code offload to the coprocessor, allowing for manual data marshaling.

A selection of available pragmas are listed below. For full documentation refer to the Intel Intel C Compiler Compiler Reference or the OpenMP 4.0 documentation.

- `#pragma omp target`: The code inside the scope is executed on the target device. This pragma is the counterpart to `#pragma offload` in the explicit offload model. To also transfer data you must use the `map()` clause. For example, to transfer array `a` *to* the coprocessor but not *from* the coprocessor, use `map(to:a[0:N])`. Other arguments are `from` for transfer from the coprocessor, `tofrom` for both to and from the coprocessor, and `alloc` for just allocation with no transfer.

- `#pragma omp target data`: Transfers data to the coprocessor and implements data persistence. All `#pragma omp target` inside the scope of the pragma will have access to the variables mapped in `#pragma omp target data`. The syntax for map is the same as `#pragma omp target`, with transfers occurring at the start and the end of the region. If the data is modified on either the coprocessor or the host, they can be synchronized using `#pragma omp target update`.

- `#pragma omp target update`: Synchronizes data that is used in `#pragma omp target data`. For example to synchronize variable `a` and `b` to the value on the host, use `#pragma omp target data to(a,b)`. To synchronize to the value on the coprocessor, use `from(a,b)` clause instead.

- `#pragma omp declare target`: Marks a code region for compilation for MIC architecture. The region must be closed with `#pragma omp end declare target`. These pragmas are counterparts to `#pragma offload_attribute(push, target(mic))` and `#pragma offload_attribute(pop)` (see Section 2.2.2).

2.5.1. Offload with Pragma Target

Listing 2.64 demonstrates how to offload to the coprocessor using `#pragma omp target`. Note that because `omp target` is part of OpenMP standard, you must include `<omp.h>` and compile with `-qopenmp` flag.

```
#include <omp.h>
int main() {
  const int N = 1024;
  int A[N];
  for(int i = 0; i < N; i++)
    A[i] = i;

#pragma omp target map(tofrom:A) // Offloaded to the coprocessor
    for(int i = 0; i < N; i++)
      A[i] = -A[i]
}
```

Listing 2.64: Example of using `#pragma omp target`.

In the target offload we used `map(tofrom:A)` so that the array A is transferred *to* and *from* the coprocessor at the beginning and the end of the offload, respectively. Note that the constant `N` was automatically transferred.

Listing 2.65 shows an example of offloading to the coprocessor.

```
#include <omp.h>

#pragma omp declare target
void MyFunction() { //compiled for MIC architecture
  // ...
}
#pragma omp end declare target

// ...
#pragma omp target
    MyFunction() // Carried out on the coprocessor
// ...
```

Listing 2.65: Marking functions for offload with `#pragma omp declare target`.

2.5.2. Data Persistence with Pragma Target Data

The source code in Listing 2.66 demonstrates how to set up data retention using `#pragma omp target data`.

```
#include <stdio.h>
#include <omp.h>
#pragma omp declare target
void Increment(int *A, const int N) {
  for(int i = 0; i < N; i++)
    A[i]++;
}
#pragma omp end declare target

int main(int argv, char** argc){
  const int N = 1024;
  int A[N];
  for (int i = 0; i < N; i++)
    A[i] = i; //Initializing

  printf("A[0] = %d Bfore offload region.\n",A[0]);
#pragma omp target data map(tofrom:A)
  {
#pragma omp target
    Increment(A, N);

#pragma omp target update from(A) // Synchronization
    printf("A[0] = %d After update.\n",A[0]); //Runs on Host

#pragma omp target
    Increment(A, N);
  }
  printf("A[0] = %d After offload region.\n",A[0]);
}
```

```
vega@lyra% icpc -qopenmp omp-target-data.cc
vega@lyra% ./a.out
A[0] = 0 Before offload region.
A[0] = 1 After synchronization.
A[0] = 2 After offload region.
```

Listing 2.66: Data retention using `#pragma omp target data`.

In the code in Listing 2.66, `omp target data` is used to implement data retention for that Array A. Unlike explicit offload, array `A` does not have to be expressed in the `omp target` pragma that is within the scope of `omp target data`. In fact, if the `omp target` pragma has mapping for `A`, it will be ignored. The `tofrom` argument in the `map` clause means that array `A` is transferred at the beginning (`to`) and the end (`from`) of the region, instead of at every offload. To Synchronize the data between the host and the coprocessor inside the scope of `omp target data`, use `omp target update` pragma.

Note that `target data` creates a "data environment" within which data retention is implemented for offloads, but it *does not* offload code to the coprocessor on its own. Any code that does not have `#pragma omp target` will be executed on the host. For example, in the code in Listing 2.66, the `printf()` statement in line 23 is executed on the host.

CHAPTER 3 Expressing Parallelism

Chapter 2 discussed the ways in which application can move data and code between the host and Intel Xeon Phi coprocessors. Chapter 3 introduces parallel frameworks and programming language extensions supported by the Intel C++ Compiler for programming the Intel Xeon and Intel Xeon Phi architectures. It discusses data parallelism (vectorization), shared-memory thread parallelism (OpenMP, Intel Cilk Plus) and distributed-memory process parallelism with message passing (MPI). The purpose of this chapter is to introduce parallel programming paradigms and language constructs, rather than to provide optimization advice. For optimization, refer to Chapter 4.

3.1. Data Parallelism (Vectorization)

This section introduces data parallelism (vector instructions) in Intel Xeon processors and Intel Xeon Phi coprocessors and outlines the Intel C++ Compiler support for these instructions. Vector operations illustrated in this section can be used in both serial and multi-threaded applications, however, examples are limited to serial applications for simplicity. This section introduces the concept of vectorized calculations and language extensions for it; optimization practices for vectorization are discussed in Section 4.3.

3.1.1. Vector Instructions: Concept and History

Intel processor architectures today include data parallelism in the form of a vector instruction set. Vector instructions are designed to apply *the same* mathematical operation to *multiple* integer or floating-point numbers. Machines with support for vector instructions fall into the broader category of SIMD (Single Instruction Multiple Data) processors. In the context of Intel Xeon processors and Intel Xeon Phi coprocessors, the terms "vector" and "SIMD" instructions mean the same concept. The following *pseudocode* illustrates vector instructions:

Scalar Loop

```
For (i = 0; i < n; i++)
  A[i] += B[i];
```

Vector Loop

```
For (i = 0; i < n; i += 4)
  A[i:(i+4)] += B[i:(i+4)];
```

Listing 3.1: This *pseudocode* illustrates the concept of vector instructions. The vector loop (*right*) performs $1/4$ the number of iterations of the scalar loop (*left*), and each addition operator acts on 4 numbers at a time (i.e., addition here is a single instruction for multiple data elements).

The maximum potential speedup of this vectorized calculation with respect to a scalar version is equal to the number of values held in the processor's *vector registers*. In the example in Listing 3.1, this factor is equal to 4. The practical speedup with vectorization depends on the width of vector registers, type of scalar operands, type of instruction and associated memory traffic. See Section 3.1.2 for more information about vector instruction sets. Additional reading for code vectorization can be found in the book [14] by Aart Bik, former lead architect of automatic vectorization in the Intel compilers.

3.1.2. Intel Architecture Vector Instruction Sets

Vector instruction sets in modern processors typically include common arithmetic operations (addition, subtraction, multiplication and division), as well as comparisons, reduction and bit-masked operations. Table 3.1 summarizes the the instruction sets supported by Intel processors and coprocessors, along with supported types of variables and operations.

Instruction Set	Year & Intel Processor	Vector registers	Packed Data Types and Operations
MMX	1997, Pentium	64-bit	8-, 16- and 32-bit integers
SSE	1999, Pentium III	128-bit	32-bit single precision FP (floating-point)
SSE2	2001, Pentium 4	128-bit	8 to 64-bit integers; single & double prec. FP
SSE3–SSE4.2	2004 – 2009	128-bit	(additional instructions)
AVX	2011, Sandy Bridge	256-bit	single and double precision FP
AVX2	2013, Haswell	256-bit	FMA, integers, additional instructions
IMCI	2012, Knights Corner	512-bit	32-, 64-bit integers; single & double prec. FP
AVX-512	(future) Knights Landing	512-bit	32-, 64-bit integers; single & double prec. FP, additional instructions

Table 3.1: History of vector instruction sets supported by the Intel processors. Processors supporting modern instruction sets are backward-compatible with older instruction sets. The Intel Xeon Phi coprocessor is an exception to this trend, supporting only the IMCI ("Initial Many-Core Instructions") instruction set.

The common property of all these instruction sets is that they support data parallelism, i.e., the application of same arithmetic operation to a set of data elements (short vector). The number of elements in each vector is the ratio of the vector register width to the size of the data type. For instance, a 512-bit vector in the Knights Corner architecture (i.e., Intel Xeon Phi coprocessor) fits either eight double precision (64-bit) floating-point numbers, or sixteen single precision (32-bit) floating-point numbers. As vector instructions sets evolve, they support longer vectors, more data types, and greater diversity of instructions.

Data parallelism through vector instructions complements multi-core parallelism. Each core of an Intel Xeon processor or an Intel Xeon Phi coprocessor has its own vector processing facilities, and therefore different cores can be working on different vector instructions at any given moment (i.e., cores do not work in “lock-step”). The programmer shares the responsibility with the compiler to ensure that wherever the application can benefit from data parallelism, vector instructions are used. Furthermore, vectorization must exist in each thread of a multi-threaded program, which may pose additional challenges.

3.1.3. Is Your Code Using Vectorization?

Programmers have three options for employing vector instructions:

1. Some high-level mathematics libraries, such as Intel MKL, contain implementations of common operations for linear algebra, signal analysis, statistics, etc., which use vector instructions. In applications using such library functions, vectorization is employed without burdening the programmer. Whenever your application performs operations that can be expressed as an Intel MKL library function, the easiest way to vectorize this operation is to call the library implementation. This applies to workloads for Intel Xeon and Intel Xeon Phi architecture alike.

2. In high performance applications compiled from a high-level language (C/C++/Fortran), vector operations may be implemented by the compiler through a feature known as automatic vectorization. Automatic vectorization is enabled at the default optimization level `-O2`. However, in order to gain the most from automatic vectorization, the programmer must organize data and loops in a certain way, and/or use compiler hints such as `#pragma simd`, as described further in this section. Automatic vectorization is the most convenient way to employ vector instruction support, because cross-platform porting is performed by the compiler.

3. Finally, vector instructions may be called explicitly via inline assembly or vector intrinsics. This method may sometimes yield better performance than automatic vectorization, but cross-platform porting is difficult.

Even if you did not know about vector instructions before, or did not make specific efforts to employ them in your code, your application may already be using vector instructions if options 1 or 2 are your case. In this section, we focus on case 2 – automatic vectorization in user applications, and mention case 3.

3.1.4. Data Alignment

Before demonstrating how to utilize vector instructions in applications for Intel Xeon processors and Intel Xeon Phi coprocessors, let us digress into a very important related topic: data alignment. A prerequisite for successful use of vector instructions is placing the data to be processed at a memory address which allows for *aligned data access*. This typically means that the value of the memory address is a multiple of the vector register width in bytes. In some architectures, for instance, Intel Xeon processors supporting the AVX instruction set, unaligned memory accesses are permitted, but may be slower than aligned accesses. In other architectures, such as first generation Intel Xeon Phi coprocessors, unaligned memory accesses cause a segmentation fault.

The definition of data alignment is this: pointer `p` is said to address a memory location aligned on an `n`-byte boundary if `((size_t)p%n==0)`.

In Intel Xeon processors, 128-bit SSE instructions require 16-byte alignment. With 256-bit AVX, alignment requirements are relaxed, however, 32-byte alignment of data is recommended for performance optimization. In Intel Xeon Phi coprocessors with Knights Corner architecture and 512-bit IMCI, vector operations require 64-byte alignment.

In addition to serving vector operations, data alignment has other applications. For instance, for memory-intensive operations, including DMA used for CPU to coprocessor communication, it is beneficial to align data on a boundary equal to the virtual memory page size. Depending on the application, it means either 4 KiB, or 2 MiB alignment. Another possibility is that false sharing considerations (see Section 4.4.2) may require the programmer to align some data elements at the beginning of a cache line. In this case, 64-byte alignment is necessary (the same value for all modern Intel architectures).

Complete information on data alignment with the Intel C++ compiler can be found in the compiler reference.

Data Alignment on the Stack

For standalone scalar variables, Intel C and C++ compilers automatically implement natural alignment, i.e., the alignment boundary is equal to the size of the type.

```
float a; // 4-byte aligned
double b; // 8-byte aligned
```

Listing 3.2: Standalone scalar variables are automatically aligned on a natural boundary.

For array data on the stack, when alignment is necessary, array declaration must be accompanied by the attribute "aligned". This attribute is a C/C++ language extension supported by the Intel C and C++ compilers. Listing 3.3 demonstrates the usage of attribute "aligned" in Linux, and Listing 3.4 shows the syntax for Windows applications. See Intel C++ Compiler Reference for more details.

```
float A[n] __attribute__((aligned(64)));
```

Listing 3.3: Declaring an array on stack and aligning it on 64-byte boundary (Linux syntax).

```
__declspec(align(64)) float A[n];
```

Listing 3.4: Declaring an array on stack and aligning it on 64-byte boundary (Windows syntax).

In both examples shown above, array `A` will be placed in memory in such a way that the address of `A[0]` is a multiple of 64, i.e., aligned on a 64-byte boundary.

Alignment of Memory Blocks on the Heap

With the Intel C++ compiler, aligned arrays can be allocated/deallocated with the functions _mm_malloc and _mm_free, which replace the unaligned malloc and free calls. See usage example in Listing 3.5.

```
#include <malloc.h>
// ...
float *A = (float*)_mm_malloc(n*sizeof(float), 64);
// ...
_mm_free(A);
```

Listing 3.5: Allocating and deallocating a 64-byte aligned a memory block.

An alternative way to achieve alignment when allocating memory on the heap is to use the malloc call to allocate a block of memory slightly larger than needed, and then point a new pointer to the first aligned address within that block. The advantage of this method is that it can be used in compilers that do not support the _mm_malloc/_mm_free calls. See Listing 3.6 for an example of this procedure.

```
#include <cstdlib>
// ...
char *holder = (char*) malloc(bytes+64-1); // May be unaligned
size_t misalign = ((size_t)(*holder))%64; // How far from boundary
size_t offset = (misalign == 0 ? 0 : 64 - misalign); // Offset
float *A=(float*) ((char*)(holder) + offset); // 64-byte aligned
// ...
free(holder); // use original pointer to deallocate memory
```

Listing 3.6: Allocating and freeing a 64-byte aligned memory block. In this case, the pointer A should be used to access data, but memory must be free-d via holder.

Alignment of Objects Created with the Operator `new`

In C++, the operator `new` does not guarantee alignment of the memory block that it reserves. To align a C++ class object on a boundary, the programmer can allocate an aligned block of memory using one of the methods shown above, and then use the placement version of the operator `new` as shown in Listing 3.7. Naturally, if this method is used for objects of derived types (classes and structures), then the internal structure of these types must be designed in such a way that the data used for vector operations is aligned.

```
#include <new>
// ...
// Allocating memory block of sufficient size in
// such a way that buf[0] is aligned on a 64-byte boundary
void *buf = _mm_malloc(sizeof(MyClass), 64);

// placing MyClass into buf without allocating new memory
MyClass *ptr = new (buf) MyClass;
// ...

// Calling the destructor of MyClass
ptr->~MyClass();

// Deallocating aligned memory
_mm_free(buf);
```

Listing 3.7: Placing an object of type `MyClass` into a memory block aligned on a 64-byte boundary. Note that the `delete` operator should not be called on `ptr`; instead, the destructor should be run explicitly, followed by freeing the allocated memory block.

Alignment in Multidimensional Arrays

With arrays stored in row-major format, it is often desirable that every row begins on an aligned boundary. However, if the row length is not a multiple of the alignment value, then some rows may be aligned and some misaligned, as shown in a matrix-vector multiplication code in Listing 3.8.

```
const int n = 1000; // Note that 1000 is not a multiple of 16
float *A = (float*) _mm_malloc(sizeof(float)*n*n, 64);
float *x = (float*) _mm_malloc(sizeof(float)*n, 64);
float *b = (float*) _mm_malloc(sizeof(float)*n, 64);
//...
for (int i = 0; i < n; i++) {
  b[i] = 0.0f;
  // For i=0, A[i*n+0] is aligned - good
  // For i=1, A[i*n+0] is misaligned - bad!
  for (int j = 0; j < n; j++)
    b[i] += A[i*n + j]*x[j];
}
```

Listing 3.8: Computing matrix-vector product `Ax=b`.

To fix the situation, the programmer may pad the rows of `A` to a value that is a multiple of 16, as shown in Listing 3.9.

```
const int n = 1000; // Note that 1000 is not a multiple of 16
const int misalign = n%16;
const int nPad = n + (misalign == 0 ? 0 : 16 - misalign);
float *A = (float*) _mm_malloc(sizeof(float)*n*nPad, 64);
float *x = (float*) _mm_malloc(sizeof(float)*n, 64);
float *b = (float*) _mm_malloc(sizeof(float)*n, 64);
//...
for (int i = 0; i < n; i++) {
  b[i] = 0.0f;
  for (int j = 0; j < n; j++) // A[i*nPad+0] aligned for all i
    b[i] += A[i*nPad + j]*x[j];
}
```

Listing 3.9: Computing matrix-vector product `Ax=b`, row padding.

Alignment of Offloaded Arrays

In offload programming models, the programmer may need to ensure that arrays transferred from host to coprocessor are properly aligned on the target. This can be done using the clause `align` in the offload pragma as shown in Listing 3.10.

```
float *A = (float*) _mm_malloc(sizeof(float)*n, 4096);

#pragma offload target(mic) in(A : length(n) align(4096))
{
  // ...
}
```

Listing 3.10: Requesting non-default alignment for an offloaded array.

The default behavior of data movement by `#pragma offload` (without the `align` clause) is that the target memory address matches the offset of the host memory address within 64 bytes. For instance, if the array `A` was 64-byte aligned on the host, it would be 64-byte aligned on the coprocessor; and if it was offset by 16 bytes from a 64-byte boundary on the host, it would be offset by 16 bytes on the coprocessor as well.

However, in some applications, greater alignment values are necessary (for example, to place data at the beginning of a virtual memory page). In these cases, the clause `align` guarantees that the memory address of the array on the coprocessor is a multiple of the specified alignment value.

3.1.5. Vector Instructions using Inline Assembly, Compiler Intrinsics and Class Libraries

Vector instructions can be explicitly called from user code using assembly code, compiler intrinsics and class libraries. Note that these methods of using vector instructions are not recommended, as they limit the portability of the code across different architectures. For example, porting a code that runs on Intel Xeon processors and uses AVX intrinsics to Intel Xeon Phi coprocessors with IMCI intrinsics requires that the portion of code with intrinsics is completely re-written. Explicit vectorization is undesirable even if there is backward compatibility between instruction sets. For example, even though an AVX code may be run on the KNL architecture with the AVX-512 set, it will use only half of the available vector register width.

Instead of explicit vector instruction calls, developers are encouraged to employ automatic vectorization with methods described in Section 3.1.6 through Section 3.1.10. However, this section provides information about intrinsics for reference.

Inline Assembly Code

While inline assembly code is a very fine-grained method of using processor instructions, it is beyond the scope of this training, and interested readers can refer to the compiler documentation. For an in-depth review of Intel Xeon Phi coprocessor instruction set, refer to the Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual.

Intrinsics

Compiler intrinsics also provide low-level of control over operations while keeping the code more readable and giving the compiler freedom to re-order operations to optimize latency masking.

For every instruction set supported by the Intel compiler, there is a corresponding header file that declares the corresponding short vector types and vector functions. Table 3.2 lists these header files.

The usage of intrinsics on data stored in memory involves three steps:

1. Loading data into variables representing the content of vector registers;
2. Calling intrinsics on these vector register variables;
3. Storing the data in resultant vector register back into memory.

In addition, in some cases, the data loaded into vector registers must be aligned, i.e., placed at a memory address which is a multiple of a certain number of bytes. See Section 3.1.4 for more information on data alignment.

Instruction Set	Include header file
MMX	`mmintrin.h`
SSE	`xmmintrin.h` or `ia32intrin.h`
SSE2	`emmintrin.h` or `ia32intrin.h`
SSE3	`pmmintrin.h` or `ia32intrin.h`
SSSE3	`tmmintrin.h` or `ia32intrin.h`
SSE4	`smmintrin.h` and `nmmintrin.h`
AVX, AVX2, IMCI, AVX-512	`immintrin.h`

Table 3.2: Header files for the Intel C++ Compiler intrinsics. See also Table 3.1.

Codes in Listing 3.11 illustrate using AVX and IMCI intrinsics for the addition of two arrays shown in Listing 3.1. Note that the stride of the loop variable `i` is 8 for the AVX code and 16 for the IMCI code.

```
for (int i=0; i<n; i+=8) {
__m256 Avec=_mm256_load_ps(A+i);
__m256 Bvec=_mm256_load_ps(B+i);
Avec=_mm256_add_ps(Avec, Bvec);
_mm256_store_ps(A+i, Avec);
}
```

```
for (int i=0; i<n; i+=16) {
__m512 Avec=_mm512_load_ps(A+i);
__m512 Bvec=_mm512_load_ps(B+i);
Avec=_mm512_add_ps(Avec, Bvec);
_mm512_store_ps(A+i, Avec);
}
```

Listing 3.11: Addition of two arrays using AVX intrinsics (*left*) and IMCI intrinsics (*right*). These codes assume that the arrays `float A[n]` and `float B[n]` are aligned on a 64-byte boundary, and that `n` is a multiple of 8 for AVX and a multiple of 16 for IMCI. Variables `Avec` and `Bvec` are 256 = 8 × `sizeof(float)` bits in size for AVX and 512 = 16 × `sizeof(float)` bits for the Intel Xeon Phi architecture.

The AVX code in Listing 3.11 will run only on Intel Xeon processors, and the IMCI code will run only on Intel Xeon Phi coprocessors. The necessity to maintain a separate version of vectorized code for each target instruction set is generally undesirable, however, it cannot be avoided when code is vectorized with intrinsics. A better approach to expressing data parallelism is using the Intel Cilk Plus extensions for array notation (see Section 3.1.7) or automatically vectorizable C loops and vectorization pragmas (see Section 3.1.6 through Section 3.1.10).

Note that switching between different instruction sets in a code employing intrinsics should be done with care. In some cases, in order to switch between different instruction sets supported by a processor, register have to be set to a certain state to avoid a performance penalty. See the Intel C++ Compiler Reference [15] and Intel Intrinsics Guide [16] for details.

Class Libraries

The C++ vector class library provided by the Intel Compilers defines short vectors as C++ classes, and operators acting on these vectors are implemented with vector instructions. A similar library is maintained by Agner Fog. Table 3.3 lists the header files that should be included in order to gain access to the Intel C++ Class Library.

Instruction Set	Include header file
MMX	`ivec.h`
SSE	`fvec.h`
SSE2, AVX	`dvec.h`
IMCI	`micvec.h`

Table 3.3: Header files for the Intel C++ Class Library.

Codes in Listing 3.12 demonstrate how the C++ vector class library included with the Intel C++ compiler can be used to execute the vector loop shown in Listing 3.1.

```
for (int i=0; i<n; i+=4) {
F32vec4 *Avec=(F32vec4*)(A+i);
F32vec4 *Bvec=(F32vec4*)(B+i);
*Avec = *Avec + *Bvec;
}
```

```
for (int i=0; i<n; i+=8) {
F64vec8 *Avec=(F64vec8*)(A+i);
F64vec8 *Bvec=(F64vec8*)(B+i);
*Avec = *Avec + *Bvec;
}
```

Listing 3.12: Addition of two double precision arrays using the Intel C++ vector class library with AVX (*left*) and IMCI instructions (*right*). These codes assume that the arrays `float A[n]` and `float B[n]` are aligned on a 64-byte boundary, and that `n` is a multiple of 4 for AVX and a multiple of 8 for the Intel Xeon Phi architecture.

3.1.6. Automatic Vectorization of Loops

To take advantage of vector instructions, the developer does not need to call them explicitly via inline assembly or intrinsics. The alternative route is using the automatic vectorization feature of the Intel compiler. This function transforms scalar C/C++ loops into loops with short vectors and vector instructions during code compilation. The benefits of using automatic vectorization instead of intrinsic functions or vector class libraries are:

1) code porting to a new architecture is possible by re-compilation;
2) ease of development and improved code readability, and
3) (in some cases) heuristic compile-time analysis and empirical run-time analysis of the profitability of various vectorization paths implemented by the compiler.

A practical example of automatic vectorization the loop from Listing 3.1 is shown in Listing 3.13. This example, while trivial, demonstrates some of the important aspects of automatic vectorization.

```
#include <cstdio>
int main(){
  const int n=8;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));
  int C[n] __attribute__((aligned(64)));

  // Initialization. Will be auto-vectorized
  for (int i = 0; i < n; i++)
    A[i] = B[i] = i;

  // This loop will be auto-vectorized
  for (int i = 0; i < n; i++)
    C[i] = A[i] + B[i];

  // Output
  for (int i = 0; i < n; i++)
    printf("Element #%2d: %2d + %2d = %2d (%s)\n",
           i, A[i], B[i], C[i], (C[i]==A[i]+B[i]?"OK":"ERROR"));
}
```

Listing 3.13: The code `Vec-Addition.cc` gets auto-vectorized by the compiler.

Unlike codes in Listing 3.11 and Listing 3.12, the code in Listing 3.13 is oblivious of the architecture that it is compiled for. Indeed, this code can be compiled and auto-vectorized for Pentium 4 processors with SSE2 instructions as well as for Intel Xeon Phi coprocessors. The only place where architecture is implicitly assumed is the alignment boundary value of 64. This value is chosen to satisfy the IMCI alignment requirements.

```
vega@lyra% CCFLAGS="-qopt-report=2 -qopt-report-phase:vec"
vega@lyra% icpc ${CCFLAGS} -o Vec-Addition Vec-Addition.cc
icpc: remark #10397: optimization reports are generated in
*.optrpt files in the output location
vega@lyra% ./Vec-Addition
Element # 0:   0 +   0 =   0 (OK)
Element # 1:   1 +   1 =   2 (OK)
Element # 2:   2 +   2 =   4 (OK)
Element # 3:   3 +   3 =   6 (OK)
Element # 4:   4 +   4 =   8 (OK)
Element # 5:   5 +   5 =  10 (OK)
Element # 6:   6 +   6 =  12 (OK)
Element # 7:   7 +   7 =  14 (OK)
```

Listing 3.14: Compiling and running `Vec-Addition.cc` with optimization report generation for the vectorization phase.

The output of the code is predictable: the result of addition of two integers is calculated correctly. However, as we will soon see from the optimization report, the addition loop in line 13 was vectorized and performed with AVX vector instructions.

The compiler arguments that we used in Listing 3.14 are diagnostic. The argument `-qopt-report` sets the verbosity of diagnostic output, and `-qopt-report-phase:vec` tells the compiler to only generate vectorization report. The report goes into the text file `Vec-Addition.optrpt`. It is also possible to direct the optimization report to the screen by using the argument `-qopt-report-stdout`.

The arguments discussed above pertain to Intel C++ compiler version 15.0 and above. In older compilers, the argument that produces diagnostic report is `-vec-report=n` (deprecated in 15.0).

Now let's take a look at the optimization report (see Listing 3.15).

```
vega@lyra% cat Vec-Addition.optrpt
Begin optimization report for: main()
    Report from: Vector optimizations [vec]

LOOP BEGIN at Vec-Addition.cc(9,3)
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at Vec-Addition.cc(13,3)
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at Vec-Addition.cc(17,3)
   remark #15344: loop was not vectorized: vector dependence
   prevents vectorization. First dependence is shown below. Use
   level 5 report for details
LOOP END
```

Listing 3.15: Automatic vectorization report produced by the compiler.

The message "LOOP WAS VECTORIZED" indicates successful compilation with vector instructions. Two loops were vectorized here: initialization in line 9 and addition in line 13. Remarks such as "loop was not vectorized" are also helpful. They may be telling the programmer that something needs to be changed in order for vectorization to succeed (the rest of this section discusses these cases). However, in our case, loop in line 17 was not vectorized because it contains an I/O operation, which, naturally, cannot be performed with vector instructions.

Note that no special optimization arguments were used in order for vectorization to occur automatically. Automatic vectorization is enabled for the default optimization level `-O2` and higher levels.

As proof of that this C code is indeed a portable solution, one can compile it for native execution on Intel Xeon Phi coprocessors. Listing 3.16 illustrates the result.

```
vega@lyra% icpc ${CCFLAGS} -mmic -o Vec-Addition Vec-Addition.cc
icpc: remark #10397: optimization reports are generated in
*.optrpt files in the output location

vega@lyra% micnativeloadex ./Vec-Addition # run on Xeon Phi
Element # 0:  0 +  0 =  0 (OK)
Element # 1:  1 +  1 =  2 (OK)
Element # 2:  2 +  2 =  4 (OK)
Element # 3:  3 +  3 =  6 (OK)
Element # 4:  4 +  4 =  8 (OK)
Element # 5:  5 +  5 = 10 (OK)
Element # 6:  6 +  6 = 12 (OK)
Element # 7:  7 +  7 = 14 (OK)

vega@lyra% cat Vec-Addition.optrpt
Begin optimization report for: main()
    Report from: Vector optimizations [vec]

LOOP BEGIN at Vec-Addition.cc(9,3)
   remark #15335: loop was not vectorized: vectorization possible
   but seems inefficient. Use vector always directive or
   -vec-threshold0 to override
LOOP END

LOOP BEGIN at Vec-Addition.cc(13,3)
   remark #15300: LOOP WAS VECTORIZED   [ Vec-Addition.cc(13,3) ]
LOOP END

LOOP BEGIN at Vec-Addition.cc(17,3)
   remark #15344: loop was not vectorized: vector dependence
   prevents vectorization. First dependence is shown below. Use
   level 5 report for details
LOOP END
```

Listing 3.16: Compilation for MIC architecture and runtime output of the code in Listing 3.13

It is worth noting the compiler's remark on the initialization loop in line 9. On the MIC architecture, the compiler reports "vectorization possible but seems inefficient". This is likely because Knights Corner vectors hold up to 16 single precision floating-point values, and our loop is only 8 iterations long. This is an example of the compiler's heuristic analysis of the efficiency of vectorization.

Indeed, in addition to portability across architectures, reliance on automatic vectorization provides other benefits. For instance, auto-vectorizable code may release the programmer of the requirement that the number of iterations should be a multiple of the number of data elements in the vector register. If loop length is not a multiple of the vector length, the compiler will peel off the first or last few iterations and perform them with scalar instructions. It is also possible to automatically vectorize loops working with data that are not aligned on a proper boundary. In this case, the compiler will generate code to check the data alignment at runtime and, if necessary, peel off a few iterations at the start of the loop in order to perform the bulk of the calculations with fast aligned instructions. See Section 4.3.3 for a more thorough discussion of this topic.

Generally, the only type of loops that the compiler will auto-vectorize is the for-loop, with the number of iterations run known at runtime, or, better yet, at compile time. Memory access in the loop must have regular pattern, ideally with unit stride.

Non-standard loops that cannot be automatically vectorized include: loops with irregular memory access patterns, calculations with vector dependence, while-loops or for-loops in which the number of iterations cannot be determined at the start of the loop, outer loops, loops with complex branches (i.e., if-conditions), and anything else that cannot be, or is very difficult to vectorize.

Further information on automatic vectorization of loops can be found in Section 3.1.10 and Section 4.3, and in the Intel C++ compiler reference.

3.1.7. Extensions for Array Notation in Intel Cilk Plus

Intel C++ compiler supports the Intel Cilk Plus framework as an extension to the C++ language. Intel Cilk Plus, among other things, introduces array notation, which is a method for specifying slices of arrays or whole arrays, and applying element-wise operations to arrays of the same shape. The Intel C++ Compiler can auto-vectorize operations expressed in array notation.

In the example code in Listing 3.13, the addition loop in lines 14-15 can be replaced with the code shown in Listing 3.17. When this code is compiled with the Intel C++ Compiler, the addition operation will be automatically vectorized. This example assumes that array bounds are known at compile time.

```
A[:] += B[:];
```

Listing 3.17: Intel Cilk Plus extensions for array notation: to all elements of array A, add elements of array B.

It is also possible to specify slices of arrays:

```
A[0:16] += B[32:16];
```

Listing 3.18: To sixteen consecutive elements of array `A` (0 through 15) add sixteen consecutive elements of array `B` (32 through 47).

And it is possible to indicate a stride:

```
A[0:16:2] += B[32:16:4];
```

Listing 3.19: To every sixteen elements of array `A` with a stride of 2 (elements 0, 2, 4, . . . , 30) add sixteen elements of array `B` with a stride of 4 (elements 32, 36, 38, . . . 92).

Intel Cilk Plus extensions in the Intel C++ compiler are enabled by default, and therefore no additional modifications of the code or compiler arguments are necessary. However, in order to enable compilation with non-Intel compilers, the programmer must protect the expressions with array notation with preprocessor directives and provide an alternative implementation of these expressions with loops that can be understood by other compilers. See Listing 3.20 for an example.

```
#ifdef __INTEL_COMPILER
  A[:] += B[:]
#else
  for (int i = 0; i < 16; i++)
    A[i] += B[i];
#endif
```

Listing 3.20: Protecting Intel Cilk Plus array notation in order to enable compilation with non-Intel compilers.

Array notation extensions also work with multidimensional arrays. Refer to the Intel C++ Compiler documentation for more details on Intel Cilk Plus and the array notation extensions of this library.

3.1.8. SIMD-Enabled Functions

SIMD-enabled functions (formerly known as elemental functions) are an additional method available with the Intel C++ Compiler for facilitation of automatic code vectorization. They are written as a regular C/C++ functions, operating on only scalar numbers with scalar syntax. In the code, SIMD-enabled functions can be called to operate in a data-parallel context, and the compiler will automatically implement vectorization where possible. They can also be used in data- and thread-parallel contexts with automatic parallelization across multiple threads.

SIMD-Enabled functions are marked up in the code with the vector attribute (see below), which indicates to the compiler that a vector version of this function must be implemented. Not every function can be marked as SIMD-enabled. Operations in SIMD-enabled functions must be vectorizable, and these functions must not modify any data outside of their scope.

Listing 3.21 demonstrates the addition of two arrays where the addition operation is executed in a regular (not SIMD-enabled) function.

```c
float MySimpleAdd(float x1, float x2){
    return x1 + x2;
}

// ...in a separate source file:
for (int i = 0; i < N, i++)
  output[i] = MySimpleAdd(inputa[i], inputb[i]);
```

Listing 3.21: Scalar function for addition in C.

If the code of the function and the call to the function are located in the same file, the compiler may perform inter-procedural optimization (inline the function) and vectorize this loop. However, what if the function is a part of a library, and the loop is located in a separate file that uses the library? That would make it impossible for the compiler to inline the function and replace scalar addition with vector operations. The solution to this situation is offered by SIMD-enabled functions. To declare `my_simple_add` as SIMD-enabled, the function declaration must contain the vector attribute. And in order to force the vectorization of the loop using this function, `#pragma simd` may need to be used. Listing 3.22 demonstrates this method.

```
__attribute__((vector)) float MySimpleAdd(float x1, float x2) {
    return x1 + x2;
}
#pragma simd
for (int i = 0; i < N, ++i)
    output[i] = MySimpleAdd(inputa[i], inputb[i]);
```

Listing 3.22: Vectorized function for addition in Intel Cilk Plus.

The usage of SIMD-enabled functions may be combined with array notation as shown in Listing 3.23.

```
__attribute__((vector)) float MySimpleAdd(float x1, float x2){
    return x1 + x2;
}
MySimpleAdd(inputa[:], inputb[:]);
```

Listing 3.23: Vectorized function for addition in Intel Cilk Plus.

The vector attribute can be supplied with clauses that hint to the compiler how arguments are used inside the function, and how the function is used in loops. Listing 3.24 shows an example. See the comment in the code for details.

```
// ...vectorize LegendrePolynomial with respect to x,
//    but assume n is constant across all vector lanes (uniform)
__attribute__((vector(uniform(n))))
  float LegendrePolynomial(const int n, const float x);

C[:] = LegendrePolynomial(1, X[:]); // will be vectorized
D[:] = LegendrePolynomial(N[:], 0.5f); // will not be vectorized
```

Listing 3.24: Example of using clauses of the vector attribute.

For more information on SIMD-enabled functions in Intel Cilk Plus, refer to the Intel C++ Compiler compiler documentation. For an example of usage of SIMD-enabled functions in a library, see the article [17].

3.1.9. Assumed Vector Dependence

True vector dependence, such as in the code in Listing 3.25, makes it impossible to vectorize loops manually or automatically. In this code, `a[i-1]` must be known before `a[i]` can be calculated, so the algorithm must be executed serially.

```
float* a;
// ...
for (int i = 1; i < n; i++)
  a[i] += b[i]*a[i-1];
```

Listing 3.25: Vector dependence makes the vectorization of this loop impossible.

However, in some cases the compiler may not have sufficient information to determine whether true vector dependence is present in the loop. Such cases are referred to as *assumed vector dependence*.

Assumed Vector Dependence Example

Code in Listing 3.26 shows a case where it is impossible to determine whether vector dependence exists. If pointers `a` and `b` point to distinct, non-overlapping memory segments, then there is no vector dependence. However, there is a possibility that the user will pass to the function `a` and `b` pointing to overlapping memory addresses (e.g., `a=b+1`), in which case vector dependence will exist. The Intel C++ Compiler may refuse to vectorize this loop due to assumed vector dependence.

```
void MyCopy(int n, float* a, float* b) {
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

Listing 3.26: `Vec-Assumed.txt` illustrates assumed vector dependence situation.

The report in Listing 3.27 shows that the loop is not vectorized. The reason for that is assumed vector dependence.

```
vega@lyra% icpc -qopt-report=2 -qopt-report-phase:vec \
> -qopt-report-stdout -c Vec-Assumed.cc
...
   LOOP BEGIN at Vec-Assumed.cc(2,3)
   <Multiversioned v2>
      remark #15304: loop was not vectorized: non-vectorizable loop
                     instance from multiversioning
   LOOP END
...
```

Listing 3.27: Assumed vector dependence prevents loop vectorization.

Ignoring Assumed Vector Dependence

In cases when the developer knows that there will not be a true vector dependence situation, it is possible to instruct the compiler to ignore assumed vector dependencies found in a loop. This can be done with `#pragma ivdep`, as shown in Listing 3.28.

```
void MyCopy(int n, float* a, float* b) {
#pragma ivdep
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

```
vega@lyra% icpc -qopt-report=2 -qopt-report-phase:vec \
> -qopt-report-stdout -c Vec-ivdep.cc
...
   LOOP BEGIN at Vec-Assumed.cc(3,3)
   <Multiversioned v2>
      remark #15300: LOOP WAS VECTORIZED
   LOOP END
...
```

Listing 3.28: *Top*: `Vec-ivdep.cc` contains a pragma that instructs the compiler to ignore assumed vector dependence. *Bottom:* automatic vectorization succeeds thanks to `#pragma ivdep`.

If the function compiled in this way is called with overlapping arrays `a` and `b` (i.e., with true vector dependence), a runtime exception will occur (i.e., the code will crash).

Pointer Disambiguation

A more fine-grained method to disambiguate the possibility of vector dependence is the `restrict` keyword. This keyword applies to each pointer variable qualified with it, and instructs the compiler that the object accessed by the pointer is only accessed by that pointer in the given scope (i.e., that this pointer is not aliased). To enable the keyword `restrict`, the compiler argument `-restrict` must be used. An example of the usage of keyword `restrict` is shown in Listing 3.29. This time, automatic vectorization has succeeded as well. Note that the compiler was given the argument `-restrict`, without which compilation would have failed.

```
void MyCopy(int n, float* restrict a, float* restrict b) {
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

```
vega@lyra% icpc -qopt-report=2 -qopt-report-phase:vec \
> -qopt-report-stdout -restrict -c Vec-Restrict.cc
...
   LOOP BEGIN at Vec-Assumed.cc(2,3)
      remark #15300: LOOP WAS VECTORIZED
   LOOP END
```

Listing 3.29: *Top*: `Vec-Restrict.cc` uses the `restrict` keyword to disambiguate pointers. *Bottom*: Automatic vectorization succeeds.

Codes in which automatic vectorization fails often prove to be cases of assumed vector dependence. However, in some cases, vectorization may fail because the compiler incorrectly performs heuristic performance analysis, or due to explicit or implicitly inlined inner loops. In these cases, additional vectorization pragmas and compiler arguments may help, as discussed in the next section.

3.1.10. Vectorization Pragmas, Keywords and Compiler Arguments.

The following list contains some compiler pragmas that may be useful for tuning vectorized code performance. Details can be found in Intel C++ compiler reference. In the PDF version of this document, the items in the list below are hyperlinks pointing to the corresponding articles in the compiler reference.

- `#pragma simd`
  Used to guide the compiler to automatically vectorize more loops. Arguments of this pragma can guide the compiler in cases when automatic vectorization is difficult. Specifically, it is useful for vectorizing outer loops (see Section 4.5.2 for an example) and loops with SIMD-enabled functions (see Section 3.1.8).

- `#pragma vector always`
  Instructs the compiler to implement automatic vectorization of the loop following this pragma, even if heuristic analysis suggests otherwise, or if non-unit stride or unaligned accesses make vectorization inefficient.

- `#pragma vector aligned | unaligned`
  Instructs the compiler to always use aligned or unaligned data movement instructions. Useful, for instance, when the developer guarantees data alignment. In this case, placing `#pragma vector aligned` before the loop eliminates unnecessary run-time checks for data alignment, which improves performance. See Section 4.3.4 for an example.

- `__assume_aligned` keyword
  Helps to eliminate runtime alignment checks when data is guaranteed to be properly aligned. This keyword produces an effect similar to that of `#pragma vector aligned`, but provides more granular control, as `__assume_aligned` applies to an individual array that participates in the calculation, and not to the whole loop.

- `#pragma vector nontemporal | temporal`
  Instructs the compiler to use non-temporal (i.e., streaming) or temporal (i.e., non-streaming) stores. Non-temporal stores can be useful when

the result of a vector operation is not used down the line in the automatically vectorized loop. In this case, placing `#pragma vector nontemporal` will force the code to send the result of each vector instruction directly to RAM instead of placing it in cache. This may improve performance, as the data sent to RAM will not contaminate the valuable processor cache and leave more room for frequently re-used data. At the same time, if the written data is to be subsequently re-used, non-temporal stores will degrade performance. See [18] for an example of usage.

- `#pragma novector`
  Instructs the compiler not to vectorize the loop following this pragma. Can be used for convenience: the developer may place this pragma before a non-vectorizable loop in order to keep the vectorization report cleaner. In some cases, `#pragma novector` can improve performance. For example, if a loop contains a calculation-heavy branch that is rarely taken. If such a loop is auto-vectorized, the branch will be always taken, and iterations in which the branch should not have been taken will be masked out from the result. However, `#pragma novector` can turn such loop into a scalar loop, in which only the taken branches are evaluated.

- `#pragma ivdep`
  Instructs the compiler to ignore vector dependence, which increases the likelihood of automatic loop vectorization. See Section 3.1.9 for more information and Section 4.3.5 for an example. The keyword `restrict` can often help to achieve a similar result.

- `restrict` qualifier and `-restrict` command-line argument
  This keyword qualifies a pointer as restricted, i.e., the developer guarantees to the compiler that in the scope of this pointer's visibility, no other pointer refers to the data referenced by the restricted pointer. Qualifying function arguments with the `restrict` keyword helps in the elimination of assumed vector dependencies. The `restrict` keyword in the code must be enabled by the `-restrict` compiler argument. Keyword `restrict` achieves the same effect as `#pragma vector ivdep`, but allows more fine-grained control, because it

applies to a single pointer rather than the entire loop. See Section 3.1.9 for more detail.

- `#pragma loop count`
  Informs the compiler of the number of loop iterations anticipated at runtime. This helps the auto-vectorizer to make more accurate predictions regarding the optimal vectorization strategy.

- `-qopt-report=[n]` `-qopt-report-phase:vec`
  These compiler arguments indicate the level of verbosity of the automatic vectorizer. It replaces the argument `-vec-report[=n]`, which was used in earlier versions of Intel compilers. `n=5` provides the most verbose report including vectorized and non-vectorized loops and any proven or assumed data dependencies.

- `-O[n]`
  Optimization level, defaults to `-O2`. Automatic vectorization is enabled with `-O2` and higher optimization levels.

- `-x[code]`
  Instructs the compiler to target specific processor features, including instruction sets and optimizations. For example, to generate AVX code, `-xAVX` can be used; for SSE2, `-xSSE2`. Using `-xhost` targets the architecture found in the system that performs the calculation. See Section 4.2.1 for more information.

3.1.11. Exclusive Features of the IMCI Instruction Set

When the Intel Xeon Phi architecture is operated in tandem with Intel compilers, they allow to achieve high performance without low-level optimizations, intrinsics or assembly code in user applications. Indeed, automatic vectorization and support for parallel libraries make it possible to write a single code that runs efficiently on both Intel Xeon processors and Intel Xeon Phi coprocessors. This code is also able to scale to future generations of the MIC platforms, such as the Knights Landing architecture. For the programmer that takes advantage of this portability feature by relying on automatic vectorization, it is not necessary to know every detail of the instruction set. However, understanding the type of instructions that the compiler uses to automatically vectorize C, C++ or Fortran codes allows the programmer to design algorithms and data structures in a way that is most efficient for automatic vectorization.

Intel Initial Many Core Instructions (Intel IMCI) is the instruction set supported by Intel Xeon Phi coprocessors. Intel IMCI can be considered superior to the SSE* and AVX* instructions supported by Intel Xeon processors, however, it is important to realize that Intel Xeon Phi coprocessors do not directly support SSE or AVX instructions.

The instructions of Intel IMCI operate on special 512-bit registers, which can pack up to eight 64-bit elements (long integers or double precision floating-point numbers) or up to sixteen 32-bit elements (integers or single precision floating-point numbers). For use with intrinsic functions, these registers are represented by three data types declared in the header file `immintrin.h`: `__mm512` (single precision floating-point vector), `__mm512i` (a 32-bit integer vector, or, for a limited set of instructions, a 64-bit integer vector) and `__mm512d` (double precision floating-point vector). Most instructions operate on three arguments: either two source registers with a separate destination register, or three source registers, one of which is also a destination.

For each operation, two types of instructions are available: unmasked and masked. Unmasked instructions apply the requested operation to all elements of the vector registers. Masked instructions apply the operation to some of the elements and preserve the value of other elements in the output register. The set of elements that must be modified in the output registers is controlled by an additional argument of type `__mmask16` or `__mmask8`. This is a short integer value, in which bits set to 1 or 0 indicate that the corresponding output elements should be modified or preserved by the masked operation using this bitmask.

The classes of available IMCI instructions are outlined in the list below, illustrated with calls to the respective intrinsic functions.

Initialization instructions are used to fill a 512-bit vector register with one or multiple values of scalar elements. Example:

```
__mm512 myvec = _mm512_set1_ps(3.14f);
```

The above example creates a 512-bit short vector of sixteen SP floating-point numbers and initializes all sixteen elements to a value of 3.14f;

Load and store instructions copy a contiguous 512-bits chunk of data from a memory location to the vector register (load) or from the vector register to a memory location (store). The address from/to which the copying takes place must be 64-byte aligned. Additional versions of these instructions operate only on the high or low 256 bits of the vector. Example:

```
float myarr[128] __attribute__((align(64)));
myarr[:] = 1.0f;
__mm512 myvec = _mm512_load_ps(&myarr[32]);
```

In this example, elements 32 through 47 of array `myarr` are loaded into the vector register assigned to variable `myvec`.

Gather and scatter instructions are used to copy non-contiguous data from memory to vector registers (gather), or from vector registers to memory (scatter). The memory access pattern must have a power of 2 stride (1, 2, 4, 8, . . . elements). The copying of data can be done simultaneously with type conversion. It is also possible to specify prefetching from memory to cache for this type of operation. Example:

```
__mm512i myvec = _mm512_set1_epi32(-1);
float myarr[128] __attribute__(align(64));
_mm512_i32scatter_ps(myvec, &myarr[0], 4);
```

The above code scatters the values in integer short vector `myvec` to array `myarr` starting with the index 0 with a stride of 4. That is, elements 0, 1, 2, . . . , 15 of the short vector `myvec` will be copied to array elements `myvec[0]`, `myvec[4]`, `myvec[8]`, . . ., `myvec[60]`, respectively.

Arithmetic instructions are the core of high performance calculations. The list below illustrates the scope of these instructions.

a) Addition, subtraction and multiplication are available for all data types supported in IMCI. It is possible to specify the rounding method for floating-point operations. Example:

```
__mm512 c = _mm512_mul_ps(a, b);
```

b) FMA instruction is the basis of several operations in linear algebra, including xAXPY and dot-product calculations. These instructions perform element-wise multiplication of vectors `v1` and `v2` and add the result to vector `v3`. The FMA instruction is supported by the Intel Xeon Phi architecture and by generation 3 of Intel Xeon processors (Haswell architecture). The latency and throughput of FMA is comparable to that of individual addition or of individual multiplication instruction, and therefore it is always preferable to use FMA instead of separate addition and multiplication where possible. It is possible to specify the rounding method for floating-point operations. Example:

```
__mm512 r = _mm512_fmadd_ps(v1, v2, v3);
```

This expression computes the result of `v1*v2+v3` and places it in `v3`. This syntax of the FMA instruction calculates intermediate values to infinite precision.

c) Division and transcendental function implementations are available in the Intel Short Vector Math Library (SVML). Some of these software functions use hardware-implemented transcendentals in the MIC architecture. For lowest-precision implementations, an SVML function may just be reduced to calling a transcendental intrinsic; for higher-precision implementations, SVML may first call the transcendental intrinsic and then iterate on the obtained value in software to refine it. The following transcendental operations are supported through SVML intrinsics:

- Division and reciprocal calculation;
- Error function;
- Inverse error function;
- Exponential functions (natural, base 2 and 10) and power function. Base 2 exponential is the fastest implementation;
- Logarithms (natural, base 2 and base 10). Base 2 logarithm is the fastest implementation;
- Square and cubic root, reciprocal square root, hypotenuse;
- Trigonometric functions (`sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos`, `atan`);
- Rounding functions.

The following example calculates the reciprocal square root of each element of vector `y`:

```
__mm512 x = _m512_invsqrt_ps(y);
```

Swizzle and permute instructions rearrange (shuffle) scalar elements in a vector register. For these operations it is convenient to think of a 512-bit register as a set of four 128-bit blocks. The **swizzle** operation rearranges elements within each 128-bit block, and the **permute** operation rearranges the 128-bit blocks in the register according to the pattern specified by the user. These instructions can be used in combination with another intrinsic, which saves processor cycles. Example:

```
__mm512 myv1, myv2;
// ...
__mm512_add_ps(myv1,
        __mm512_swizzle_ps(myv2, _MM_SWIZ_REG_DCAB));
```

In this example, the swizzle operation with the pattern `DCAB` is applied to the 512-bit SP floating-point vector `myv2`, and then this vector, swizzled, is added to another vector of the same type, `myv1`.

Comparison instructions perform element-wise comparison between two 512-bit vectors and return a bit-mask value with bits set to 0 or 1 depending on the result of the respective comparison. Example:

```
__mmask16 result =
              _m512_cmp_ps_mask(x, y, _MM_CMPINT_LT);
```

The above code compares vectors `x` and `y` and returns the bitmask `result` where bits are set to 1 if the corresponding element in `x` is less than the corresponding element in `y`.

Conversion and type cast instructions perform conversion from single to double precision and from double to single precision floating-point numbers, from floating-point numbers to integers and from integers to floating-point numbers.

Bitwise instructions perform bit-wise AND, OR, XAND and XOR operations on elements in 512-bit short vectors.

Reduction and minimum/maximum instructions allow the calculation of the sum of all elements in a vector, the product of all elements in a vector, and the evaluation of the minimum or maximum of all elements in a vector. These instructions are exclusive in the Intel IMCI.

Vector mask instructions allow to set the values of type `__mmask16` and `__mmask8` and to perform bitwise operations on them. Masks can be used in all IMCI instructions to control which of the elements in the resulting vector are modified, and which are preserved in an operation. Bitmasked operations are an exclusive feature of IMCI.

Miscellaneous instructions are available for decomposing floating-point numbers into the mantissa and the exponent, fixing NaNs and performing low-precision transcendental operations;

Scalar instructions are available for bit counting, cache eviction and thread delay.

3.2. Task Parallelism in Shared Memory: OpenMP

The contents of this section are focused on multi-threaded programming (i.e., task parallelism). This section introduces the API and programming paradigms of multi-threaded codes. For advice on optimization in shared-memory parallel applications, refer to Section 4.4.

3.2.1. Multiple Cores and Task Parallelism

Task parallelism in shared memory is a method of distributing work in an application across multiple instruction sequences called threads, which run within the same computing system and share the virtual memory address space. The operating system and the runtime library for multi-threading distribute threads across the multiple cores of a CPU or a coprocessor.

Unlike with vectorization, work partitioning with threads does not have to have the SIMD syntax, i.e., different threads may have different programs operating on different data sets. At the same time, it is possible to use the same program in different threads, but feed different data to the threads, effectively parallelizing a SIMD workload across threads (as well as across vector instructions within threads). In this case, with Intel architectures, threads still run independently of each other (i.e., not in lockstep). This applies to Intel Xeon CPUs and Intel Xeon Phi coprocessors.

In multi-threaded applications in shared memory, the programmer must control access to shared data. While it is safe to read the same data with multiple threads, write operations to shared data must be protected in such a way that only one thread is guaranteed to modify data in memory at a time.

To use multi-threading in an application, the programmer must choose a parallel framework (i.e., API) and the corresponding library implementing it. There are two popular alternative low-level frameworks for multi-threading: POSIX threads, also known as Pthreads, and C++11 threads. While these frameworks are sometimes used to parallelize applications, these standards do not contain HPC-specific features such as workload balancing, automatic reduction, integration into offload programming, inter-operation with the Intel MPI library, etc. Computationally intensive algorithms are usually better implemented using one of the specialized standards for building thread-parallel applications, such as OpenMP or Intel Cilk Plus. Another popular high-level framework and library is called Intel Threading Building Blocks

(TBB). We do not discuss TBB in this book; refer to Intel's Documentation for more information.

In this section, we will discuss a popular parallel framework that highly suited for HPC applications: OpenMP. In Section 3.3, we will talk about an alternative developing framework called Intel Cilk Plus.

About OpenMP

OpenMP is a traditional, well-established cross-platform standard with which many high performance application developers are familiar. It provides high-level abstraction for task parallelism and isolates the programmer from low-level details of iteration space partitioning, data sharing, and thread creation, scheduling, and synchronization.

To parallelize an application with OpenMP, the programmer supplements the code with OpenMP pragmas. These pragmas instruct OpenMP-aware compilers to produce parallel versions of the respective statements and to bind to the OpenMP implementation.

It is possible serialize an OpenMP application by disabling OpenMP support in the compiler, and the code with OpenMP pragmas will still compile. In this case the pragmas will be treated as comments, and parallelization will not occur. It is possible, and often advisable, to develop OpenMP programs in such a way that the parallel and serialized versions produce identical results. The OpenMP standard, however, does not guarantee at the level of syntax that the results of the application will be the same with and without OpenMP (this in contrast with the Intel Cilk Plus parallel framework discussed in Section 3.3).

3.2.2. “Hello World” with OpenMP

A program with OpenMP directives begins execution as a single thread, called the *initial thread* of execution. It is executed sequentially until the first parallel construct is encountered. After that the initial thread creates a team of threads to be executed in parallel, and becomes the master of this team. All program statements enclosed by the parallel construct are executed in parallel by each thread in the team, including all routines called from within the enclosed statements. At the end of the parallel construct each thread waits for others to arrive. When that happens, the team is dissolved, and only the initial thread continues execution of the code following the parallel construct. Because applications may need to start and terminate parallel regions often, the Intel OpenMP implementation dissolves thread teams only logically but keeps the underlying machinery for threads “hot” for a period of time so that the next parallel region can be spawned quickly.

Listing 3.30 demonstrates a “Hello World” program in C++ using the OpenMP framework. Program execution proceeds serially until `#pragma omp parallel` requests a parallel region. In the parallel region, every thread executes the code in the scope of the pragma. Outside of the parallel region, execution serializes again.

```
#include <omp.h>
#include <cstdio>

int main(){
  printf("OpenMP with %d threads\n", omp_get_max_threads());
#pragma omp parallel
  {
    printf("Hello World from thread %d\n", omp_get_thread_num());
  }
  printf("Back to main thread\n");
}
```

Listing 3.30: `Hello-OpenMP.cc` – a “Hello World” program in OpenMP. Note the inclusion of the header file `omp.h`. Parallel execution is requested via `#pragma omp parallel`.

The procedure of compilation and execution of this code is shown in Listing 3.31. To link the Intel OpenMP library, the argument `-qopenmp` is used. In Intel C++ compiler versions prior to 15.0, the corresponding argument was spelled `-openmp`. By default, an OpenMP program will spawn as many threads as there are logical processors in the system (e.g., on a 24-core Intel Xeon CPU with enabled two-way hyper-threading, that number is equal to $2 \times 24 = 48$). However, the number of threads may be set to a different value using the environment variable `OMP_NUM_THREADS`. Note that output in Listing 3.31 is not ordered by thread number, because the order in which threads execute parallel code is not fixed in this example.

```
vega@lyra% icpc -qopenmp -o Hello-OpenMP Hello-OpenMP.cc
vega@lyra% export OMP_NUM_THREADS=4
vega@lyra% ./Hello-OpenMP
OpenMP with 4 threads
Hello World from thread 0
Hello World from thread 1
Hello World from thread 3
Hello World from thread 2
Back to main thread
```

Listing 3.31: Compiling and running `Hello-OpenMP.cc`

OpenMP participates in this code in two forms: pragmas request parallelization, and functions `omp_*()` return the parameters of the execution environment. It is possible to serialize an OpenMP program by compiling it with the argument `-qopenmp-stubs` as shown in Listing 3.32.

```
vega@lyra% icpc -qopenmp-stubs -o Hello-OpenMP Hello-OpenMP.cc
vega@lyra% ./Hello-OpenMP
OpenMP with 1 threads
Hello World from thread 0
Back to main thread
```

Listing 3.32: Serialization of an OpenMP program with `-qopenmp-stubs`.

Note that generally, a serialized OpenMP program does not have to produce the same results as a parallel code. This is because other OpenMP constructs that affect work distribution can rely on having different jobs in different threads.

3.2.3. For-Loops in OpenMP

A significant number of HPC tasks are centered around for-loops with pre-determined loop bounds and a constant increment of the loop iterator. Such loops can be easily parallelized in shared-memory systems using `#pragma omp parallel for` in OpenMP. Additional clauses of this pragma control how loop iterations are distributed across threads.

Figure 3.1 illustrates the workflow of a loop parallelized in shared memory using OpenMP. As this diagram illustrates, the execution of a parallel loop is initiated by a single thread. When the loop starts, multiple threads are spawned, and each thread gets a portion of the loop iteration space (called "chunk" in the terminology of OpenMP) to process. Depending on the user-defined scheduling algorithm, either all iterations of the loop are statically distributed before the loop begins, or, when a thread has completed its initial chunk of iterations, it receives from the scheduler another chunk to process.

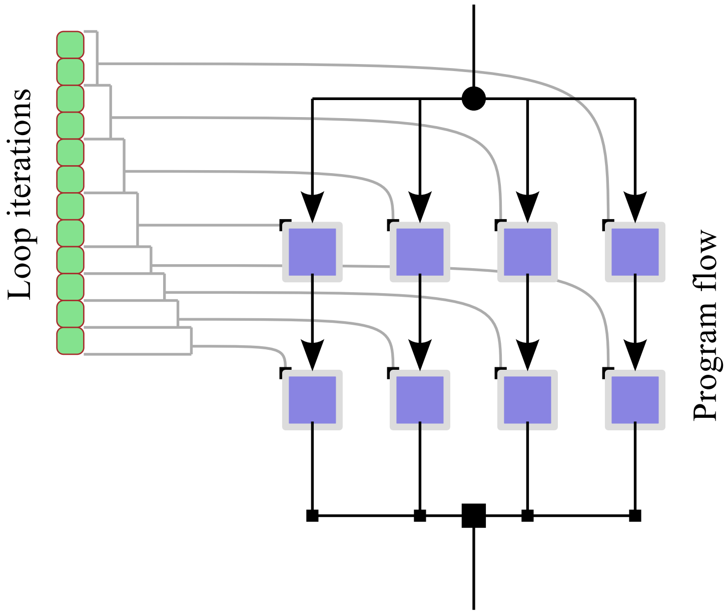


Figure 3.1: Parallelizing a for-loop with OpenMP.

Code samples illustrating the usage of OpenMP language constructs to parallelize loops follow.

Syntax for Parallel For-Loops

With OpenMP, `#pragma omp parallel for` must be placed before the loop to request its parallelization, as shown in Listing 3.33.

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
  printf("Iteration %d is processed by thread %d\n",
                                        i, omp_get_thread_num());
  // ... iterations will be distributed across available threads
}
```

Listing 3.33: The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

Alternatively, it is possible to start a parallelized loop by placing `#pragma omp for` nested inside a `#pragma omp parallel` construct, as demonstrated in Listing 3.34

```
#pragma omp parallel
{
  int private_number=0;
#pragma omp for
  for (int i=0; i<n; i++) {
    // ...iterations will be distributed across available threads
  }
}
```

Listing 3.34: When placing `#pragma omp for` closely nested inside a `#pragma omp parallel` region, there should be *no* word "parallel" before the word "for". Thread synchronization is implied at the end of the for-loop.

If a parallel loop has fewer iterations than the number of available OpenMP threads, then all iterations will start immediately with one iteration per thread. For parallel loops with more iterations than OpenMP threads, the run-time library will divide the iterations between threads as discussed in the next section. Generally, it is best to have many more iterations in a loop than the number of threads.

Loop Scheduling

Depending on the scheduling mode requested by the user, iteration assignment to threads can be either done before the start of the loop, or it can be decided dynamically. It is possible to tune the performance of for-loops in OpenMP by specifying the scheduling mode using a clause `schedule(`*mode*,*chunk_size*`)`. Here *mode* is one of: `static`, `dynamic` or `guided`, and *chunk_size* controls the granularity of work distribution.

`static` : with this mode, OpenMP evenly distributes loop iterations across threads before the loop begins. The iteration space is divided into contiguous chunks of size *chunk_size* (all chunks are equal or approximately equal in size), and these chunks are assigned to the threads in round-robin order. If *chunk_size* is not specified, iteration space is divided into as many approximately equal contiguous chunks as there are OpenMP threads assigned to the loop. This scheduling method has the smallest parallelization overhead, because no communication between threads is performed at runtime for scheduling purposes. The downside of this method is that it may result in load imbalance, if threads complete their iterations at different rates.

`dynamic` : with this scheduling mode, the iteration space is divided into chunks of size *chunk_size* (all chunks equal or approximately equal in size), and each thread executes a chunk of iterations, after which the scheduler assigns to it another chunk, until all iterations are processed. The default *chunk_size* in this mode is 1. This method has a greater overhead, but may improve load balance across threads.

`guided` : this method is similar to `dynamic`, except that the granularity of work assignment to threads (i.e., chunk size) decreases as the work nears completion. In this mode, the chunk size for each assignment to a thread is chosen proportional to the number of unprocessed iterations divided by the number of threads. The role of *chunk_size* is to set the minimum chunk size for all iterations (except, possibly, the last one). The default value of *chunk_size* in this mode is 1. This method requires more decision making in the scheduler than `dynamic`, but may result in higher performance due to overall fewer scheduling events and better load balancing.

With small chunk size, `dynamic` and `guided` have the potential to achieve better load balance at the cost of performing more scheduling work. With greater chunk size, the scheduling overhead is reduced, but load imbalance may be increased. Typically, the optimal chunk size must be chosen by the programmer empirically.

There are two methods to request the method of scheduling in a loop. The first method is to set the environment variable `OMP_SCHEDULE` (see Listing 3.35). This environment variable affects all loops in the application except those for which scheduling is specified with the second method.

```
vega@lyra% export OMP_SCHEDULE="dynamic,4"
vega@lyra% ./my_application
```

Listing 3.35: Controlling run-time scheduling of parallel loops with an environment variable. The format of the value of `OMP_SCHEDULE` is "`mode[,chunk_size]`", where `mode` is one of: `static`, `dynamic`, `guided`, and `chunk_size` is an integer.

The second method is to indicate the scheduling mode in the clauses of `#pragma omp for`. This method provides finer control, because different loops can be scheduled in different modes. However, the programmer has less freedom for modifying program behavior after compilation, because the clause takes precedence over the environment variable.

Listing 3.36 illustrates the clause method:

```
#pragma omp parallel for schedule(dynamic, 4)
for (int i = 0; i < N ; i++) {
  // ...
}
```

Listing 3.36: Controlling the run-time scheduling of a parallel loop with clauses of `#pragma omp for`.

See Section 4.4.3 for an example illustrating that the loop scheduling mode is a parameter of optimization.

3.2.4. Tasks in OpenMP

The OpenMP functionality called "tasks" allows to implement the so-called "fork-join" pattern of parallel execution, which involves:

i) creating one or more child tasks (the 'fork' step),
ii) running the child tasks concurrently with the parent task, and
iii) spinning at a barrier until child tasks terminate and only the parent task continues (the 'join' step).

Child tasks can fork, too, creating a tree of tasks. This model allows the programmer to express parallel algorithms that cannot be expressed with loop-centric parallelism, such as parallel recursion.

Note that we use the term "fork-join" in the sense of parallel pattern, following the definition of [19]. This term is also used to express the OpenMP execution pattern where the initial thread forks parallel threads, which are thereafter joined (see, e.g., [20]).

Figure 3.2 illustrates the progress of a parallel application employing the OpenMP tasks.

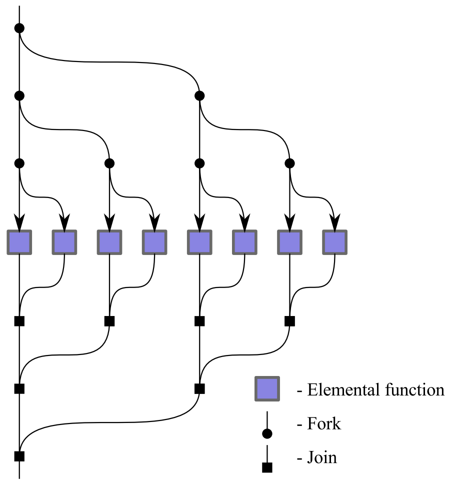


Figure 3.2: Fork-join pattern of parallel execution.

In OpenMP, it is possible to create a very large number of tasks (much greater than the number of threads). The OpenMP library schedules their execution in such a way that the hardware system is never over-subscribed.

The order of execution of parallel tasks in available cores is generally not the same as the order in which the tasks were spawned.

Tasks first appeared in OpenMP 3.0. The pragma for setting up a task is `#pragma omp task`. The task must be spawned in a parallel region in order to be scheduled for parallel execution. However, depending on the algorithm, not every thread must spawn initial tasks.

Listing 3.37 illustrates the usage of the OpenMP `task` pragma to create a parallel recursive algorithm.

```
#include <omp.h>
#include <cstdio>

void Recurse(const int task) {
  if (task < 10) {
    printf("Creating task %d...\n", task+1);
#pragma omp task
    {
      Recurse(task+1);
    }
    long foo=0; for (long i = 0; i < (1<<20); i++) foo+=i;
    printf("result of task %d in thread %d is %ld\n",
                              task, omp_get_thread_num(), foo);
  }
}

int main() {
#pragma omp parallel
  {
#pragma omp single
    Recurse(0);
  }
}
```

Listing 3.37: Source code `OpenMP-Task.cc` demonstrating the use of `#pragma omp task` to effect the fork-join parallel algorithm.

This code calls the function `Recurse()`, which forks off recursive calls to itself, requesting that those recursive calls are run in parallel. There are no explicit barriers in the code, however, a there is an implicit barrier at the end of the scope of `#pragma omp single`. The for-loop in the code is used only to make the tasks perform some arithmetic work, so that we can see the pattern of task creation and execution.

Note that `#pragma omp task` occurs inside of a parallel region, however, parallel execution is initially restricted to only one thread with `#pragma omp single`. This is a necessary condition for parallel recursion. Without `#pragma omp parallel`, all tasks will be executed by a single thread. Without `#pragma omp single`, multiple threads will start task number 0, which is not the desired behavior.

Listing 3.38 demonstrates the execution of the code `OpenMP-Task.cc` (Listing 3.37) with four threads.

```
vega@lyra% icpc -qopenmp -o OpenMP-Task OpenMP-Task.cc
vega@lyra% export OMP_NUM_THREADS=4
vega@lyra% ./OpenMP-Task
Creating task 1...
Creating task 2...
Creating task 3...
Creating task 4...
result of task 0 in thread 0 is 549755289600
result of task 1 in thread 2 is 549755289600
Creating task 5...
Creating task 6...
result of task 2 in thread 1 is 549755289600
Creating task 7...
result of task 3 in thread 3 is 549755289600
Creating task 8...
result of task 4 in thread 0 is 549755289600
Creating task 9...
result of task 5 in thread 2 is 549755289600
Creating task 10...
result of task 6 in thread 1 is 549755289600
result of task 7 in thread 3 is 549755289600
result of task 9 in thread 2 is 549755289600
result of task 8 in thread 0 is 549755289600
```

Listing 3.38: Compilation and running `OpenMP-Task.cc` shown in Listing 3.37.

One can see that the code forked off as many jobs as there were available threads (in this case, four), and the creation of other jobs had to wait until one of the threads became free.

It is also informative to see the difference between the parallel execution pattern and serial execution. To run the code serially, we can set the maximum number of OpenMP threads to 1, as shown in Listing 3.39

```
vega@lyra% export OMP_NUM_THREADS=1
vega@lyra% ./OpenMP-Task
Creating task 1...
Creating task 2...
Creating task 3...
Creating task 4...
Creating task 5...
Creating task 6...
Creating task 7...
Creating task 8...
Creating task 9...
Creating task 10...
result of task 9 in thread 0 is 549755289600
result of task 8 in thread 0 is 549755289600
result of task 7 in thread 0 is 549755289600
result of task 6 in thread 0 is 549755289600
result of task 5 in thread 0 is 549755289600
result of task 4 in thread 0 is 549755289600
result of task 3 in thread 0 is 549755289600
result of task 2 in thread 0 is 549755289600
result of task 1 in thread 0 is 549755289600
result of task 0 in thread 0 is 549755289600
vega@lyra%
```

Listing 3.39: Running `OpenMP-Task.cc` from Listing 3.37 with a single OpenMP thread.

Evidently, in the serial version, the execution recursed into the deepest level before returning to the calling function. This is the behavior what one would expect from this code if it was stripped of all OpenMP pragmas.

3.2.5. Shared and Private Variables

In OpenMP parallel regions and loops, multiple threads have access to variables that had been declared before the parallel region was started. Consider example in Listing 3.40.

```
#include <omp.h>
#include <cstdio>

int main() {
  int someVariable = 5;
#pragma omp parallel
  {
    printf("For thread %d, someVariable=%d\n",
                   omp_get_thread_num(), someVariable);
  }
}
```

```
vega@lyra% icpc -o OpenMP-Shared OpenMP-Shared.cc -qopenmp
vega@lyra% export OMP_NUM_THREADS=4
vega@lyra% ./OpenMP-Shared
For thread 0, someVariable=5
For thread 2, someVariable=5
For thread 1, someVariable=5
For thread 3, someVariable=5
vega@lyra%
```

Listing 3.40: Code `OpenMP-Shared.cc` illustrating the use of shared variables in OpenMP.

In `OpenMP-Shared.cc`, all threads execute the code inside of the scope of `#pragma omp parallel`. All of these threads have access to variable `someVariable` declared before the parallel region. By default, all variables declared before the parallel region are shared between threads. This means that (a) all threads see the value of shared variables, and (b) if one thread writes to the shared variable, all other threads see the modified value. The latter case may lead to race conditions and unpredictable behavior, unless the write operation is protected as discussed in Section 3.2.6.

In some cases, it is preferable to have a variable of private nature, i.e., have an independent copy of this variable in each thread. To effect this behavior, the programmer may declare this variable inside the parallel region as shown in Listing 3.41. Naturally, the programmer can initialize the value of this private variable with the value of a shared variable.

```
int varShared = 3;
#pragma omp parallel
  {
    // Each thread will have a copy of varPrivateLocal
    int varPrivateLocal = varShared;
    // ...
#pragma omp for
    for (int i = 0; i < N; i++) {
       int varTemporary = varPrivateLocal;
    }
  }
}
```

Listing 3.41: Variables declared outside the OpenMP parallel region are shared, variables declared inside are private.

In the code in Listing 3.41, an independent copy of `varPrivateLocal` is available in each thread. This variable persists throughout the parallel region. Similarly, an independent copy of `varTemporary` will exist in each thread. The value of this variable persists for the duration of a single loop iteration, but does not persist across loop iterations.

Declaring variables outside or inside of the parallel scope is sufficient to control variable sharing in C and C++. However, for compatibility with Fortran, OpenMP provides an additional variable sharing control, which also works in C and C++. To create in each thread a *private* copy of some of the variables declared *before* the parallel region, it is possible to use clauses `private` and `firstprivate` in `#pragma omp parallel` as shown in Listing 3.42. With clause `private`,

a) the variable is private to each thread,
b) the initial value of a private variable is undefined, and
c) the value of the variable in the encompassing scope does not change at the end of the parallel region.

Clause `firstprivate` is similar to `private`, but the initial value is initialized with the value outside of the parallel region.

```
#include <omp.h>
#include <cstdio>

int main() {
  int varShared = 5;
  int varPrivt = 1;
  int varFirstprvt = 2;
#pragma omp parallel private(varPrivt) firstprivate(varFirstprivt)
  {
    printf("Thread %d: varShared=%d varPrivt=%d varFirstprivt=%d\n",
           mp_get_thread_num(), varShared, varPrivt, varFirstprivt);
    if (omp_get_thread_num() == 0) {
       varShared = -varShared; // Race condition=undefined behavior
       varPrivt = -varPrivt; // OK: each thread has own varPrivt
       varFirstprivt = -varFirstprivt; // OK for the same reason
    }
  }
  printf("Serial reg: varShared=%d varPrivt=%d varFirstprivt=%d\n",
          varShared, varPrivt, varFirstprivt);
}
```

```
vega@lyra% icpc -o OpenMP-Private OpenMP-Private.cc -qopenmp
vega@lyra% export OMP_NUM_THREADS=4
vega@lyra% ./OpenMP-Private
For thread 0, varShared=5 varPrivt=0 varFirstprivt=2
For thread 1, varShared=5 varPrivt=0 varFirstprivt=2
For thread 2, varShared=5 varPrivt=0 varFirstprivt=2
For thread 3, varShared=-5 varPrivt=0 varFirstprivt=2
After parallel region, varShared=-5 varPrivt=1 varFirstprivt=2
vega@lyra%
```

Listing 3.42: Code `OpenMP-Private.cc` illustrating the use of shared variables in OpenMP.

Note that in C++, clauses `private` and `firstprivate` duplicate the functionality of scope-local variables demonstrated in Listing 3.41. However, in Fortran the user must declare all variables at the beginning of the function, and therefore there is no way to avoid using the clauses `private`, `firstprivate` and `lastprivate`.

Another type of private variable behavior in OpenMP is effected by clause `lastprivate`, which applies to `#pragma omp parallel for`. For `lastprivate` variables, the value in the last iteration is copied back to the scope outside the parallel region.

Some programmers may also find the clause `default` useful, as it allows to define the default nature of variables as `shared`, or `none` e.g.,

```
#pragma omp parallel for default(none) shared(a,b) lastprivate(c,d)
  for (int i = 0; i < n; i++) {
    ...
  }
```

Listing 3.43: Using clause `default` to request that all variables declared outside the OpenMP parallel region is not visible within the region.

In the above code, variables `a`, `b`, and `i` will be `shared`, variables `c` and `d` will be `lastprivate`, and all other variables will be `none`, thus not visible for the parallel region. With `default(none)`, if the programmer forgets to specify the sharing type for any of the variables used in the parallel region, code compilation will fail — this behavior may be desirable in complex cases for explicit variable behavior check.

3.2.6. Synchronization: Avoiding Unpredictable Behavior

Up until now, the discussion of parallelism in shared memory was restricted to algorithms without any interaction between threads. However, for certain algorithms and operations, synchronization between threads may be necessary. This section discusses the functionality available in OpenMP for synchronization: mutexes and barriers. Note that in general, synchronization impedes the parallel scalability of applications. Whenever possible, instead of synchronization operations, programmers must use reduction and private variables as discussed in Section 3.2.7.

Data Races

One must never allow multiple threads to simultaneously modify a shared variable, because concurrent modification of data may lead to unpredictable results. Operations that modify shared data must be protected with a synchronization construct called mutex (mutually exclusive events).

Consider a parallel program in which two threads increment a shared variable. Timeline in the left-hand side diagram in Figure 3.3 illustrates how threads read the variable, increment it, and write it back to memory without waiting for each other.

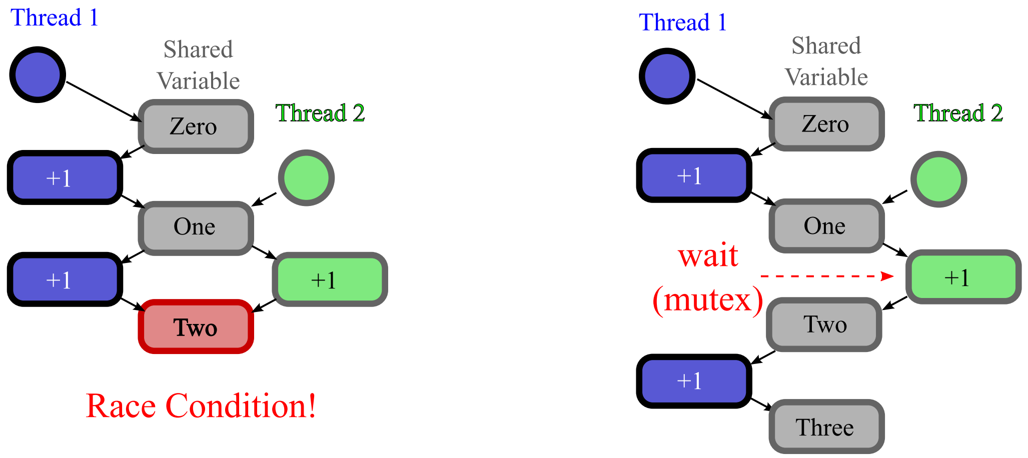


Figure 3.3: Illustration of data races (left) and mutexes (right) in multi-threaded programs.

With unprotected concurrent access to a shared variable illustrated in Figure 3.3, a situation may occur when two threads simultaneously read the same state of the variable. In this case, when the data is written back to memory, only the result of the slower thread will survive, and the increment performed by the faster thread will be lost. This situation is known as a data race, and it leads to unpredictable results in a parallel program.

Because generally, parallel programs are expected to produce consistent results from run to run, race conditions must be avoided. They can be avoided using a synchronization tool known as a lock, or a mutex. The code protected by a lock can be executed by only one thread at a time. All other threads must wait until the lock is lifted by the thread executing the code, i.e., the thread team processing the mutex is serialized. The effect of a mutex is illustrated in the right-hand side diagram in Figure 3.3.

Mutexes in OpenMP: Critical Sections

For specificity, consider the following example:

```
 1  #include <omp.h>
 2  #include <cstdio>
 3
 4  int main() {
 5     const int n = 1000;
 6     int sum = 0;
 7  #pragma omp parallel for
 8     for (int i = 0; i < n; i++)
 9       sum = sum + i; // Race condition
10     printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
11  }
```

```
vega@lyra% icpc -o OpenMP-Race OpenMP-Race.cc -qopenmp
vega@lyra% export OMP_NUM_THREADS=32
vega@lyra% ./OpenMP-Race
sum=208112 (must be 499500)
vega@lyra%
```

Listing 3.44: Code `OpenMP-Race.cc` has unpredictable behavior and produces incorrect results due to a race condition in line 9.

In line 9 of code `OpenMP-Race.cc` in Listing 3.44, a situation known as a race condition occurs. The problem is that variable `sum` is shared between all threads, and therefore more than one thread may execute this line concurrently. However, what if one thread updates `sum` while another thread was incrementing the old value of `sum`? This may, and will, lead to an incorrect calculation. Indeed, the output shows a value of `sum=208112` instead of `499500`. Moreover, if we run this code multiple times, every time the result will be different, because the pattern of races between threads will vary from run to run. This parallel program has unpredictable behavior! How does one resolve this problem?

The easiest, albeit the most inefficient way to protect a portion of a parallel code from concurrent execution in OpenMP is a critical section, as illustrated in Listing 3.45. `#pragma omp critical` used in this code protects the code inside its scope from concurrent execution. The whole iteration space will still be executed in parallel, but only one thread at a time will be allowed to enter the critical section, while other threads wait their turn. At this stage in the curriculum we are not concerned with performance, but let us note that this is a very inefficient way to resolve the race condition in the problem shown in Listing 3.44. We provide this example because in some cases, a critical section is the only way to avoid unpredictable behavior.

```
#pragma omp parallel for
   for (int i = 0; i < n; i++) {
#pragma omp critical
     { // Only one thread at a time can execute this section
          sum = sum + i;
     }
   }
```

```
vega@lyra% icpc -o OpenMP-Critical OpenMP-Critical.cc -qopenmp
vega@lyra% ./OpenMP-Critical
sum=499500 (must be 499500)
vega@lyra%
```

Listing 3.45: Parallel fragment of code `OpenMP-Critical.cc` has predictable behavior, because the race condition was eliminated with a critical section.

Mutexes in OpenMP: Atomic Operations

A more efficient mutex, albeit limited to certain functions and data types, is the use of atomic operations. Atomic operations allow the program to safely update a scalar variable in a parallel context. These operations are effected with `#pragma omp atomic`, as shown in Figure 3.46.

```
#pragma omp parallel for
    for (int i = 0; i < n; i++) {
        // Lightweight synchronization
#pragma omp atomic
        sum += i;
    }
```

Listing 3.46: This parallel fragment of code `OpenMP-Atomic.cc` has predictable behavior, because the race condition was eliminated with an atomic operation. Note that for this specific example, atomic operations are not the most efficient solution.

Only the following operations can be executed as atomic:

Read : operations in the form `v = x`
Write : operations in the form `x = v`
Update : operations in the form `x++`, `x--`, `--x`, `++x`, `x` *binop*= *expr* and `x = x` *binop expr*
Capture : operations in the form `v = x++`, `v = x--`, `v = --x`, `v = ++x`, `v = x` *binop expr*

Here `x` and `v` are scalar variables, *binop* is one of `+, *, -, - /, &, ^ , |, «, »`. No "trickery" is allowed for atomic operations: no operator overload, no non-scalar types, no complex expressions.

In many cases, atomic operations are an adequate solution for accessing and modifying shared data. However, in this particular case, the parallel scalability of the algorithm may be further improved by using reducers instead of atomic operations, as discussed in Section 3.2.7.

Barriers in OpenMP: `#pragma omp barrier`

When multiple threads execute a parallel region, there are no guarantees that they execute each line of parallel code at the same time. That is because Intel Xeon and Intel Xeon Phi cores do not run in lockstep. However, it is possible to occasionally synchronize all threads at a barrier. This may be necessary to get a consistent view of a shared variable modified by one of the threads.

Usage of a barrier is illustrated in Listing 3.47 and Listing 3.48. Without the barrier, it is possible that threads other than 0 will see `sharedVar=0`.

```
#include <cstdio>
#include <omp.h>

int main() {
  int sharedVar = 0;
#pragma omp parallel
  {
    const int i = omp_get_thread_num();
    if (i == 0) sharedVar = 1;

    // Some threads may reach this line before thread 0
    // reaches line 9. They will see sharedVar=0
    printf("Thread %d sees sharedVar=%d\n", i, sharedVar);
  }
}
```

```
vega@lyra% icpc -o OpenMP-NoBarrier OpenMP-NoBarrier.cc -qopenmp
vega@lyra% export OMP_NUM_THREADS=4
vega@lyra% ./OpenMP-NoBarrier
Thread 0 sees sharedVar=1
Thread 3 sees sharedVar=1
Thread 2 sees sharedVar=0
Thread 1 sees sharedVar=1
```

Listing 3.47: Without a barrier, threads do not to operate in lockstep.

The barrier in Listing 3.48 guarantees that all threads will see the value `sharedVar=1`.

```
 1 #include <cstdio>
 2 #include <omp.h>
 3 
 4 int main() {
 5   int sharedVar = 0;
 6 #pragma omp parallel
 7   {
 8     const int i = omp_get_thread_num();
 9     if (i == 0) sharedVar = 1;
10 
11 #pragma omp barrier
12 
13     // At this point, all threads see sharedVar=1
14     // due to the barrier in line 11
15     printf("Thread %d sees sharedVar=%d\n", i, sharedVar);
16   }
17 }
```

```
vega@lyra% icpc -o OpenMP-Barrier OpenMP-Barrier.cc -qopenmp
vega@lyra% export OMP_NUM_THREADS=4
vega@lyra% ./OpenMP-Barrier
Thread 0 sees sharedVar=1
Thread 3 sees sharedVar=1
Thread 2 sees sharedVar=1
Thread 1 sees sharedVar=1
```

Listing 3.48: Synchronizing steps at a barrier.

Barriers are not permitted inside of parallel for-loops, critical sections, and some other OpenMP environments.

Note that a barrier is implied at the end of a parallel region. This means that code execution does not proceed until all iterations of the parallel loop have been performed, or until the last statement of the parallel region has been executed in every thread. Implicit barrier also exist at the end of a parallel for-loop.

Barriers in OpenMP: `#pragma omp taskwait`

For task parallelism with `#pragma omp task` (see Section 3.2.4), OpenMP has `#pragma omp task` that pauses execution in the current thread until all tasks spawned by that thread are completed.

```
#include <omp.h>
#include <cstdio>
int main() {
  const int N=1000;
  int* A = (int*)malloc(N*sizeof(int));
  for (int i = 0; i < N; i++) A[i]=i;
#pragma omp parallel
  {
#pragma omp single
    {
      // Compute the sum in two threads
      int sum1=0, sum2=0;
#pragma omp task shared(A, N, sum1)
      {   for (int i = 0; i < N/2; i++) sum1 += A[i];    }
#pragma omp task shared(A, N, sum2)
      {   for (int i = N/2; i < N; i++) sum2 += A[i];    }
      // Wait for forked off tasks to complete
#pragma omp taskwait
      printf("Result=%d (must be %d)\n", sum1+sum2, ((N-1)*N)/2);
    }  }
  free(A); }
```

```
vega@lyra% icpc -o OpenMP-TaskWait OpenMP-TaskWait.cc -qopenmp
vega@lyra% ./OpenMP-TaskWait
Result=499500 (must be 499500)
```

Listing 3.49: Code `OpenMP-TaskWait.cc` illustrates `#pragma omp taskwait`.

The code in Listing 3.46 is an inefficient way to approach the problem, because it uses only two threads. A better way to perform parallel reduction is described in Section 3.2.7. Nevertheless, for scalable task-based parallel algorithms, `#pragma omp taskwait` is a native way in OpenMP to implement synchronization points.

3.2.7. Reduction: Avoiding Synchronization

Parallel algorithms that require synchronization only to modify a common quantity can be expressed in terms of reduction. This possibility arises if the operation with which the common quantity is calculated is associative (such as integer addition or multiplication) or approximately associative (such as floating-point addition or multiplication). The associative property means that the order of operations does not affect the result.

OpenMP has reduction clauses for parallel pragmas, which works for reduction of scalars. It is also possible to instrument a reduction algorithm using private variables and minimal synchronization. Properly instrumented parallel reduction avoids excessive synchronization and communication, which improves the parallel scalability of an application.

Reduction Clause in OpenMP

In OpenMP, parallel regions can automatically perform reduction for certain operations on scalar variables. Listing 3.50 illustrates the algorithm originally shown in Listing 3.44, Listing 3.45 and Listing 3.46, this time instrumented using the OpenMP reduction clause.

```
#include <omp.h>
#include <cstdio>

int main() {
  const int n = 1000;
  int sum = 0;
#pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < n; i++)
     sum = sum + i;
   printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
}
```

```
vega@lyra% icpc -o OpenMP-Reduction OpenMP-Reduction.cc -qopenmp
vega@lyra% ./OpenMP-Reduction
sum=499500 (must be 499500)
```

Listing 3.50: In `OpenMP-Reduction.cc`, data race is eliminated using a reduction clause.

The syntax of the reduction clause is `reduction(`*operator* **:** *variables*`)`, where *operator* is one of: `+`, `*`, `-`, `&`, `|`, `^` , `&&`, `||`, `max` or `min`, and *variables* is a comma-separated list of variables to which these operations are applied.

Reduction with Private Variables

An alternative method of implementing reduction is via private variables and a critical section or an atomic operation after the loop. This method is useful when the type of data or the type of the reduction operation is not supported by the OpenMP reduction clause.

With the private variable method, each thread must have a private variable of the same type as the global reduction variable. In each thread, the reduction operation is applied to that private variable without synchronization with other threads. At the end of the loop, critical sections or atomic operations are used in order to reduce the private variables from each thread into the global variable. The principle of this method is shown in Listing 3.51. It works well if `n` is much greater than the number of threads.

```
#include <omp.h>
#include <cstdio>
int main() {
  const int n = 1000;
  int sum = 0;
#pragma omp parallel
  {
    int sum_th = 0;
#pragma omp for
    for (int i = 0; i < n; i++)
      sum_th = sum_th + i;

#pragma omp atomic
    sum += sum_th;
    }
  printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
}
```

Listing 3.51: Code `OpenMP-Reduction2.cc` implements reduction using private variables and a minimum reduction section.

Reduction with a Container Array

Alternatively, the programmer may declare a shared container array for thread-private data as in Listing 3.52. A word of warning here is that a condition called false sharing may ruin the performance in this case, and measures should be taken to avoid it. See Section 4.4.2 for more details.

```
#include <omp.h>
#include <cstdio>
int main() {
  const int n = 1000;
  const int T = omp_get_max_threads();
  int sum = 0;
  int sumContainer[T]=0;
#pragma omp parallel
  {
    const int th = omp_get_thread_num();
    sumContainer[th] = 0;

    // Beware of false sharing! Use padding for better performance.
#pragma omp for
    for (int i = 0; i < n; i++)
      sumContainer[th] = sumContainer[th] + i;
  }

  // Perform serially in initial thread
  for (int th = 0; th < T; th++)
    sum = sum + sumContainer[th];

  printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
}
```

Listing 3.52: Code `OpenMP-Reduction3.cc` implements reduction using a container array with subsequent serial reduction.

3.3. Task Parallelism with Intel Cilk Plus

Intel Cilk Plus is an alternative to the OpenMP framework. Compared to OpenMP, Intel Cilk Plus gives the programmer less control over low-level details, but in return, allows to express complex algorithms with ease.

There are only three keywords in the Cilk Plus framework: `_Cilk_for`, `_Cilk_spawn` and `_Cilk_sync`[1]. They allow the programmer to implement of a variety of parallel algorithms. Programming for Intel Xeon Phi coprocessors may require two additional keywords `_Cilk_shared` and `_Cilk_offload`.

The nature of Intel Cilk Plus keywords and semantics preserves the serial nature in parallel programs. The lack of locks in the code is compensated by the availability of hyper-objects, which facilitate and motivate more scalable parallel algorithms.

Instead of threads, Intel Cilk Plus uses instances called "workers". Each worker has a queue of work-items to process, and the number of workers and work distribution between them is decided at runtime. Intel Cilk Plus uses an efficient scheduling algorithm based on "work stealing", which may be more efficient than OpenMP in complex multi-program applications. With work stealing, a worker that has run out of work will query other workers for unprocessed work-items. If unfinished work-items are found, the idle worker will "steal" a part of the queue from a busy worker.

Another new concept to learn with Intel Cilk Plus is strands. A strand is a sequence of instructions, which starts or ends with a statement that changes the parallelism (i.e., invokes a parallel loop, spawns off a function or sets up a barrier to sync spawned children).

With Intel C++ compiler, the Intel Cilk Plus library is automatically linked if the application uses the Intel Cilk Plus keywords mentioned above. However, in order to make certain additional objects of Intel Cilk Plus API available, the programmer must include `<cilk/cilk_api.h>`.

See the Intel C++ Compiler Reference for more information on the available functions and on the execution model of Intel Cilk Plus.

[1] `cilk_for`, `cilk_spawn` and `cilk_sync` are alternative spellings. They produce identical results

3.3.1. “Hello World” in Intel Cilk Plus

A sample Intel Cilk Plus program is shown in Listing 3.53.

```
 1 #include <cilk/cilk_api.h>
 2 #include <cstdio>
 3 
 4 int main(){
 5   const int nw=__cilkrts_get_nworkers();
 6   printf("Cilk Plus with %d workers.\n", nw);
 7 
 8   _Cilk_for (int i=0; i<nw; i++) { // No work; gets serialized
 9     printf("Hello World from worker %d (no work in loop)\n",
10            __cilkrts_get_worker_number());
11   }
12 
13   _Cilk_for (int i=0; i<nw; i++) {
14     float foo=10.0; // Some work to do
15     while (foo > 0) { foo -= 1.0; }
16     printf("Hello Again from worker %d (result=%f)\n",
17            __cilkrts_get_worker_number(), foo);
18   }
19 }
```

```
vega@lyra% export CILK_NWORKERS=4
vega@lyra% icpc Cilk-Hello.cc
vega@lyra% ./a.out
Cilk Plus with 4 workers.
Hello World from worker 0 (no work in loop)
Hello World from worker 0 (no work in loop)
Hello World from worker 0 (no work in loop)
Hello World from worker 0 (no work in loop)
Hello Again from worker 0 (result=0.000000)
Hello Again from worker 1 (result=0.000000)
Hello Again from worker 3 (result=0.000000)
Hello Again from worker 2 (result=0.000000)
```

Listing 3.53: Hello World program in Intel Cilk Plus and its compilation.

In `Cilk-Hello.cc` shown above, two parallel loops are run: one in line 8 and the other in line 13. Important thing to note here is that the first loop was executed by only one worker, i.e., this loop was serial. At runtime, it turned out to be inefficient to parallelize the first loop because no computational workload is present. Thus the first loop was serialized. The second loop has some computing workload, and at runtime it was run in parallel. The internal work scheduling is an integral part of the Intel Cilk Plus execution model, and, unlike OpenMP, currently there is no way to explicitly control work distribution.

Another important piece of information to infer from Listing 3.53 is that the environment variable `CILK_NWORKERS` controls the number of workers across which the application is parallelized. By default, the number of workers is set equal to the number of logical processors in the system.

The programmer may force serialization of an Intel Cilk Plus application by compiling it with the flag `-cilk-serialized`, as shown in Listing 3.54. In this case, only one worker is created, regardless of the value of `CILK_NWORKERS`.

```
vega@lyra% icpc -cilk-serialize Cilk-Hello.cc
vega@lyra% export CILK_NWORKERS=4
vega@lyra% ./a.out
Cilk Plus with 1 workers.
Hello World from worker 0 (no work in loop)
Hello Again from worker 0 (result=0.000000)
```

Listing 3.54: Serialization of an Intel Cilk Plus application.

3.3.2. For-Loops in Intel Cilk Plus

In Intel Cilk Plus, a parallel for-loop is created as shown in Listing 3.55.

```
int sharedVar = 0; // shared across strands

_Cilk_for (int i=0; i<n; i++) {
  // ... iterations will be distributed across workers...
  int privateVar = 0; // private to each strand

  printf("Iteration %d is processed by worker %d\n",
                              i, __cilkrts_get_worker_number());
}
```

Listing 3.55: The Intel Cilk Plus library will distribute the iterations of the loop following across available workers.

Variables declared in the body of the loop are available only on the worker processing these variables. For example, `privateVar` in Listing 3.55 is private to each strand. Variables visible in the scope in which the loop is launched (e.g., `sharedVar` in Listing 3.55) are shared across all strands, and therefore must be protected from race conditions.

There are no native locks in the Intel Cilk Plus framework. To efficiently share data between Intel Cilk Plus workers (rather than strands), special C++ templates called *hyper-objects* must be used, as described in Section 3.3.5. Reducers are hyper-objects that enable parallel reduction of data across workers, and holders are hyper-objects that enable private scratch objects that persist in each worker across multiple strands.

Just like with OpenMP, the run-time Intel Cilk Plus library will process iterations of a parallel loop concurrently with as many workers as are available. The total iteration space will be divided into chunks, each of which will be executed serially by one of the Intel Cilk Plus workers. By default, the maximum number of workers in Intel Cilk Plus is equal to the number of logical processors in the system. The number of workers actually used at runtime is dependent on the amount of work in the loop, and may be smaller than the maximum. This behavior is different from OpenMP, as OpenMP by default spawns a pre-determined number of threads, regardless of the amount of work in the loop.

Similarly to OpenMP, Intel Cilk Plus allows the user to control the work sharing algorithm in for-loops by setting the granularity of work distribution. This is done with `#pragma cilk grainsize`, as illustrated in Listing 3.56

```
#pragma cilk grainsize = 4
_Cilk_for (int i = 0; i < N; i++) {
 // ...
}
```

Listing 3.56: Controlling grain size in Intel Cilk Plus.

The value of grainsize is the maximum number of iterations assigned to any worker in one scheduling step. Like with OpenMP, the choice of grainsize is a compromise between load balancing and the overhead of scheduling. The default value of grainsize chosen by Intel Cilk Plus works well enough in many cases.

Unlike OpenMP, Intel Cilk Plus has only one mode of scheduling, work stealing. Work stealing, depending on the nature of the calculation, may be more or less efficient than OpenMP scheduling methods.

3.3.3. Fork-Join Model and Spawning in Intel Cilk Plus

In Intel Cilk Plus, the keyword `_Cilk_spawn` effects the fork-join model. This keyword must be placed before the function that is forked off, and the function will then be executed in parallel with the current scope. Listing 3.57 demonstrates the same program as Listing 3.37, but now in the Intel Cilk Plus framework.

```
#include <cstdio>
#include <cilk/cilk_api.h>

void Recurse(const int task) {
  if (task < 10) {
    printf("Creating task %d...\n", task+1);

    _Cilk_spawn Recurse(task+1);
    long foo=0;
    for (long i = 0; i < (1L<<20L); i++)
      foo += i;

    printf("result of task %d in worker %d is %ld\n", task,
           __cilkrts_get_worker_number(), foo);
  }
}

int main() {
  Recurse(0);
}
```

Listing 3.57: Source code `Cilk-Spawn.cc` demonstrating the use of `_Cilk_spawn` to effect the fork-join parallel algorithm.

Note that there is no explicit synchronization in this code (see the discussion of the barrier `_Cilk_sync` in Section 3.3.4). There is, however, implicit synchronization at the end of the function `Recurse()` with all tasks spawned off by this instance of the function.

Listing 3.58 demonstrates compiling and running this code. Unlike OpenMP code `OpenMP-Task.cc`, this code parallelized with Intel Cilk Plus had spawned all tasks first and queued them for pick up by workers before proceeding to run the tasks.

```
vega@lyra% icpc -o Cilk-Spawn Cilk-Spawn.cc
vega@lyra% export CILK_NWORKERS=4
vega@lyra% ./Cilk-Spawn
Creating task 1...
Creating task 2...
Creating task 3...
Creating task 4...
Creating task 5...
Creating task 6...
Creating task 7...
Creating task 8...
Creating task 9...
Creating task 10...
result of task 9 in worker 0 is 549755289600
result of task 8 in worker 0 is 549755289600
result of task 1 in worker 1 is 549755289600
result of task 0 in worker 3 is 549755289600
result of task 2 in worker 2 is 549755289600
result of task 7 in worker 0 is 549755289600
result of task 6 in worker 0 is 549755289600
result of task 3 in worker 1 is 549755289600
result of task 5 in worker 2 is 549755289600
result of task 4 in worker 3 is 549755289600
vega@lyra%
```

Listing 3.58: Compiling and running `Cilk-Spawn.cc` from Listing 3.57 with four workers.

3.3.4. Synchronization with Spawned Tasks

At this point, the discussion will proceed to synchronization in Intel Cilk Plus. The equivalent of `#pragma omp taskwait` in Intel Cilk Plus is the language construct `_Cilk_sync`. It plays the same role for tasks that have been forked off with `_Cilk_spawn`. However, in contrast with OpenMP, this statement is the only native means of explicit synchronization in Intel Cilk Plus. Listing 3.59 is an Intel Cilk Plus implementation of the algorithm for which the OpenMP implementation is given in Listing 3.49.

```
#include <cstdio>

void Sum(const int* A,
         const int start, const int end, int & result) {
  for (int i = start; i < end; i++)
    result += A[i];
}

int main() {
  const int N=1000;
  int* A = (int*)malloc(N*sizeof(int));
  for (int i = 0; i < N; i++) A[i]=i;

  // Compute the sum with two tasks
  int sum1=0, sum2=0;

  _Cilk_spawn Sum(A, 0, N/2, sum1);
  _Cilk_spawn Sum(A, N/2, N, sum2);

  // Wait for forked off sums to complete
  _Cilk_sync;

  printf("Result=%d (must be %d)\n", sum1+sum2, ((N-1)*N)/2);

  free(A);
}
```

Listing 3.59: Code `Cilk-Sync.cc` illustrates the usage `_Cilk_sync`.

Just as with OpenMP, this is an inefficient way to implement parallel reduction, and a better method is described in Section 3.3.5

Implicit Synchronization Intel Cilk Plus

In addition to the explicit synchronization method described above, Intel Cilk Plus contains implicit synchronization points at the beginning and end of parallel loops and at the end of functions. This means that execution does not proceed until all iterations of the parallel loop have been performed, or until the spawned off children of the current instance of the function return.

Locks in Intel Cilk Plus

There are no native locks in Intel Cilk Plus. However, Intel Cilk Plus interoperates with locks in, e.g., Threading Building Blocks. This topic is beyond the scope of this training. As a rule of thumb, for optimum performance and portability, developers should try to design parallel algorithms using only native Intel Cilk Plus keywords and hyper-objects, instead of resorting to additional synchronization methods.

3.3.5. Reduction: Avoiding Synchronization

Data Sharing in Intel Cilk Plus and Hyper-Objects

In Intel Cilk Plus, there is no additional pragma-like control over shared or private nature of variables. All variables declared before `_Cilk_for` are shared, and all variables declared inside the loop are only visible to the strand executing the iteration, and exist for the duration of the loop iteration.

Also, Intel Cilk Plus, compared to OpenMP, allows the user less fine-grained control over synchronization, but makes up for it with versatile support for hyper-objects: reducers and holders.

Reducers in Intel Cilk Plus

Reducers are variables that hold shared data, yet these variables can be safely used by multiple strands of a parallel application. At runtime, each worker operates on its own private copy of the data, which decreases synchronization and communication between workers. At the end of a parallel for-loop, reduction is performed across all worker-private copies of data held by the reducer template.

Let us demonstrate an Intel Cilk Plus implementation of the example shown in Listing 3.50. Listing 3.60 demonstrates the parallel sum reduction algorithm with Intel Cilk Plus.

```
#include <cilk/reducer_opadd.h>
#include <cstdio>

int main() {
  const int n = 1000;
  cilk::reducer_opadd<int> sum;
  sum.set_value(0);
  _Cilk_for (int i = 0; i < n; i++) {
    sum += i;
  }
  printf("sum=%d (must be %d)\n", sum.get_value(), ((n-1)*n)/2);
}
```

Listing 3.60: Code `Cilk-Reduction.cc`: parallel reduction using reducers.

Note the following details in `Cilk-Reduction.cc`:

a) Header file corresponding to a specific reducer must be included. In this case, it is `cilk/reducer_opadd.h` for the addition reducer.

b) Reducers are generic (template) C++ classes.

c) Inside the parallel loop, the reducer `sum` is used just like a regular variable of type `int`, except that only operations `+=`, `-=`, `+`, `-`, `++` and `--` are allowed for it.

d) Outside the parallel region, the reducer can only be used via accessors and mutators (in this case, `get_value()` and `set_value()`).

Predefined Reducers

The list of reducers supported by Intel Cilk Plus is shown below. Reducer names are self-explanatory, and additional information can be found in Intel C++ Compiler reference.

Name	Header file	Operation
`reducer_list_append`	`<cilk/reducer_list_append.h>`	`push_back()`
`reducer_list_prepend`	`<cilk/reducer_list_prepend.h>`	`push_front()`
`reducer_max`, `reducer_max_index`	`<cilk/reducer_max.h>`	`cilk::max_of`
`reducer_min`, `reducer_min_index`	`<cilk/reducer_min.h>`	`cilk::min_of`
`reducer_opadd`	`<cilk/reducer_opadd.h>`	+=, −=, +,−, ++ and −−
`reducer_opand`	`<cilk/reducer_opand.h>`	`&` and `&=`
`reducer_opor`	`<cilk/reducer_opor.h>`	`|` and `|=`
`reducer_opxor`	`<cilk/reducer_opxor.h>`	`^` and `^=`
`reducer_ostream`	`<cilk/reducer_ostream.h>`	<<
`reducer_basic_string`	`<cilk/reducer_string.h>`	+= to create a string. `reducer_string` and `reducer_wstring` are shorthands for `reducer_basic_string` for types `char` and `wchar_t`, respectively

Table 3.4: Predefined Intel Cilk Plus reducers

Custom Reducers

The power of reducers in Intel Cilk Plus is greatly enhanced by support for user-defined reducers. This procedure is described in the Intel C++ Compiler reference. However, for a lot of applications, the scope of reducers provided in Intel Cilk Plus may be sufficient.

Holders in Intel Cilk Plus

Holders in Intel Cilk Plus are hyper-objects that allow thread-safe write accesses to common data. Holders are similar to reducers, with the exception that they do not support synchronization at the end of the parallel region. This allows to instrument holders with a single C++ template class called `cilk::holder`.

The role of holders in Intel Cilk Plus is similar to the role of private variables in OpenMP declared in the same way that the variable `sum_th` is declared in Listing 3.51. However, holders provide additional functionality in fork-join codes. Namely, the view of a holder upon the first spawned child of a function (or the first child spawned after a sync) is the same as upon the entry to the function, even if a different worker is executing the child. This functionality allows to use holders as a replacement for argument passing. Unlike a truly shared variable, a holder has undetermined state in some cases (in spawned children after the first child, and in an arbitrary iteration of a `_Cilk_for` loop), because each strand manipulates its private view of the holder.

Listing 3.61 and Listing 3.62 demonstrate the use of a holder as a private variable. The purpose of a holder in this example is to reduce the number of times that the constructor of the scratch data class is called.

- In Listing 3.61, in the `_Cilk_for` loop, a separate copy of variable `scratch` is constructed for each iteration (6 times total). The cost of constructor `ScratchType` may be high. If that is the case, then the solution with a holder may improve efficiency.
- Listing 3.62 uses a holder for scratch data. It does so by wrapping `ScratchType` in the template class `cilk::holder`. In this implementation, the constructor of `ScratchType`) is called only once for each worker (2 times total). At the same time, the view of the variable `scratch` is undetermined in an arbitrary iteration of the loop.

Note that in the run performed in Listing 3.62, no work stealing occurred. If work stealing occurs, a new instance of `ScratchType` will be initialized for each stolen batch of iterations.

```
#include <cmath>
#include <cstdio>
#include <cilk/cilk_api.h>


const long N = 10000000L;
class ScratchType {
 long *data;
public:
 ScratchType(){
  data=new long[N];  data[0:N] = 0;
  printf("CONSTRUCTOR:worker %d\n",__cilkrts_get_worker_number());}
 ~ScratchType(){ delete(data); }
 long* Data() { return data; }
};

int main(){
 _Cilk_for (int i = 0; i < 6; i++) {
  ScratchType scratch;
  for (int j = 0; j < N; j++)
    scratch.Data()[j] += long(sqrt(j));
  printf("i=%d, worker=%d, value=%ld\n",
          i, __cilkrts_get_worker_number(), scratch.Data()[1]);
}}
```

```
vega@lyra% icpc -o Cilk-NoHolder Cilk-NoHolder.cc
vega@lyra% CILK_NWORKERS=2 ./Cilk-NoHolder
CONSTRUCTOR:worker 0
CONSTRUCTOR:worker 1
i=0, worker=0, value=1
i=3, worker=1, value=1
CONSTRUCTOR:worker 0
CONSTRUCTOR:worker 1
i=1, worker=0, value=1
i=4, worker=1, value=1
CONSTRUCTOR:worker 0
CONSTRUCTOR:worker 1
i=2, worker=0, value=1
i=5, worker=1, value=1
```

Listing 3.61: Compiling and running code `Cilk-NoHolder.cc`. Constructor of `ScratchType` is called for every loop iteration (a total of 6 times).

```
#include <cmath>
#include <cstdio>
#include <cilk/cilk_api.h>
#include <cilk/holder.h>

const long N = 10000000L;
class ScratchType {
 long *data;
public:
 ScratchType(){
  data=new long[N];  data[0:N] = 0;
  printf("CONSTRUCTOR:worker %d\n",__cilkrts_get_worker_number());}
 ~ScratchType(){ delete(data); }
 long* Data() { return data; }
};

int main(){
 cilk::holder<ScratchType> scratch;
 _Cilk_for (int i = 0; i < 6; i++) {
  for (int j = 0; j < N; j++) // Operator () is accessor to data:
    scratch().Data()[j] += long(sqrt(j));
  printf("i=%d, worker=%d, value=%ld\n",
          i, __cilkrts_get_worker_number(), scratch().Data()[1]);
}}
```

```
vega@lyra% icpc -o Cilk-Holders Cilk-Holders.cc
vega@lyra% CILK_NWORKERS=2 ./Cilk-Holders
CONSTRUCTOR:worker 0
CONSTRUCTOR:worker 1
i=0, worker=0, value=1
i=3, worker=1, value=1
i=1, worker=0, value=2
i=4, worker=1, value=2
i=2, worker=0, value=3
i=5, worker=1, value=3
vega@lyra%
```

Listing 3.62: Compiling and running `Cilk-Holders.cc`. Constructor of `ScratchType` is called once for every worker (a total of 2 times) rather than once for every iteration.

3.3.6. OpenMP versus Intel Cilk Plus

Intel Cilk Plus is an emerging standard currently supported by GCC 4.7 and the Intel C++ Compiler. Its functionality and scope of application are similar to those of OpenMP. There are only three keywords in the Cilk Plus standard: `_Cilk_for`, `_Cilk_spawn`, and `_Cilk_sync`. Programming for Intel Xeon Phi coprocessors may also require keywords `_Cilk_shared` and `_Cilk_offload`. However, these keywords allow to implement a variety of parallel algorithms. Language extensions such as array notation, hyper-objects, SIMD-enabled function and `#pragma simd` are also a part of Intel Cilk Plus. Unlike OpenMP, the Cilk Plus standard guarantees that serialized code will produce the same results as parallel code, if the program has a deterministic behavior. Last, but not least, Intel Cilk Plus is designed to seamlessly integrate vectorization and thread-parallelism in applications using this framework.

OpenMP and Cilk Plus have the same scope of application to parallel algorithms and similar functionality. The choice between OpenMP and Cilk Plus as the parallelization method may be dictated either by convenience, or by performance considerations. It is often easy enough to implement the code with both frameworks and compare the performance. Our experience shows that Intel Cilk Plus may have advantages over OpenMP in the case of complex algorithms with multi-level parallelism (see, e.g., [21]).

For complex algorithms with nested parallelism and heterogeneous tasks,

- Intel Cilk Plus generally provides good performance "out of the box", but offers little freedom for fine-tuning. With this framework, the programmer should focus on exposing the parallelism in the application rather than optimizing low-level aspects such as thread creation, work distribution and data sharing.
- OpenMP may require more tuning to perform well, however, it allows more control over scheduling and work distribution.

Intel OpenMP and Intel Cilk Plus libraries may be used side by side in the same code. In case of nested parallelism, it is preferable to use Cilk Plus parallel regions inside OpenMP parallel regions.

3.3.7. Additional Resources on Shared Memory Parallelism

We have provided a cursory overview of parallel programming in shared memory in two frameworks: OpenMP and Intel Cilk Plus. We focused on expressing and controlling task parallelism, and we left the discussion of optimization for the next chapter.

Understanding the methods and language extensions covered here are sufficient for leveraging the performance optimization examples in Chapter 4. In many real-world applications, this tool set will also be sufficient.

However, for readers wishing to continue studying OpenMP and Intel Cilk Plus, or to learn about other parallel frameworks and parallel programming, we provide a list of references below.

1) A hands-on video tutorial on OpenMP by Tim Mattson, Senior Research Scientist at Intel, is available at https://www-ssl.intel.com/content/www/us/en/education/university/intel-many-core-curriculum-list/openmp-videos.html.

2) A comprehensive description can be found in OpenMP specifications which can be found at the OpenMP Architecture Review Board Web site http://openmp.org/wp/openmp-specifications/.

3) A detailed written OpenMP tutorial from Blaise Barney of Lawrence Livermore National Laboratory is available at https://computing.llnl.gov/tutorials/openMP/.

4) Intel Cilk Plus pages in the Intel C++ Compiler reference provide details and examples for programming with this parallel framework.

5) The Intel Threading Building Blocks project (TBB) is another powerful parallel framework and library: http://threadingbuildingblocks.org. This product has an open-source implementation.

6) The book "Intel Xeon Phi Coprocessor High Performance Programming" by Jim Jeffers and James Reinders [3] (see also http://lotsofcores.com/).

7) The book "Structured Parallel Programming: Patterns for Efficient Computation" by Michael McCool, Arch D. Robinson and James Reinders [19] is a developer's guide to patterns for high-performance parallel

programming (see also http://parallelbook.com/). The book discusses fundamental parallel algorithms and their implementations in Intel Cilk Plus and TBB.

8) The book “Parallel Programming in C with MPI and OpenMP” by Michael J. Quinn [20] is full of examples of high performance applications implemented in OpenMP and MPI, illustrating the programming, optimization and benchmarking methodology studied in detail in this work.

3.4. Process Parallelism in Distributed Memory with MPI

At this point, we have discussed two levels of parallelism. Data parallelism in Intel Xeon family processors and in Intel Xeon Phi coprocessors is accessible via automatic vectorization by the compiler, array notation and, if necessary, intrinsic functions. Task parallelism in multi-core and manycore systems is harnessed through frameworks for multi-threading such as OpenMP and Intel Cilk Plus. The next level of parallelism is scaling an application across multiple compute nodes, i.e., in distributed memory. The most widely adopted HPC framework for distributed parallel applications is the Message Passing Interface (MPI). This section discusses expressing process parallelism with MPI.

3.4.1. Parallel Computing in Clusters with Multi-Core and Many-Core Nodes

MPI is a communication protocol. It allows multiple processes, which do not share common memory, but reside on common network, to perform parallel calculations, communicating with each other by way of passing messages. MPI messages are arrays of predefined and user-defined data types. The purpose of MPI messages is defined by the programmer. It may range from task scheduling to exchanging large amounts of data necessary to perform the calculation. MPI guarantees that the order of sent messages is preserved on the receiver side. The MPI protocol also provides error control. However, the developer is responsible for communication fairness control, as well as for task scheduling and computational load balancing.

Multiple implementations of MPI have been developed since the protocol's inception in 1991. In this training, we will be using the Intel MPI library version 5.0, which implements MPI version 3.0 specification. Intel MPI has native support for Intel Xeon Phi coprocessors, integrates with Intel software development tools, and operates with a variety of interconnect fabrics.

Originally, in the era of single-core compute nodes, the dominant MPI usage model in clusters was to run one MPI process per physical machine. With the advent of multi-core, multi-socket, and now heterogeneous systems, the range of usage models of MPI has grown as discussed below.

Heterogeneous MPI

One single-threaded MPI process can run on each physical core or logical processor of every compute node in the cluster (Figure 3.4). Intel Xeon Phi coprocessors in this case are treated as individual compute nodes.

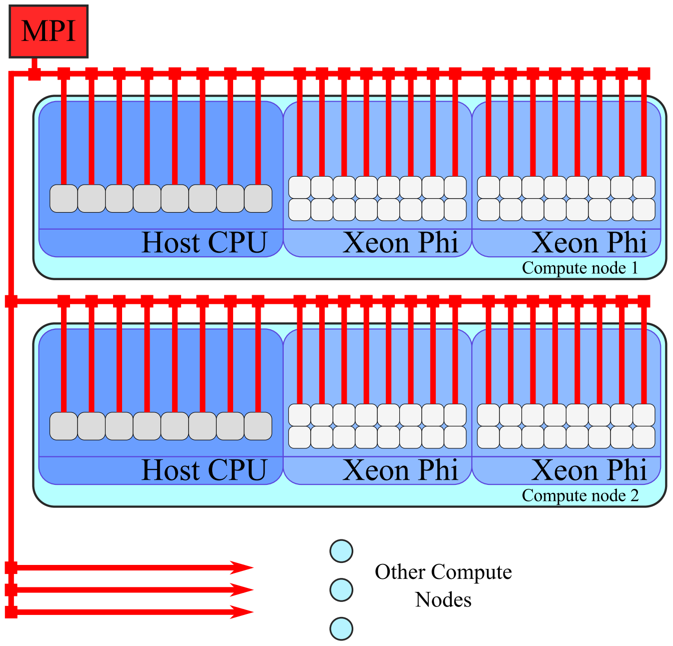


Figure 3.4: Pure MPI parallelism. One single-threaded MPI process per core.

In this case, MPI processes running on one compute node do not share memory address space. To share data, MPI processes have to send messages to each other. Message passing between these processes can be efficient, because fast virtual fabrics based on memory copy can be used for communication.

The drawback of this approach is that with up to 240 MPI processes on a single Intel Xeon Phi coprocessor, the communication load may be excessively high. Additionally, if processes access some shared read-only data, this data has to be replicated in each process, so the memory consumption may be unnecessarily large.

Hybrid MPI + OpenMP Approach

Alternatively, it is possible to run one MPI process per compute node (Figure 3.5), exploiting thread parallelism in each machine with a shared-memory parallel framework, such as OpenMP or Intel Cilk Plus (see Section 3.2). Again, coprocessors are treated as individual compute nodes.

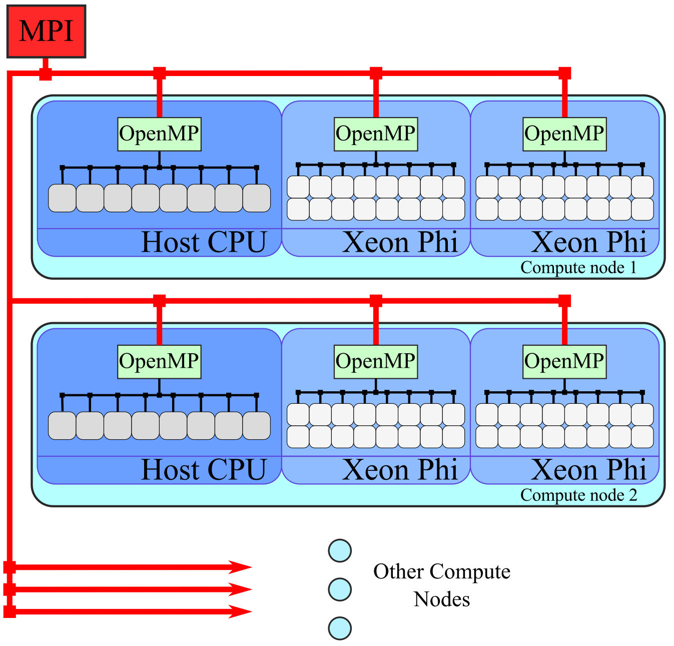


Figure 3.5: Hybrid MPI and OpenMP parallelism. One multi-threaded MPI process per node.

In this case, communication takes place only between different coprocessors, or between coprocessors and hosts. All threads within the OpenMP or Cilk Plus parallel region share the memory address space.

This hybrid MPI+OpenMP approach relaxes both the communication issues and the memory overhead of the MPI-only approach. On the other hand, the code inside the OpenMP region works with a larger data partition, and the programmer has to take care of sub-partitioning the problem in thread-parallel domain.

A variation of the hybrid MPI+OpenMP approach is to run *multiple* MPI processes per compute node, as shown in Figure 3.6.

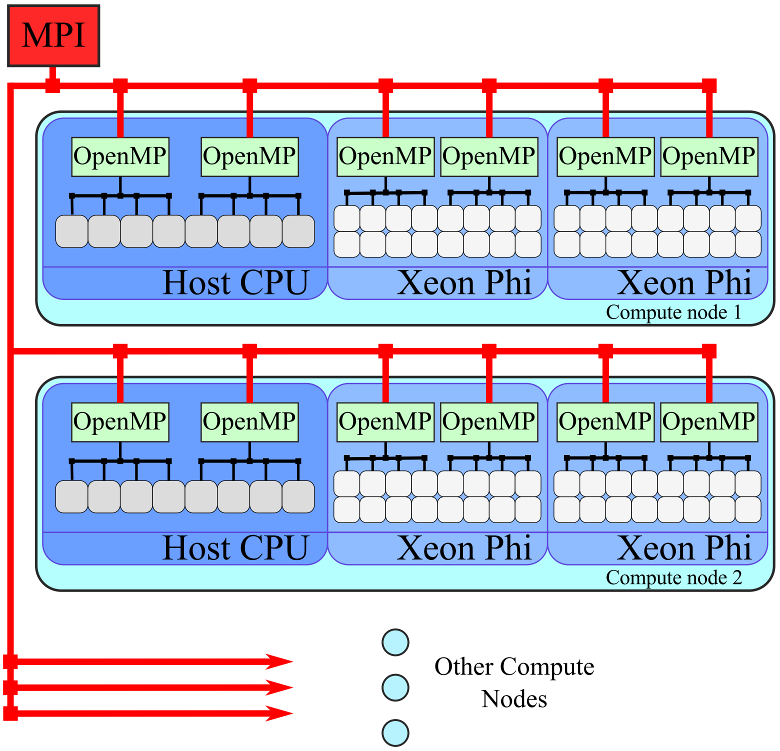


Figure 3.6: Hybrid MPI and OpenMP parallelism. Several multi-threaded MPI processes per node.

In this case, each process exploits parallelism in shared memory, and MPI communication between processes adds distributed-memory parallelism.

In many applications, the balance between thread and process parallelism (i.e., the number of OpenMP threads per MPI process) is a tuning parameter (see, e.g. [22]).

MPI with Offload

Finally, in heterogeneous clusters with Intel Xeon Phi coprocessors, MPI programmers have the option of running MPI processes only on hosts and performing offload to coprocessors (see Figure 3.7).

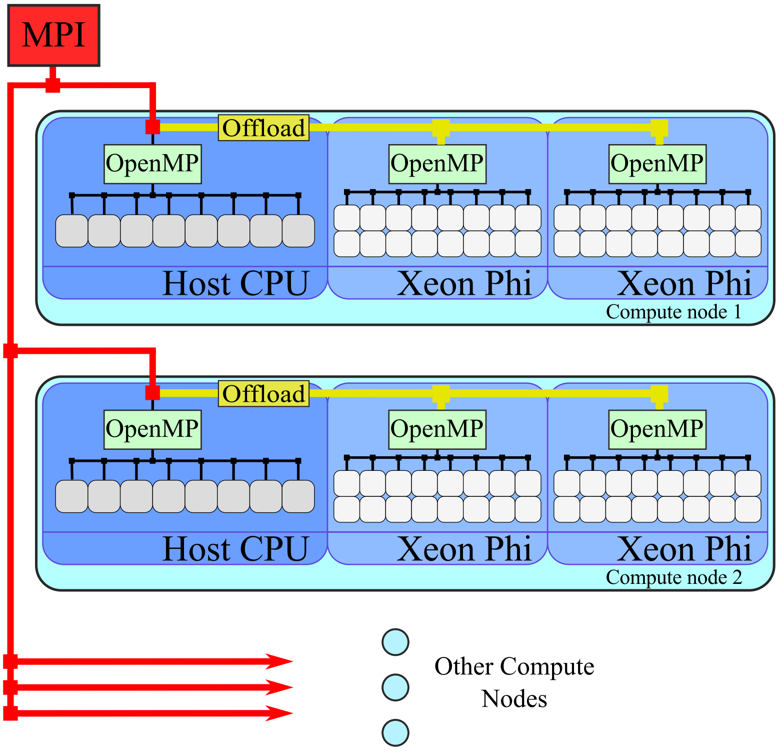


Figure 3.7: Hybrid MPI and OpenMP parallelism with offload from hosts to coprocessors.

The offload approach requires explicit enablement of the host code with data marshaling offload directives. However, it offers two advantages over native approaches (Figure 3.4, Figure 3.5 and Figure 3.6). First, it allows the MPI process to access a large amount of RAM that the host may have. Second, it offers a clearer programming model for applications that use CPUs for different purposes than coprocessors (for example, running serial operations or file I/O on hosts and computation on coprocessors).

Homogeneous MIC-hosted MPI

For all of the methods of scaling across clusters with MPI, it is possible to run calculations with MPI only on coprocessors (see Figure 3.8).

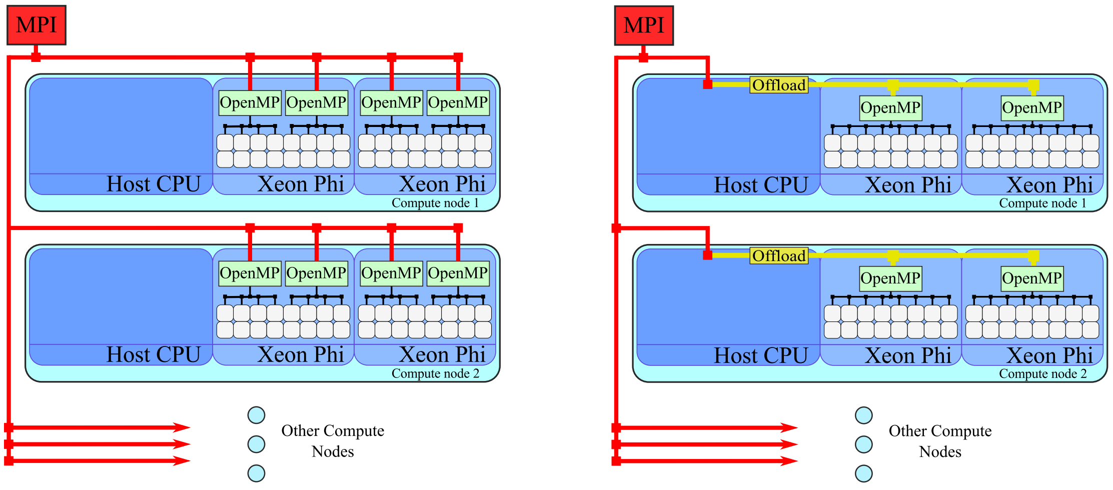


Figure 3.8: MIC-hosted execution in a cluster with coprocessors.

In this MIC-hosted execution environment, CPUs are used only to boot the coprocessors, operate network interconnects and, possibly, perform offload. This approach makes sense especially in computationally dense systems where CPUs contribute a small fraction to the total performance (e.g., CXP9000 in Figure 1.2). Because this system is homogeneous (all MPI processes deliver the same performance), the developer does not need to worry about load balancing, which would have been a difficult task in a heterogeneous system (see Section 4.7.1 and Section 4.7.2).

MIC-hosted execution with first generation Intel Xeon Phi coprocessors may also serve as a prototype for a cluster based on the socket version of the second generation Intel Xeon Phi processors.

3.4.2. Program Structure in MPI

Compiling and Running Applications

MPI applications in C, C++ and Fortran may be compiled with special wrappers over the respective compilers. The following commands invoke Intel compilers and automatically link the appropriate MPI libraries:

`mpiicc` for C language (`icc` is default compiler),

`mpiicpc` for C++ language (`icpc` is default compiler),

`mpiifort` for Fortran 77 and Fortran 95 (`ifort` is default compiler).

To run an MPI application, it must be executed with an MPI execution tool. Intel MPI contains a simplified script that starts MPI applications, called `mpirun`. This script accepts the list of hosts on which the application is executed, either as command line arguments, or in a machine file.

Typically, the same MPI application is launched on each MPI host. That is, each MPI host executes the same program. However, it does not mean that all processes perform the same work. At runtime, each MPI process is assigned a unique identifier called MPI rank. MPI ranks are integers that begin at 0 and increase contiguously. Using these ranks, processes can coordinate execution and identify their role in the application even before they exchange any messages. It is also possible to launch multiple executables on different hosts as a part of a single application. For complex applications, processes can be bundled into communicators and groups.

A "Hello World" MPI application was demonstrated in Chapter 2 in Section 2.1 and Section 2.4, and the reader is advised to refer to these sections to refresh this material.

Structure of MPI Applications

Listing 3.63 schematically demonstrates the structure of all MPI applications. We treat all code in this book as C++, however, the code below uses the MPI interface that works in C and C++.

```
#include "mpi.h"

int main(int argc, char** argv) {

  // Set up MPI environment
  int ret = MPI_Init(&argc,&argv);
  if (ret != MPI_SUCCESS) {
          MyErrorLogger("...");
          MPI_Abort(MPI_COMM_WORLD, ret);
  }

  int worldSize, myRank, myNameLength;
  char myName[MPI_MAX_PROCESSOR_NAME];
  MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Get_processor_name(myName, &myNameLength);

  // Perform work
  // Exchange messages with MPI_Send, MPI_Recv, etc.
  // ...

  // Terminate MPI environment
  MPI_Finalize();
}
```

Listing 3.63: Structure of an MPI application.

The code in Listing 3.63 illustrates the following rules:

- Header file `#include <mpi.h>` is required for all programs that make Intel MPI library calls.
- MPI calls begin with `MPI_`
- The MPI portion of the program begins with a call to `MPI_Init`

and ends with `MPI_Finalize`. No `MPI_*` calls are allowed prior to `MPI_Init` and after `MPI_Finalize` – that would cause the application to fail.

- Communicators can be used to address a substructure of MPI processes, and the default communicator `MPI_COMM_WORLD` includes all current MPI processes.
- Each process within a communicator identifies itself with a rank, which can be queried by calling the function `MPI_Comm_rank`
- The number of processes in the given communicator can be queried with `MPI_Comm_size`.
- Using the ranks and the world size, it is possible to distribute roles between processes in an application even before any messages are exchanged.
- Most MPI routines return an error code. The default MPI behavior is to abort program execution if there was an error.

3.4.3. Point-to-Point Communication

Now that we have created, compiled and executed a “Hello world” parallel MPI application (see Section 2.1 and Section 2.4), let us move on to passing messages between MPI processes.

In this section we will discuss only blocking communication routines. These routines pause the execution of a task until it is safe to re-use (for sending) or use (for receiving) the memory space holding the message. Non-blocking routines are discussed in Section 3.4.4.

At this point, we will only consider point-to-point communication, i.e., operations in which messages have only one source rank and one destination rank. In Section 3.4.5, we will also discuss collective communication, i.e., message exchange operations with more than one source or more than one destination.

Example

Listing 3.64 demonstrates the use of blocking point-to-point communication routines. In this code, multiple “worker” processes report to the “boss” process with rank equal to 0 (see Section 4.7.2 for an example of an application that uses the “boss-worker” model for load balancing).

Program shown in Listing 3.64 uses two functions that are new in our discussion: `MPI_Send` and `MPI_Recv`. These functions, respectively, send and receive a message. `MPI_Send` and `MPI_Recv` and their variations (discussed later) are the basis of MPI.

Note that the loop in line 13 goes through worker processes in order. Messages from these workers do not necessarily arrive in order, and therefore, the application may idle waiting for the message from a specific worker. In a practical application, other methods of communication may be employed to process messages that arrive, for example, from *any* source rather than a specific source.

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
 int i, rank, size, namelen;
 char name[MPI_MAX_PROCESSOR_NAME];
 MPI_Status stat;
 MPI_Init (&argc, &argv);
 MPI_Comm_size (MPI_COMM_WORLD, &size);
 MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 MPI_Get_processor_name (name, &namelen);
 if (rank == 0) {
  printf("Boss, rank %d of %d on %s\n", rank, size, name);
  for (i = 1; i < size; i++) {
   // Blocking receive operation in the boss process
   MPI_Recv(&rank, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &stat);
   MPI_Recv(&namelen, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &stat);
   MPI_Recv(name, namelen+1, MPI_CHAR, i, 1, MPI_COMM_WORLD, &stat);
   printf ("Received hello from worker %d running on %s\n",
                                                      rank, name);
  }
 } else {
  // Blocking send operations in all other processes
  MPI_Send (&rank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
  MPI_Send (&namelen, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
  MPI_Send (name, namelen + 1, MPI_CHAR, 0, 1, MPI_COMM_WORLD);
 }
 MPI_Finalize ();
}
```

```
vega@lyra% mpiicpc -o MPI-p2p MPI-p2p.cc
vega@lyra% mpirun -np 4 -host lyra ./MPI-p2p
Boss, rank 0 of 4 on lyra
Received hello from worker 1 running on lyra
Received hello from worker 2 running on lyra
Received hello from worker 3 running on lyra
```

Listing 3.64: Source code `MPI-p2p.cc` illustrates basic MPI communication.

Blocking Point to Point Message Passing Routines

The syntax of standard MPI routines for blocking point to point communication is described below.

MPI_Recv (&recv_buf, count, datatype, source, tag, comm, &status) is a basic blocking receive operation. It posts the intent to receive a message and blocks (i.e., waits) until the requested message is received into the receive buffer `recv_buf`.

MPI_Send (&send_buf, count, datatype, dest, tag, comm) is a basic blocking send operation. It sends the message contained in the send buffer `send_buf` and blocks until it is safe to re-use the send buffer.

Here and elsewhere, the meaning and type of common parameters are:

Type and Name	**Role**
`void* recv_buf`	Pointer to the received message data
`void* send_buf`	Pointer to the sent message data
`int count`	Number of elements in the send buffer
`MPI_Datatype datatype`	Indicates the type of data elements in the buffer. Table 3.6 lists predefined MPI data types
`int dest`	Rank of the process to which the message is sent
`int source`	Rank of the process from which a message is received. Special wild card value `MPI_ANY_SOURCE` allows to receive a message from any task
`int tag`	User-defined arbitrary non-negative integer assigned used to uniquely identify a message. Tag specified in a send operation must be matched in the corresponding receive operation. Special tag value `MPI_ANY_TAG` overrides this behavior, allowing to receive a message with any tag. According to the MPI standard, integers $0-32767$ can be used as tags. Depending on the implementation, the allowed range may be wider.
`MPI_Comm comm`	Communication context, or set of processes for which the source or destination fields are valid. `MPI_COMM_WORLD` is used to access all processes belonging to the current MPI application.
`MPI_Status* status`	Pointer to a structure containing the source, the tag and the length of the received message. To access the length from `status`, function `MPI_Get_count` must be used.

Table 3.5: Common MPI function arguments.

The data types supported by MPI are shown in Table 3.6. Note that user-defined data types can be created in MPI.

Required types	**Length, bytes**
MPI_PACKED	1
MPI_BYTE	1
MPI_CHAR	1
MPI_UNSIGNED_CHAR	1
MPI_SIGNED_CHAR	1
MPI_WCHAR	2
MPI_SHORT	2
MPI_UNSIGNED_SHORT	2
MPI_INT	4
MPI_UNSIGNED	4
MPI_LONG	4
MPI_UNSIGNED_LONG	4
MPI_FLOAT	4
MPI_DOUBLE	8
MPI_LONG_DOUBLE	16
MPI_CHARACTER	1
MPI_LOGICAL	4
MPI_INTEGER	4
MPI_REAL	4
MPI_DOUBLE_PRECISION	8
MPI_COMPLEX	2*4
MPI_DOUBLE_COMPLEX	2*8

Optional types	**Length, bytes**
MPI_INTEGER1	1
MPI_INTEGER2	2
MPI_INTEGER4	4
MPI_INTEGER8	8
MPI_LONG_LONG	8
MPI_UNSIGNED_LONG_LONG	8
MPI_REAL4	4
MPI_REAL8	8
MPI_REAL16	16

Table 3.6: Data types in required and recommended by the MPI standard. For a list of the types available in a specific MPI implementation, read the `<mpi.h>` file of that implementation.

Reliability, Order and Fairness

The MPI protocol provides reliable message transmission. This means that a sent message is always received correctly. If messages are sent over unreliable network layers (e.g., TCP/IP), then it is the job of the MPI implementation to ensure reliability at the MPI message level. At the same time, MPI does not provide any mechanisms available to the user for transmission error correction.

Additionally, MPI guarantees that messages will not overtake each other. Namely, if a single sender sends two messages, they will be received in the order that they were sent. If a single receiver posts receives for two messages, they will be satisfied in the order posted. Note that these rules do not apply if multiple threads in a host are performing communication.

On the other hand, MPI does not guarantee fairness in servicing connection attempts. For instance, if a task posts a receive from `MPI_ANY_SOURCE`, and two other tasks send messages with matching tags (see Figure 3.9), then only one of the sends will compete. There is no guarantee which send will complete. It is the programmer's responsibility to prevent such conflicts.

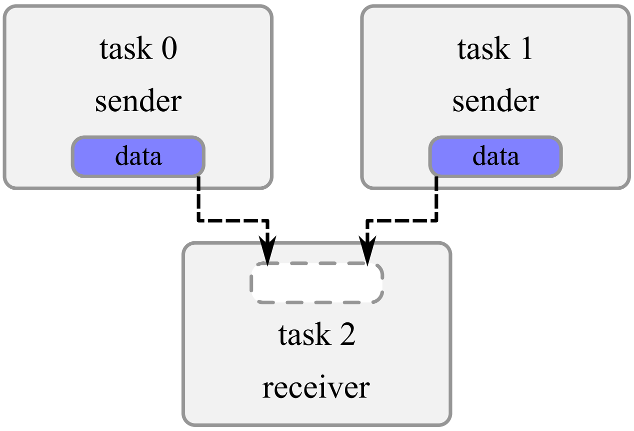


Figure 3.9: Illustration of MPI fairness conflict.

“Now you know your MPI"

Functions `MPI_Recv` and `MPI_Send` are easy to use, and they provide message passing functionality that is sufficient for many real-world HPC applications. That said, the discussion of parallelism in MPI could be terminated at this point. However, we will discuss additional topics of MPI in the rest of Section 3.4. Here is what to expect in the continuation of this discussion:

1. Buffering is a system-level functionality of MPI that enables significant optimization for communication efficiency. The use of buffering may require additional efforts to prevent errors. We discuss it in Section 3.4.4.

2. Non-blocking send and receive operations can be used to overlap computation and communication. This topic is also discussed in Section 3.4.4.

3. Collective communication is helpful for certain parallel patterns. In addition to convenience, collective communication offers performance optimizations and portability. See Section 3.4.5 for more information.

Finally, in this chapter we did not touch the issues of performance with MPI. This, along with other performance tuning questions, is left for Chapter 4 (Section 4.7).

3.4.4. MPI Communication Modes

Functions `MPI_Send` and `MPI_Recv` are the "standard" functions for sending and receiving messages. MPI implementations optimize these functions for a balance between efficiency and consistency. The expected behavior is that these functions take as little time as possible, and yet, other send and receive operations may be safely called after `MPI_Send` or `MPI_Recv` complete. However, there are other flavors of send and receive operations available to users who wish to fine-tune their applications, as discussed below.

Terminology: Application (Send) Buffer, System Buffer and User Space Buffer

Historically, the word "buffer" in the context of MPI is used in multiple terms with very different meaning. It is important to understand the difference between these terms for future discussion.

a) **Application buffer** collectively refers to send buffers and receive buffers. This is a memory region in the user application which holds the data of the sent or received message. In Table 3.5, the variable `void *buf` represents either the send, or the receive buffer. In the code in Listing 3.64, the role of send and receive application buffers is played by variables `rank`, `namelen` and `name`.

b) **System buffer** is a memory space managed by the MPI runtime library, which is used to hold messages that are pending for transmission on the sender side, or for reception to the application on the receiver side. The purpose of the system buffer is to enable asynchronous communication. The system buffer is not visible to the programmer. System buffers in MPI may exist on both the sender side and the receiver side. The standard functions `MPI_Send` and `MPI_Recv` typically use system-level buffers provided and managed by the MPI runtime library.

c) **User space buffer** plays the same role as the system buffer: it can temporarily store messages to enable asynchronous communication. However, this special buffer space is allocated and managed by the user and can only be used in specialized buffered send functions.

Terminology: Synchronous and Asynchronous Communication

In this discussion, we will be using the terms *synchronous* and *asynchronous* communication modes and the terms *blocking* and *non-blocking operations*. In MPI, these pairs of terms are not synonymous. It may further add to the confusion that the meaning of synchronous and asynchronous in MPI is different from that in the offload programming model for Intel Xeon Phi coprocessors. Let us clarify these terms before discussing specific MPI communication modes.

a) **Synchronous communication** means that the sender must wait until the corresponding receive request is posted by the receiver. After a "handshake" between the sender and receiver occurs, the message is passed without buffering. This mode is more deterministic and uses less memory than asynchronous communication, but at the cost of the time lost for waiting.

b) **Asynchronous communication** in the case of sending means that the sender does not have to wait for the receiver to be ready. The sender may put the message into the system buffer (either on the sender, or on the receiver side) or into the user space buffer, and return.

Terminology: Blocking and Non-Blocking Functions

Another concept in MPI is blocking and non-blocking functions.

a) **Blocking** send functions pause execution until it is safe to modify the current send buffer. Blocking receive functions wait until the message is fetched into the receive buffer.

b) **Non-blocking** send functions return immediately and execute the transmission "in background". Non-blocking receive functions only post the intent to receive a message, but do not pause execution. It is not safe to re-use or modify the send buffer before ensuring that a non-blocking send operation has completed. Likewise, it is unsafe to read from the receive buffer before ensuring that a non-blocking receive operation has completed. To ensure that a non-blocking operation is complete, each non-blocking MPI function must have a corresponding `MPI_Wait` or `MPI_Test` function.

Blocking and non-blocking functions exist in synchronous as well as asynchronous flavors.

Terminology: Ready Communication

If the programmer can guarantee that by the time that a send function is called, a matching receive is already pending, it is possible to use the *ready mode* send. This eliminates the hand-shake and may accelerate communication. If the receive is not posted, this is an error condition and the whole application must be aborted. There are both blocking and non-blocking ready mode functions. Any mode of the send function can be paired with any mode of the receive function.

Explanation of Communication Modes

To better illustrate synchronous and asynchronous, blocking and non-blocking, and ready mode functions, consider this real-world analogy. Suppose the sender (let us call her Sierra) wants to communicate to the receiver (let us call him Romeo) the time and place of their lunch meeting. The following situations are equivalent to the various communication modes in MPI:

1) **Blocking asynchronous send**: Sierra dials Romeo's telephone number and leaves a message on Romeo's answering machine. Sierra does not return to her activities until she had left the message. This reflects the blocking nature of this transaction. At the same time, after the transaction is complete, there is no guarantee that Romeo has personally received the message. This reflects the asynchronous nature of the transaction. The answering machine plays the role of a receiver-side system buffer in this case.

2) **Blocking synchronous send**: Sierra keeps dialing Romeo's telephone number until Romeo personally picks up the phone. This transaction is blocking, because Sierra cannot return to her other activities until she speaks to Romeo. This transaction is synchronous because at the end of the transaction, Romeo has definitely received the message.

3) **Non-blocking asynchronous send**: Sierra tells her assistant to call Romeo and leave a message on his answering machine. Sierra returns to her other activities immediately, so this transaction is non-blocking. Another property of non-blocking transactions: Sierra must wait for her assistant to finish with this task before assigning him another task (in MPI, after a non-blocking send, it is not safe to re-use the application send buffer before ensuring that the send has completed). Her assistant does not have to reach Romeo personally; leaving the message on the answering machine is satisfactory in this case, so this transaction is asynchronous.

4) **Non-blocking synchronous send**: Sierra tells her assistant to call Romeo, and to make sure to talk to him personally, and not to his answering machine. Sierra can do other things while her assistant works on transmitting the message, so this is a non-blocking transaction. This non-blocking transaction is synchronous, because after the assistant has finished with this task, Romeo is sure to have received the message.

5) **Blocking ready mode send**: Romeo is already on hold on Sierra's phone line when Sierra picks up the phone (ready mode). Sierra returns to her other activities only after she had transmitted her message (blocking transaction).

6) **Non-blocking ready mode send**: Romeo is already on hold on Sierra's phone line (ready mode), but she re-directs him to her assistant to relay the message. Sierra returns to her other activities immediately, so this is non-blocking communication.

Summary of Communication Modes

Function	Effect	Use Scenarios
MPI_Send	Blocking send operation. Synchronous or asynchronous depending on MPI implementation and runtime conditions. Returns when it is safe to re-use the application send buffer.	Default blocking send operation.
MPI_Bsend, MPI_Buffer_attach, MPI_Buffer_detach	Blocking asynchronous send operation with user space buffer. Returns when it is safe to re-use the application send buffer. User space buffer must be allocated with MPI_Buffer_attach prior to calling MPI_Bsend.	Used for asynchronous blocking communication when system buffer is inefficient, prone to overflows, or is not used by MPI_Send.
MPI_Ssend	Blocking synchronous send operation. Not buffered. Returns when it is safe to re-use the application send buffer.	Used (a) when message must be received before function return, or (b) to eliminate memory overhead of system or user space buffers.
MPI_Rsend	Blocking ready mode send operation. Synchronous or asynchronous depending on the MPI implementation and runtime conditions. Returns when it is safe to re-use the application send buffer. Assumes that the matching receive had already been posted, error otherwise.	Used in codes with fine-grained communication to improve performance. It is programmer's responsibility to ensure that matching receives post before MPI_Rsend.
MPI_Recv	Blocking receive operation.	Can be paired with any send operation.
MPI_Isend	Non-blocking send operation. Synchronous or asynchronous depending on the MPI implementation and runtime conditions. MPI_Wait must be called prior to re-using the application send buffer.	Default non-blocking send operation. Used to overlap communication and computation between MPI_Isend and MPI_Wait.
MPI_Ibsend, MPI_Buffer_attach, MPI_Buffer_detach	Non-blocking asynchronous send operation with user space buffer. MPI_Wait must be called prior to re-using the application send buffer. User space buffer must be allocated with MPI_Buffer_attach prior to calling MPI_Bsend.	Potentially the most efficient send method. Asynchronous and non-blocking, allows to overlap computation and communication. See also comment for MPI_Bsend.
MPI_Issend	Non-blocking synchronous send operation. Not buffered. MPI_Wait must be called prior to re-using the application send buffer.	Used to overlap communication and computation. At the same time, eliminates memory overhead of system or user space buffers.
MPI_Irsend	Non-blocking ready send operation. Synchronous or asynchronous depending on the MPI implementation and runtime conditions. MPI_Wait must be called prior to re-using the application send buffer. Assumes that the matching receive had already been posted, error otherwise.	Used instead of MPI_Isend in codes with fine-grained communication to improve performance by eliminating "handshakes".
MPI_Irecv	Non-blocking receive operation. MPI_Wait must be called prior to using the application receive buffer.	Can be paired with any send operation. Used to overlap computation and communication between MPI_Irecv and MPI_Wait.
MPI_Wait, MPI_Waitall MPI_Waitany, MPI_Waitsome	Blocks execution until one or more matching non-blocking send or receive operations return. After that, it is safe to re-use the application send buffer or use the application receive buffer.	Every asynchronous operation must have a matching MPI_Wait.

Table 3.7: Basic MPI functions. See MPI documentation for more details

Example: Blocking Asynchronous Send with User Space Buffer

Functions `MPI_Send` and `MPI_Recv` demonstrated in Listing 3.64 are the standard blocking send and receive operations in MPI. The runtime MPI library decides how to use the system buffer and whether to perform asynchronous transfers. For applications with large and frequent data transfers, the user may wish to take control over buffering in order to ensure that (a) buffering is used consistently and therefore all transactions are asynchronous, and (b) system buffer does not overflow. To let the user control over buffering of blocking transactions, MPI provides function `MPI_Bsend`. Listing 3.65 demonstrates how this function is used with user space buffer.

The required size of the buffer is calculated using the MPI function `MPI_Pack_size`. A constant `MPI_BSEND_OVERHEAD` is added to the buffer size. In our code, we intend to use the buffer for two send operations, so the overhead is counted twice.

```
#include <mpi.h>
#include <cstdio>
#include <cstdlib>

int main (int argc, char *argv[]) {
 const int M = 100000, N = 200000;
 float data1[M]; data1[:]=1.0f; // Any type of data can use
 double data2[N]; data2[:]=2.0; // the user space buffer
 int myRank, worldSize, size1, size2;

 MPI_Init (&argc, &argv);
 MPI_Comm_size (MPI_COMM_WORLD, &worldSize);
 MPI_Comm_rank (MPI_COMM_WORLD, &myRank);

 if (worldSize > 1) {
  if (myRank == 0) {
   // Sender: allocate user-space buffer for asynchr communication
   MPI_Pack_size(M, MPI_FLOAT, MPI_COMM_WORLD, &size1);
   MPI_Pack_size(N, MPI_DOUBLE, MPI_COMM_WORLD, &size2);
   int bufsize = size1 + size2 + 2*MPI_BSEND_OVERHEAD;
   printf("Buffer size: %d=%d+%d+2*%d bytes\n",
                    bufsize, size1, size2, MPI_BSEND_OVERHEAD);
   void* buffer = malloc(bufsize);
   MPI_Buffer_attach(buffer, bufsize);
   MPI_Bsend(data1, M, MPI_FLOAT, 1, 1, MPI_COMM_WORLD);
   MPI_Bsend(data2, N, MPI_DOUBLE, 1, 1, MPI_COMM_WORLD);
   MPI_Buffer_detach(&buffer, &bufsize);
   free(buffer);
  } else if (myRank == 1) {
   // Receiver side does not have to do anything special
   MPI_Status stat;
   MPI_Recv(data1, M, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &stat);
   MPI_Recv(data2, N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &stat);
  } }
 MPI_Finalize (); }
```

```
vega@lyra% mpiicpc -o MPI-Buffered MPI-Buffered.cc
vega@lyra% mpirun -np 2 -host lyra ./MPI-Buffered
Buffer size: 2000190=400000+1600000+2*95 bytes
```

Listing 3.65: Source code `MPI-Buffered.cc` illustrates blocking asynchronous transactions with user space buffer.

Example: Non-Blocking Standard Send

To overlap communication and computation, MPI provides non-blocking send and receive functions `MPI_Isend` and `MPI_Irecv`. Non-blocking functions return immediately, and the code can perform other operations while communication proceeds. However, for non-blocking send operations, it is unsafe to re-use the send buffer until blocking function `MPI_Wait` is called. Likewise, for non-blocking receive operations, it is unsafe to assume that the message was delivered to the receive buffer until `MPI_Wait` is called.

Calling `MPI_Wait` immediately after `MPI_Isend` or `MPI_Irecv` is equivalent to using `MPI_Send` or `MPI_Recv`. The purpose of non-blocking functions is to enable the code to perform some additional operations while the message is in transit.

Usually, when communicating processes use a network interconnect (e.g., Ethernet or InfiniBand) as physical network fabric, communication does not stall calculations. In this case, non-blocking communication may improve performance by masking communication time. However, if the sender and receiver are executing on the same host, their communication may proceed over a shared-memory copy. In this case, non-blocking communication may be detrimental to performance, because communication and computation will compete for resources, resulting in undesirable contention.

Listing 3.66 demonstrates the use of the non-blocking send operation.

```
#include <mpi.h>
#include <cstdio>

int main (int argc, char *argv[]) {
 const int N = 100000, tag=1;
 float data1[N], data2[N]; data1[:]=0.0f;
 int myRank, worldSize;

 MPI_Request request;
 MPI_Status stat;

 MPI_Init (&argc, &argv);
 MPI_Comm_size (MPI_COMM_WORLD, &worldSize);
 MPI_Comm_rank (MPI_COMM_WORLD, &myRank);

 if (worldSize > 1) {
  if (myRank == 0) {
   // Sender side: starting non-blocking send of data1
   MPI_Isend(data1, N, MPI_FLOAT, 1, tag, MPI_COMM_WORLD, &request);
   // Sender can perform some other work while data1 is in transit
   for (int i = 0; i < N; i++)
     data2[i] = 1.0f;
   // MPI_Wait will block until it safe to modify data1
   MPI_Wait(&request, &stat);
  } else if (myRank == 1) {
   // Receiver side: blocking receive of data1
   MPI_Recv(data1, N, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &stat);
   // At the end of blocking MPI_Recv, it is safe to use data1
  }
 }

 MPI_Finalize ();
}
```

Listing 3.66: Source code `MPI-NonBlocking.cc` illustrates non-blocking standard send.

3.4.5. Collective Communication and Reduction

Intel MPI collective communication routines involve *all* processes in the scope of a communicator. This is in contrast with peer-to-peer communication routines, which include two processes.

There are three major types of collective communication routines:

i) **Synchronization** — all processes wait until each of them has reached a synchronization point;
ii) **Data Movement** — broadcast, scatter and gather, all-to-all;
iii) **Collective Computation (reduction)** — multiple members of the group collect data from multiple other members and perform an associative operation (min, max, add, multiply, etc.) on that data.

MPI collective communication functions are summarized in Table 3.9.

Some of the most commonly used collective communication patterns are discussed later in this section.

Collective Communication and Performance

The usage of collective communication offers multiple advantages over instrumenting the same patterns with `MPI_Send`/`MPI_Recv`:

1. Vendor-optimized MPI libraries (for example, Intel MPI) provide the best performance of collective routines for the architectures and interconnects that they target.

2. Using collective communication routines, the developer delegates to MPI the handling of the complexity of network topology.

3. By extension, reliance on collective communication routines provides performance portability across different computing systems.

4. Collective communication algorithms enacted by Intel MPI may be tuned to the specific network topology of the cluster using the tool mpitune.

Broadcast

The broadcast operation, illustrated in Figure 3.10, sends a message from a single process to all processes in the communicator of this operation (including the sender). All processes in the communicator must post the broadcast in order for this operation to succeed. This operation is implemented by the MPI function `MPI_Bcast`.

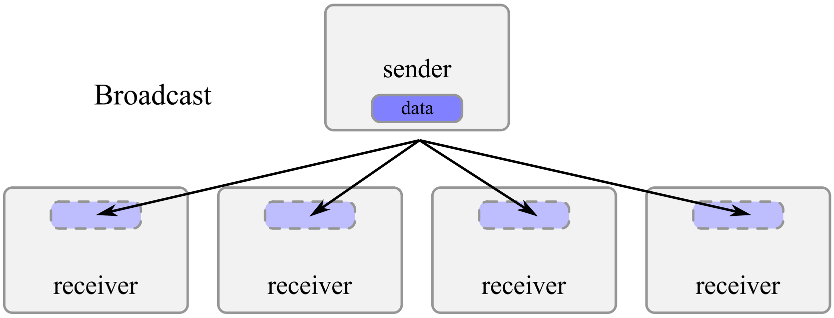


Figure 3.10: MPI collective communication: broadcast operation.

Note that in this and most other collective operations, only one line of code is necessary to execute communication. When all processes in the communicator of `MPI_Bcast` execute that line, the communication event will occur. This is in contrast with point-to-point communication, where at least two lines of code are necessary: one for the sender, another for the receiver.

Scatter and Gather

The scatter and gather operations (Figure 3.11 and Figure 3.12) also move data from/to a single source to/from multiple destinations. However, unlike broadcast, the scatter operation partitions a larger array of data and sends the respective pieces to the recipients of the scatter operation. Gather works similarly, but for receiving data, rather than sending it.

The MPI function for scatter is `MPI_Scatter`, and for gather, the function is `MPI_Gather`. There are also functions `MPI_Allgather` and `MPI_Allscatter` with similar patterns, in which each process receives the result of the operation.

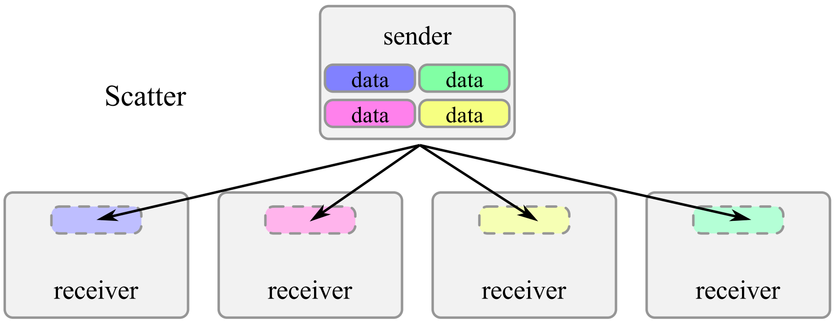


Figure 3.11: MPI collective communication: scatter operation.

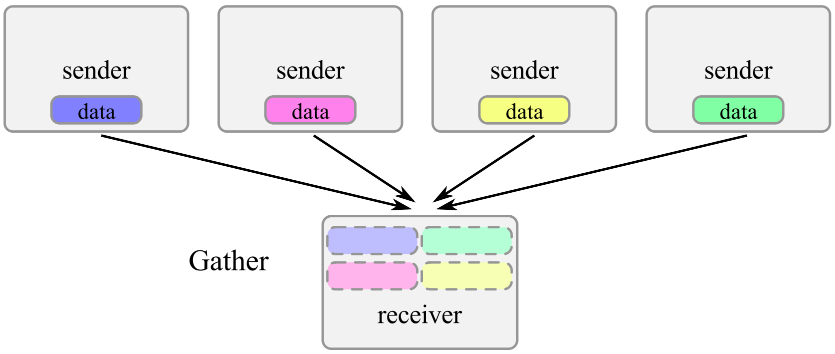


Figure 3.12: MPI collective communication: gather operation.

Reduction

Reduction is the application of an associative operation to data sets distributed across multiple processes. Figure 3.13 illustrates the pattern of a sum reduction. Note that the MPI and communication fabric do not necessarily implement the reduction operation with an "all-to-one" pattern: it may utilize smarter reduction mechanisms, such as tree patterns, to reduce the amount and increase the parallelism of communication.

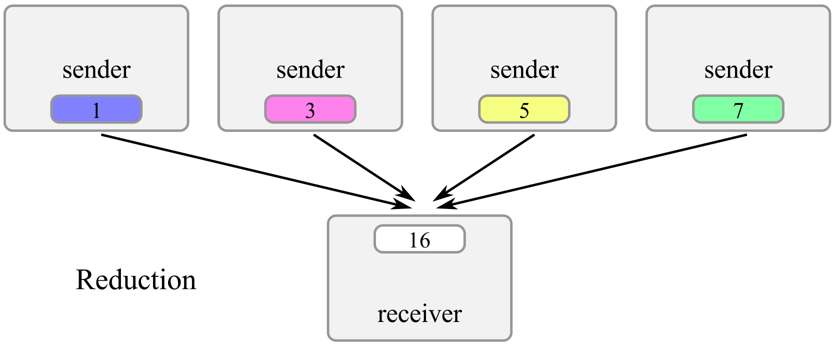


Figure 3.13: MPI collective communication: sum reduction.

Table 3.8 summarizes the reduction operations supported in MPI.

Intel MPI Reduction Operations		C Data Types	Fortran Data Types
`MPI_MAX`	maximum	integer, float	integer, real, complex
`MPI_MIN`	minimum	integer, float	integer, real, complex
`MPI_SUM`	sum	integer, float	integer, real, complex
`MPI_PROD`	product	integer, float	integer, real, complex
`MPI_LAND`	logical AND	integer	logical
`MPI_BAND`	bit-wise AND	integer, `MPI_BYTE`	integer, `MPI_BYTE`
`MPI_LOR`	logical OR	integer	logical
`MPI_BOR`	bit-wise OR	integer, `MPI_BYTE`	integer, `MPI_BYTE`
`MPI_LXOR`	logical XOR	integer	logical
`MPI_BXOR`	bit-wise XOR	integer, `MPI_BYTE`	integer, `MPI_BYTE`
`MPI_MAXLOC`	max value, location	float, double, long double	real, complex, double precision
`MPI_MINLOC`	min value, location	float, double, long double	real, complex, double precision

Table 3.8: MPI reduction operations

Function	Effect
`MPI_Barrier`	Performs group barrier synchronization. Upon reaching the `MPI_Barrier` call, each process is blocked until all processes in the group reach the same `MPI_Barrier` call.
`MPI_Bcast`	Broadcasts (i.e., sends) a message from one process to all other processes in the group.
`MPI_Scatter`	Distributes distinct messages from a single source process to each process in the group.
`MPI_Gather`	Gathers distinct messages from each process in the group into a single destination process.
`MPI_Allgather`	For each process, performs a one-to-all broadcasting operation within the group.
`MPI_Reduce`	Applies a reduction operation on all processes in the group and places the result in one process. Predefined MPI reduction operations are summarized in Table 3.8.
`MPI_Allreduce`	Applies a reduction operation and places the result in all processes in the group. This is equivalent to `MPI_Reduce` followed by `MPI_Bcast`.
`MPI_Reduce_scatter`	Performs an element-wise reduction on a vector across all processes in the group. The resulting vector is split into disjoint segments and distributed across the processes. This is equivalent to `MPI_Reduce` followed by `MPI_Scatter` operation.
`MPI_Op_create`	Creates a user-defined reduction operation for `MPI_Reduce` and `MPI_Allreduce`.
`MPI_Alltoall`	Each process in a group performs a scatter operation, sending a distinct message to all the processes in the group ordered by index.
`MPI_Scan`	Performs a scan operation with respect to a reduction operation across a process group.

Table 3.9: Collective communication functions in MPI. Details may be found in MPI Reference or by clicking function names.

Example: Using the Scatter Operation in MPI

The usage of one of the collective communication operations in OpenMP is shown in Listing 3.67.

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(int argc, char *argv[]) {
  int numtasks, rank, sendcount, recvcount, source;
  float sendbuf[SIZE][SIZE] = {
      {1.0, 2.0, 3.0, 4.0},
      {5.0, 6.0, 7.0, 8.0},
      {9.0, 10.0, 11.0, 12.0},
      {13.0, 14.0, 15.0, 16.0}};
  float recvbuf[SIZE];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  if (numtasks == SIZE) {
    source = 1;
    sendcount = SIZE;
    recvcount = SIZE;
    MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
                                 MPI_FLOAT,source,MPI_COMM_WORLD);
    printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
                               recvbuf[1],recvbuf[2],recvbuf[3]);
  } else {
      printf("Must specify %d processors. Terminating.\n",SIZE);
  }

  MPI_Finalize();
}
```

Listing 3.67: Source code `MPI-Scatter.cc` demonstrates one of Intel MPI collective communication operations: a scatter operation on the rows of a two-dimensional array.

This code demonstrates how function `MPI_Scatter` is used to distribute (i.e., scatter) the rows of a matrix from one source process to all other processes. Execution of this application with 4 processes produces the following output:

```
vega@lyra% mpirun -n 4 ./MPIscatter
rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
```

Listing 3.68: Intel MPI scatter collective communication example output.

3.4.6. Further Reading

The following list suggests additional sources of information on expressing parallelism in MPI and using MPI implementations.

1) The official source of all information on MPI is the MPI Forum Web page http://www.mpi-forum.org/.

2) Intel MPI, as of version 5.0 uses MPI standard version 3.0. Documentation on MPI 3.0 is available from the MPI documents page at the MPI forum site.

3) A detailed list of resources on MPI (manuals, tutorials, white papers, etc.) compiled by Argonne National Laboratory (ANL) is available at http://www.mcs.anl.gov/mpi/

4) The MPICH Web site features a reference of all MPI routines, which includes syntax and specification. This reference can be found at the URL http://www.mpich.org/static/docs/latest/. All hyperlinks attached to MPI function names in the PDF version of this book point to that document.

5) For more information on Intel MPI version 5.0 refer to the Intel MPI reference guide. This manual contains information about the specifics of Intel's implementation of MPI.

6) A popular book on MPI and its inter-operation with OpenMP is "Parallel Programming in C with MPI and OpenMP" by Michael J. Quinn [20]

Chapter 4 contains more MPI examples and discusses performance analysis and optimization methods in distributed memory computing.

CHAPTER 4
Optimizing Parallel Applications

This chapter delves deeper into the issues related to the performance of parallel applications on Intel Xeon processors and Intel Xeon Phi coprocessors and provides practical examples of high performance codes. This chapter re-iterates on the skills and methods introduced in Chapter 3.

4.1. Optimization Roadmap for Intel Xeon Phi Coprocessors

Intel Xeon Phi coprocessors are massively parallel vector processors, and optimization areas for them are qualitatively the same as those for Intel Xeon processors: vectorization, thread parallelism and memory traffic control. However, these requirements are *quantitatively* more strict for Intel Xeon Phi coprocessor applications: the code must be able to utilize wider vectors, support a greater number of threads, and the penalty for non-local memory access is greater on the coprocessor.

4.1.1. Optimization Checklist

In general, in order to expect better performance from an Intel Xeon Phi coprocessor than from the host system, the developer should be able to answer "*yes*" to the following questions:

1. **Has general optimization been performed?** Some applications can be improved by consistently employing single precision floating-point arithmetic instead of double or mixed precision, removing unnecessary type conversions, eliminating common sub-expressions, strength reduction, using transcendental functions supported by hardware, and choosing a compromise between precision and speed. Section 4.2 discusses these optimizations.

2. **Are performance-critical loops vectorized?** The compiler report should indicate that automatic vectorization of performance-critical loops has succeeded, otherwise a significant portion of the computational power of the Intel Xeon Phi coprocessor will be unused. In addition, the programmer must use data structures that enable unit-stride memory access pattern, enforce proper data alignment and inform the compiler of opportunities for choosing the optimal vectorization code path. See Section 4.3 for details.

3. **Does the applications scale beyond 100 threads?** Some applications designed for earlier generation processors may be serial or scale up to only a few threads. These applications will not show satisfactory performance on Intel Xeon Phi coprocessors, which derive performance from concurrent work of over 200 hardware threads on low clock-speed cores. Even if an application can utilize hundreds of threads, thread contention due to excessive synchronization or false sharing can quench performance. Methods for improving the parallel scalability of task-parallel calculations are described in Section 4.4.

4. **Is the workload arithmetically intensive or bandwidth-limited?** Applications that are not optimized for data locality in space and time, and programs with complex memory access patterns may exhibit better performance on a CPU than on an Intel Xeon Phi coprocessor. If complex memory access is an inherent property of the algorithm, it may be possible to re-structure data to pack memory accesses more compactly. Some algorithms must be modified with techniques such as loop tiling and cache-oblivious recursion to better utilize the cache hierarchy. These optimizations are described in Section 4.5.

5. **Is communication between the host(s) and the coprocessor(s) efficient?** An application that utilizes more than one coprocessor or more than one compute node must control the efficiency of data movement between the host system(s) and coprocessor(s). It may be possible to reduce communication overhead by overlapping communication with computation, optimizing data marshaling policies and relying on high-performing communication fabrics. These optimizations are described in Section 4.6 and Section 4.7.

4.1.2. Expectations

It is often the case that an application providing satisfactory performance on Intel Xeon processors initially performs poorly on Intel Xeon Phi coprocessors. This does not necessarily mean that the problem is not "MIC-friendly". Intel Xeon processors have resource-rich architecture with large caches, aggressive hardware prefetchers, branch predictors, out-of-order cores, deep pipelines and high clock speeds, which can compensate sub-optimal aspects of a variety of workloads. On the other hand, on Intel Xeon Phi coprocessors, the same sub-optimal behaviors of non-optimized applications are exposed by the resource-efficient MIC architecture. Thus, some optimizations may be required before an application provides satisfactory performance on the MIC architecture.

However, the good news is that, generally, optimization for Intel Xeon processors leads to performance benefits for Intel Xeon Phi coprocessors, and vice versa. In fact, one of the most effective methods for optimizing an application for the MIC architecture is to optimize it for the multi-core architecture first. Optimization methods described in this Chapter yield performance benefits for both the manycore and the multi-core architecture.

In the best case scenario, a single Intel Xeon Phi coprocessor is capable of outperforming two Intel Xeon processors system by a factor of 2x to 3x. This estimate is based on the theoretical peak arithmetic performance and memory bandwidth of a single Intel Xeon Phi coprocessor compared to those of a two-way Intel Xeon processor-based host system with the Ivy Bridge microarchitecture. The reason for comparing one coprocessor to two CPU sockets is that they have approximately the same thermal design power. However, applications that do not achieve this speedup can still benefit from an Intel Xeon Phi coprocessor in the system, because available programming models allow to team up the host and the coprocessor using asynchronous offload or heterogeneous MPI (see [23]).

4.1.3. Benchmark Methodology

In this Chapter, in order to illustrate performance gains due to optimization, we will often benchmark applications before and after an optimization, and also benchmark them on a general-purpose Intel Xeon processor and on an Intel Xeon Phi coprocessor. Unless stated otherwise, the benchmarks adhere to the following rules:

1. For benchmarking an entire application, we may use the Linux utility `time`. For convenience, we set up a Bash script `tmr` to display the timing information in user-friendly format. In future listings we assume that the script shown in Listing 4.1 is at a location defined by `PATH`.

```
#!/bin/bash
# Run and time program, direct output of time into tmr.tmp
(time ( $@ 2>&1 )) 2>tmr.tmp
# Process tmr.tmp and print human-friendly result
t=`tail -n 3 tmr.tmp | head -n 1 | tr ms ' '`
echo $t | awk '{printf "Time: %.3f s\n", $2*60+$3}'
# Clean up
rm -f tmr.tmp
```

Listing 4.1: Bash script `tmr` for benchmarking the wall clock time of application execution. This script is assumed to be in the user's `PATH` in all future listings.

Listing 4.2 illustrates the usage of `tmr`.

```
vega@lyra% tmr ./myapp
This is the output of myapp on CPU...
Time: 2.103 s
vega@lyra% ssh mic0 tmr ${PWD}/myapp
This is the output of myapp on MIC...
Time: 1.119 s
```

Listing 4.2: In this example, we benchmark the wall clock time of execution of application `myapp`. This application takes 2.10 seconds on the host and 1.05 seconds on the coprocessor.

2. For benchmarking only performance-critical functions, we use OpenMP timing functions inside the code. The timing is performed multiple times, and the mean and standard deviation are computed. The first one or two iterations are not included in statistical analysis because various warm-up processes typically make these iterations slower than the sustained performance. The execution time is usually translated to performance in application-specific units (GB/s, GFLOP/s or other values). Listing 4.3 demonstrates the typical procedure.

```
// Conversion factor from time in seconds to performance in GFLOP/s
const float HztoGFLOPs =
                   20.0*1e-9*float(nParticles)*float(nParticles-1);
double rate = 0, dRate = 0; // Statistical averages
const int nTrials = 10;
const int skipTrials = 2; // Skip first two warm-up iterations
for (int trial = 1; trial <= nTrials; trial++) {

  const double tStart = omp_get_wtime(); // Start timing
  PerformComputation(...);
  const double tEnd = omp_get_wtime(); // End timing

  if (trial > skipTrials) { // Collect statistics
    const double perf = HztoGFLOPs/(tEnd - tStart);
    rate  += perf;      // Mean
    dRate += perf*perf; // Standard deviation
  }
}
rate /= (double)(nTrials-skipSteps);
dRate=sqrt(dRate/double(nTrials - skipTrials) - rate*rate);
printf("Average performance: %10.1f +- %.1f GFLOP/s", rate, dRate);
```

Listing 4.3: Benchmarking a performance-critical function in the code and computing statistics.

When we report performance in plots, normally we include the standard deviation (e.g, 250 ± 4 GFLOP/s). If the standard deviation is not included (e.g., 250 GFLOP/s), it means that the reported average is accurate to the last reported significant figure (i.e,. the standard deviation of 250 GFLOP/s is no greater than 1 GFLOP/s, and standard deviation of 60.4 GB/s is no greater than 0.1 GB/s).

4.1.4. Benchmark Computing System

Timing and performance results in examples in this chapter were obtained on a Colfax ProEdge™ SXP8600 server. It was configured as follows:

CPU: A two-way Intel E5-2697 V2 processor. This two-way CPU, based on the Ivy Bridge architecture, has 12 cores per socket, i.e., a total of 24 cores clocked at 2.70 GHz. The Intel hyper-threading technology is enabled, so we see a total of 48 logical processors. The Intel SpeedStep technology was disabled on the host to prevent its interference with benchmark results.

RAM: 128 GiB of DDR3 memory in 16 GiB modules at 1600 MHz.

Coprocessors: Four Intel Xeon Phi coprocessors 7120P. Each coprocessor, based on the Knights Corner architecture, has 61 cores at clocked 1.24 GHz. With 4 hardware threads per core, we see a total of 244 cores per coprocessor. Most benchmarks use only one of these four coprocessors. Power management functions: `cpufreq`, `corec6`, `pc3`, `pc6` and the ECC functionality were enabled on coprocessors.

Interconnects: Two Intel True Scale IBA7322 QDR host channel adapters (InfiniBand interconnects), one per CPU socket. These were used for MPI communication bandwidth tests. For tests with remote devices, the sytem was connected to an identical system via an Intel True Scale QDR switch.

The software configuration of the system includes:

OS: CentOS 7.0 Linux operating system with kernel 3.10.0-123.el7.x86_64.

Software development tools: Intel Parallel Studio XE 2015 Update 1 Cluster Edition (compiler version 15.0.1.133, MPI version 5.0.2.044, MKL version 11.2.1, VTune version 2015 update 1).

MIC Driver stack: MPSS version 3.4.1.

InfiniBand software: OFED-3.5-2-MIC software stack.

4.2. Scalar and General Optimizations

Before proceeding to the details of data- and task-parallel code optimization, it is useful to consider general optimizations that improve the performance of each parallel task. These code modifications will naturally translate to improve the performance of vectorized parallel applications. Optimizations discussed in this section work by reducing the *total number* and *computational cost* of operations.

4.2.1. Compiler Controls for Optimization

Intel Compilers can perform some of the optimizations described in this section automatically. However, it is important to know how to facilitate the compiler's work.

Optimization Level

Performance-critical code should be compiled with the optimization level `-O2` or `-O3`. A simple way to set a specific optimization level is to use the compiler argument `-O2` or `-O3`. This setting applies to the whole file being compiled. It is also possible to apply a specific optimization level to a single function within a file using `#pragma intel optimization_level`. Figure 4.4 illustrates these methods.

```
vega@lyra% icpc -o mycode -O3 source.cc
```

```
#pragma intel optimization_level 3
void my_function() {
  //...
}
```

Listing 4.4: Top: specifying the optimization level `-O3` as a compiler argument. The specified optimization level is applied to the whole source file. Bottom: specifying the optimization level `-O3` as a pragma. The optimization level specified in this way applies only to the statement following the pragma.

The default optimization level is `-O2`, which optimizes the application for speed. At this level, enabled optimization functions include: automatic vectorization, inlining, constant propagation, dead-code elimination, loop unrolling, and others. This is the recommended optimization level for most purposes.

The optimization level `-O3` enables more aggressive optimization than `-O2`. It includes all of the features of `-O2` and, in addition, performs loop fusion, block-unroll-and-jam, `if`-statement collapse, and others.

Generally, one can expect better performance with `-O3` than with `-O2`. At the same time, at `-O2`, the programmer can control certain optimization parameters (e.g., the loop unroll factor), however, at `-O3`, the compiler takes over this tuning. As a result, `-O3` may sometimes result in *worse* performance than `-O2`. Therefore, the general recommendation is to try compiling with both optimization levels. If some functions respond better to `-O2` and other work better with `-O3`, the programmer may use the pragma illustrated in Listing 4.4 for a more fine-grained approach.

Processor-Specific Optimization

In addition to the `-On` argument, the compiler can be instructed to target a specific processor using the argument `-x<code>`, where `<code>` is the one of the following:

`code`	Target architecture
MIC-AVX512	Future Intel processors
CORE-AVX512	Future Intel processors
CORE-AVX2	Intel Xeon processor E3 v3 family
CORE-AVX-I	Intel Xeon processor E3 v2, E5 v2 and E7 v2 family
AVX	Intel Xeon processor E3 and E5 family
SSE4.2	Intel Xeon processor 55XX, 56XX, 75XX and E7 family

Table 4.1: Some of the architectures that can be targeted with `-x<code>`.

With Intel Xeon Phi coprocessors, this argument has no effect. However, it is important for most Intel Xeon processors, and it also may be important for future generations of Intel MIC architectures. For a complete list of supported architectures, refer to the Intel Compiler Reference.

4.2.2. Compiler Controls for Precision

Floating-Point Semantics

The Intel C++ Compiler may represent floating-point expressions in executable code differently, depending on the *floating-point semantics*, i.e., rules for finite-precision algebra allowed in the code. These rules are controlled by an extensive set of command-line compiler arguments. The argument `-fp-model` controls floating-point semantics at a high level.

Table 4.2 explains the usage of the argument `-fp-model`. For more information, see the Compiler Reference and the white paper "Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?" by Dr. Martyn J. Corden and David Kreitzer [24].

In the context of floating-point semantics, "value-unsafe" optimizations refer to code transformations that produce only approximately the same result. For example, floating-point multiplication is generally non-associative in finite-precision arithmetics, i.e., `a*(b*c)`$\neq$ `(a*b)*c`. If value-unsafe optimizations are enabled, the compiler may replace an expression like `bar=a*a*a*a` with `foo=a*a; bar=foo*foo`. However, if only value-safe optimizations are allowed, then the expression will be computed from left to right, i.e., `bar=((a*a)*a)*a`. The two expressions produce approximately the same result, but the former employs one less operation.

Listing 4.5 and Listing 4.6 illustrate the usage of argument `-fp-model` to control the floating-point semantics. In this code Listing 4.5, a single floating-point number is subjected to an iterative procedure. It can be demonstrated analytically that this procedure has a stochastic character, i.e., small perturbations in initial conditions lead to large deviations in the result after several iterations. Listing 4.6 demonstrates that up to 20000 iterations, codes compiled with `-fp-model fast=1` and `-fp-model fast=2` produce identical results on an Intel Xeon Phi coprocessor. However, by iteration 25000, the results are completely different. This occurs because at iteration 23431 (as tested on our hardware), the two codes produce slightly different results due to different numerical accuracy, and this subtle difference is subsequently amplified by the stochastic iteration. Note that at the same time, the code compiled with `-fp-model fast=2` performs 1.5 times as fast as the code compiled with the default floating-point model.

Argument	Effect
`-fp-model strict`	Only value-safe optimizations, exception control is enabled (but may be disabled using `-fp-model no-except`), floating-point contractions (e.g., the fused multiply-add instruction) are disabled. This is the strictest floating-point model.
`-fp-model precise`	Only value-safe optimizations, exception control is disabled (but may be enabled using `-fp-model except`). Serial floating-point calculations are reproducible from run to run. Some parallel OpenMP calculations can be made reproducible by using the environment variable `KMP_DETERMINISTIC_REDUCTION`. The combination `-fp-model precise -fp-model source` produces floating-point results compliant with the IEEE-754 standard.
`-fp-model fast=1`	Value-unsafe optimizations are allowed, exceptions are not enforced, contractions are enabled. This is the default floating-point semantics model. The short-hand for this model is `-fp-model fast`.
`-fp-model fast=2`	Enables more aggressive optimizations than `fast=1`, possibly leading to better performance at the cost of lower accuracy.
`-fp-model source`	Intermediate arithmetic results are rounded to the precision defined in the source code. Using `source` also assumes `precise`, unless overridden by `strict` or `fast`.
`-fp-model double`	Intermediate arithmetic results are rounded to 53-bit (double) precision. Using `double` also assumes `precise`, unless overridden by `strict` or `fast`.
`-fp-model extended`	Intermediate arithmetic results are rounded to 64-bit (extended) precision. Using `extended` also assumes `precise`, unless overridden by `strict` or `fast`.
`-fp-model [no-]except`	`except` enables, `no-except` disables the floating-point exception semantics.

Table 4.2: Command-line arguments for high-level floating-point semantics control with the Intel C++ Compiler.

```
#include <cstdio>
#include <cmath>
int main() {
  for (int i = 0; i < 100; i++) {
    const int N=i*10000;
    double A = 0.1;
    for (int r = 0; r < N; r++)
      A = sqrt(1.0-4.0*(A-0.5)*(A-0.5));
    if (i<5) printf("After %5d iters, A=%.6f\n", N, A);
  }
}
```

Listing 4.5: Code `fp-model.cc` used in Listing 4.6 to illustrate the effect of relaxed floating-point model. The loop with the `sqrt()` function performs an iterative update of the value `A`.

```
vega@lyra% icpc -mmic \
> -fp-model fast=1 fpmodel.cc\
> -o slowcode
vega@lyra% ssh mic0 \
>           tmr ${PWD}/slowcode
After     0 iters, A=0.100000
After 10000 iters, A=0.633073
After 20000 iters, A=0.534324
After 30000 iters, A=0.513582
After 40000 iters, A=0.552932
Time: 3.588 s
```

```
vega@lyra% icpc -mmic \
> -fp-model fast=2 fpmodel.cc\
> -o fastcode
vega@lyra% ssh mic0 \
>           tmr ${PWD}/fastcode
After     0 iters, A=0.100000
After 10000 iters, A=0.633073
After 20000 iters, A=0.534324
After 30000 iters, A=0.244553
After 40000 iters, A=0.997650
Time: 2.411 s
```

Listing 4.6: Compiling and running the code illustrated in Listing 4.5 on an Intel Xeon Phi coprocessor. Test case shown in panel on the left uses default semantics, `-fp-model fast=1`. Case shown in the other panel uses relaxed semantics, `-fp-model fast=2`.

This example demonstrates how relaxing the floating-point model may lead to a significant performance increase on Intel Xeon Phi coprocessors. However, this is only safe in well-behaved, numerically stable applications.

Precision Control for Transcendental Functions

To use transcendental functions such as square root, trigonometric, logarithmic and exponential functions, the programmer must include the header file `math.h` (or `cmath` for C++). This causes linking to the Intel Math Library, which is a part of the Intel compiler distribution. Including these files makes the program compatible with the interface of the GCC math library `libm`, so that the code can also be compiled with a GNU C/C++ compiler. Alternatively, the programmer may include `mathimf.h`, which includes additional functions available only in the Intel Math Library, but makes the code incompatible with GNU compilers.

By default, the Intel C++ Compiler replaces calls to Intel Math Library functions with Intel Short Vector Math Library functions or to the processor vector instructions. It is possible to instruct the compiler to use low-precision implementations of these functions for some operations in order to gain more performance. Naturally, this must be done with care and only in "well-behaved" applications that can tolerate the imprecise results.

Table 4.3 summarizes the Intel C++ Compiler command line arguments for precision control. Of all the settings listed there, `-fimf-precision`, `-fimf-max-error` and `fimf-accuracy-bits` express the same requirements in different terms: as a grade of precision, as the maximum tolerable error, and as the required number of accurate bits. The setting `-fimf-domain-exclusion` is different: it determines whether special-value numbers (extremes, NaNs, infintes, denormals and zeroes) must be handled by the Intel Math Library functions or not.

The effect of transcendental function precision control may be different on different architectures. For instance, in the example demonstrated below, on an Intel Xeon processor, the argument `-fimf-precision` impacts the performance and precision of the result. The same code on an Intel Xeon Phi coprocessor produces the same results in the same amount of time regardless of the value of `-fimf-precision`.

For more information of the function precision control in Intel C++ Compiler see the Intel C++ Compiler Reference and the white paper "Advanced Optimizations for Intel MIC Architecture, Low Precision Optimizations" by Wendy Doerner [25].

Argument	Effect
`-fimf-precision=value[:funclist]`	Defines the precision for math library functions. Here, `value` is one of: `high`, `medium` or `low`, which correspond to progressively less accurate but more efficient math functions, and `funclist` is a comma-separated list of functions that this rule is applied to. Value `high` is equivalent to `max-error=0.6`, `medium` to `max-error=4` and `low` to `accuracy-bits=11` in single precision or `accuracy-bits=26` in double precision (see below). By default, this option is not specified, and the compiler uses default heuristics when calling math library functions. This is an aggregate compiler option;see `-fimf-max-error` and `-fimf-accuracy-bits` for fine-grained control.
`-fimf-max-error=ulps[:funclist]`	The maximum allowable error expressed in ulps (*units in last place*) [26]. Max error of 1 ulps corresponds to the last mantissa bit being uncertain; 4 ulps is three uncertain bits, etc. This is a more fine-grained method of setting accuracy than `-fimf-precision`.
`-fimf-accuracy-bits=n[:funclist]`	The number of correct bits required for mathematical function accuracy. The conversion formula between accuracy bits and ulps is: `ulps = ` $2^{p-1-bits}$, where p is 24 for single precision, 53 for double precision and 64 for long double precision (the number of mantissa bits). This is a more fine-grained method of setting accuracy than `-fimf-precision`.
`-fimf-domain-exclusion=n[:funclist]`	Defines a list of special-value numbers that do not need to be handled by the functions. Here, `n` is an integer derived by the bitwise OR of the following values: extremes: 1, NaNs: 2, infinites: 4, denormals: 8, zeroes: 16. For example, `n=15` indicates that extremes, NaNs, infinites and denormals should be excluded from the domain of numbers that the mathematical functions must correctly process.

Table 4.3: Intel C++ Compiler arguments for mathematical function precision control.

Listing 4.7 and Listing 4.8 illustrate math function precision control. The change of the precision of the exponential function from `high` to `low` results in almost a factor of 2 speedup. The results are different, and the difference can be detected from the 11th significant figure in the decimal representation. Assembly listing shows that the computation of the exponential was instrumented for `-fimf-precision=high` with function `__svml_exp_ha()` ("high accuracy"), and for `=low`, with function `__svml_exp_ep()` ("enhanced performance").

```
#include <cstdio>
#include <cmath>
int main() {
  const int N = 1000000;
  const int P = 10;
  double A[N];
  const double startValue = 1.0;
  A[:] = startValue;
  for (int i = 0; i < P; i++)
    for (int r = 0; r < N; r++)
      A[r] = exp(-A[r]);
  printf("Result=%.17e\n", A[0]);
}
```

Listing 4.7: Code `precision.cc` used in Listing 4.8 to illustrate the effect of relaxed transcendental function precision.

```
vega@lyra% icpc -o precision-2\
> -fimf-precision=high\
> precision.cc
vega@lyra% tmr ./precision-2
Result=5.68428725029060722e-01
Time: 0.073 s
```

```
vega@lyra% icpc -o precision-1\
> -fimf-precision=low\
> precision.cc
vega@lyra% tmr ./precision-1
Result=5.68428725010313829e-01
Time: 0.046 s
```

Listing 4.8: Compiling and running the code illustrated in Listing 4.7 on an Intel Xeon Phi coprocessor. with high precision (left) and low precision (right).

4.2.3. Optimizing Arithmetic Expressions

In some cases, the code may perform unnecessary calculations even though the programmer did not intend that. Removing those redundant operations, as outlined in this section, can benefit the application performance.

Precision of Constants and Variables

Intel Xeon Phi coprocessors have a theoretical peak performance of up to 1 TFLOP/s in double precision and up to 2 TFLOP/s in single precision. Therefore, for floating-point calculations, single precision floating-point numbers should be used instead of double precision wherever possible. Similarly, signed 32-bit integers should be preferred to unsigned and 64-bit integers, including array indices.

However, it is generally preferable to consistently use single precision or double precision throughout the application, as opposed to mixing them. Mixing single and double precision variables and constants leads to complications with vectorization, and often results in even lower performance than in double precision.

Declaring precision for variables in C and C++ is straightforward: type `float` is for single precision, `double` for double precision. There is support for 128-bit type `long double` in Intel Xeon Phi coprocessors, however, only for scalar operations. For integers, `int` is for 32-bit signed integers, `long` is for 64-bit signed integers. Type `long long` is equivalent to `long` (i.e., there is no 128-bit integer support).

For literal constants, the following conventions determine the type:

1. Constants without a decimal point or exponent or suffix have type `int`. Examples: `0`, `1`, `300`.
2. Constants without a decimal point or exponent but with suffix `l` or `L` have type `long`. Examples: `0L`, `1L`, `1000000000000L`.
3. Constants with a decimal point or exponent and without a suffix have type `double`. Examples: `0.0`, `1.0`, `1.0e100`.
4. Constants with a decimal point or exponent and with suffix `f` or `F` have type `float`. Examples: `0.0F`, `1.0F`, `1.0e10F`.
5. Constants with a decimal point or exponent and with suffix `l` or `L` have type `long double`. Examples: `0.0L`, `1.0L`, `1.0e1000L`.

These conventions are summarized in Table 4.4.

Type	Decimal Point	Exponent	Suffix	Example
`int`	no	no	none	`0`, `1`, `300`
`long`	no	no	`l` or `L`	`0L`, `1L`, `1000000000000L`
`double`	yes	yes	none	`0.0`, `1.0`, `1.0e100`
`float`	yes	yes	`f` or `F`	`0.0F`, `1.0F`, `1.0e10F`
`long double`	yes	yes	`l` or `L`	`0.0L`, `1.0L`, `1.0e1000L`

Table 4.4: Conventions for defining literal constants in C and C++.

It is important to follow the convention of suffixes to avoid accidental introduction of mixed precision in expressions. Listing 4.9 illustrates the good and bad practices for working with precision of constants and variables.

```
// Bad: 2 is "int"
long b=a*2;

// Bad: overflow
long n=100000*100000;

// Bad: excessive
float p=6.283185307179586;

// Bad: 2 is "int"
float q=2*p;

// Bad: 1e9 is "double"
float r=1e9*p;

// Bad: 1 is "int"
double t=s+1;
```

```
// Good: 2L is "long"
long b=a*2L;

// Good: correct
long n=100000L*100000L;

// Good: accurate
float p=6.283185f;

// Good: 2.0f is "float"
float q=2.0f*p;

// Good: 1e9f is "float"
float r=1e9f*p;

// Good: 1.0 is "double"
double t=s+1.0;
```

Listing 4.9: Controlling the implied precision of constants and functions.

Precision of Functions

Transcendental functions in the Intel Math Library have single precision and double precision implementations. Each function takes an argument and returns the result of the same data type. Double precision functions have names such as `sin()`, `exp()`, `fabs()`, etc., and the names of single precision versions are derived by adding the suffix `-f`, e.g., `sinf()`, `expf()`, `fabsf()`, etc. For a complete list of functions in the Intel Math Library, refer to the compiler reference.

There is no overload for Intel Math Library functions in C and in the global namespace in C++. That is, for `float x`, the function `sin(x)` does not fall back to the single precision implementation. However, when the namespace `std` is used, the functions become overloaded in C++. Mixing the precision of arguments and functions may lead to unintended type conversions. For instance, in order to compute `sin(x)`, the value of `x` will be converted to double precision, the sine function will be computed in double precision, and, depending on the usage of this function in the expression, the value of the sine may be down-converted again to single precision.

Listing 4.10 illustrates the proper and improper practices for using the Intel Math Library functions.

```
// Bad: 3.14 is a double
float x = 3.14;

// Bad: sin() is a
// double precision function
float s = sin(x)

// Bad: round() takes double
// and returns double
long v = round(x);

// Bad: abs() is not from IML
// it takes int and returns int
int v = abs(x);
```

```
// Good: 3.14f is a float
float x = 3.14f;

// Good: sin() is a
// single precision function
float s = sinf(x)

// Good: lroundf() takes float
// and returns long
long v = lroundf(x);

// Good: fabsf() is from IML
// It takes and returns a float
float v = fabsf(x);
```

Listing 4.10: Controlling precision of Intel Math Library functions and arguments.

The recommendations of this section are particularly important for the scenario when Intel Xeon Phi coprocessors are used for single precision calculations. In this case, careful precision control in C and C++ is necessary, because literal constants are interpreted as `double` without a suffix, and the default IML functions expect double precision arguments and return double precision values. It is easy to convince oneself of these facts by running the code shown in Listing 4.11.

```
#include <cmath>
#include <cstdio>

int main() {
  // Proof that exp() is not overloaded:
  printf("    exp (1.0f)=%18.16f\n", exp (1.0f));
  printf("    exp (1.0 )=%18.16f\n\n", exp (1.0));
  // Proof that expf() gives lower precision:
  printf("    expf(1.0f)=%18.16f\n", expf(1.0f));
  printf("    expf(1.0 )=%18.16f\n\n", expf(1.0));
  // Overload in namespace std:
  printf("std::exp(1.0f)=%18.16f\n", std::exp(1.0f));
  printf("std::exp(1.0 )=%18.16f\n", std::exp(1.0));
  printf("Exact:        e=2.7182818284590452353 6...\n\n");
}
```

```
vega@lyra% icpc -o Scalar-TestFOverload Scalar-TestFOverload.cc
vega@lyra% ./Scalar-TestFOverload
    exp (1.0f)=2.7182818284590451
    exp (1.0 )=2.7182818284590451

    expf(1.0f)=2.7182817459106445
    expf(1.0 )=2.7182817459106445

std::exp(1.0f)=2.7182817459106445
std::exp(1.0 )=2.7182818284590451
Exact:        e=2.71828182845904523536...
```

Listing 4.11: Verifying the defaults for precision of constants, variables and functions.

Strength Reduction

Strength reduction is an optimization method based on replacing expensive arithmetic operations with approximately equivalent less expensive operations.

One of the best known examples of strength reduction is the replacement of division with multiplication by the reciprocal of the denominator. It works because floating-point division is significantly more expensive than floating-point multiplication (see, e.g., [27] for benchmark results). This particular optimization is trivial in expressions involving constants. For example, `x*0.5` computes significantly faster than `x/2.0`. In loops, performance can be improved by precomputing the reciprocal of the denominator and multiplying by it, as illustrated in Listing 4.12.

Sub-optimal

```
// Expensive division in loop
for (int i = 0; i < n; i++)
  A[i] /= B;

// Two divisions
// are expensive
for (int i = 0; i < n; i++)
  P[i] = (Q[i]/R[i])/S[i];
```

Optimized

```
const double Br = 1.0/B;
for (int i = 0; i < n; i++)
  A[i] *= Br;

// One division and one
// multiplication is better
for (int i = 0; i < n; i++)
  P[i] = Q[i]/(R[i]*S[i]);
```

Listing 4.12: Strength reduction: replacement of division with multiplication.

In some cases, depending on floating-point semantics (see Section 4.2.2), the compiler can automatically perform strength reduction. However, doing it explicitly like in Listing 4.12 may improve cross-platform and cross-compiler portability of the code.

Strength reduction in expressions may take other forms. For instance, the programmer may simplify expressions to expose opportunities for hardware-supported transcendentals. Intel Xeon Phi coprocessor architecture supports operations such as the reciprocal square root and base 2 logarithms and exponentials. The programmer can take advantage of this functionality by performing strength reduction as illustrated in Listing 4.13.

Sub-optimal

```c
// No hardware support for pow
double r = pow(r2, -0.5);

// Natural base exponentials
// are expensive
float v = expf(x);

// Linear interpolation in
// log-log space (=power-law)
double y = y0*exp(log(x/x0)*
          log(y1/y0)/log(x1/x0));
```

Optimized

```c
// Hardware support for rsqrt
double r = 1.0/sqrt(r2);

// Base 2  exponentials
// are supported in hardware
float v = exp2f(x*1.442695f);

// The same result, faster
//
double y = y0*exp2(log2(x/x0)*
      log2(y1/y0)/log2(x1/x0));
```

Listing 4.13: Left: non-optimized code uses functions not supported in hardware. Right: optimized code produces approximately the same results but allows the compiler to instrument it with more efficient or hardware-supported transcendentals.

Optimizations shown in Listing 4.13 may work for some architectures and compilers, but not for others. For instance, while `exp2()` is expected to be faster than `exp()` on Intel architectures with the Intel Math Library, however, the effect may be opposite with other platforms and compilers. The programmer may improve code portability using the preprocessor macros as shown in Listing 4.14.

```c
#ifdef __INTEL_COMPILER
#define FASTEXP exp2
#else
#define FASTEXP exp
#endif
y=FASTEXP(x);
```

Listing 4.14: Using preprocessor macros for performance portability.

It is important to stress again that expressions optimized with strength reduction generally do not yield bitwise-identical results.

Common Subexpression Elimination

Sometimes the compiler can automatically detect when an expression is re-used multiple times or within a loop, and precompute it, as was shown in Listing 4.18. This procedure is known as *common subexpression elimination*. To insure against situations when the compiler is unable to implement this optimization, it can be expressed in the code as shown in Figure 4.15.

```
for (int i = 0; i < n; i++) {
 for (int j = 0; j < m; j++) {
  const double r=
     sin(A[i])*cos(B[j]);
  // ...

 }
}
```

```
for (int i = 0; i < n; i++) {
 const double sin_A=sin(A[i]);
 for (int j = 0; j < m; j++) {
  const double cos_B=cos(B[j]);
  const double r=sin_A*cos_B;
     // ...
 }
}
```

Listing 4.15: Left: non-optimized code computes the value of `sin(A[i])` for every iteration in `j`. Right: optimized code eliminates redundant calculations of `sin(A[i])` by precomputing it. Note that the assignment of the variable `cos_B` will be eliminated by the compiler at `-O2` and higher in a procedure known as *constant propagation*.

4.2.4. Programming Practices for High Performance

This section presents a list of practices which we found to be helpful for applications designed with the Intel C and C++ compiler. At the same time, from working with third-party codes, and from feedback on our developer training, we understand that these practices are not always followed by experts in the field. That is because, at different times and with different platforms and compilers, some of these practices either have no effect, or have the opposite effect to what we observe with the Intel tools and platforms. Therefore, we hope that the information in this section will be useful for developers working today on applications for Intel parallel architectures developed with the Intel C and C++ compilers.

Ternary Operator Trap

The ternary operator `?:` is a short-hand expression for the `if...else` statement. Sometimes the syntax of this operator can cause redundant calculations, like shown in the example below.

```
#define min(a, b) ( (a) < (b) ? (a) : (b) )
const float c = min(my_function(x), my_function(y));
```

Listing 4.16: In this sub-optimal code, *three* calls to `my_function()` will be made: two calls to evaluate the condition `(a<b)` and one more call to substitute the result.

To avoid this trap, the programmer may precompute the expressions used in the ternary operator.

```
#define min(a, b) ( (a) < (b) ? (a) : (b) )
const float result_a = my_function(x);
const float result_b = my_function(y);
const float c = min(a, b);
```

Listing 4.17: In this optimized code, only *two* calls to `my_function()` will be made.

Using the `const` Qualifier

Whenever a local variable or a function argument is not supposed to change value in the code, it is beneficial to declare it with the `const` qualifier. This enables more aggressive compiler optimizations, including precomputing commonly used combinations of constants. For example, the code in Listing 4.18 executes almost 4x faster when `T` is declared with the `const` qualifier. This code is representative of numerical integration with the rectangle rule.

Sub-optimal

```
#include <cstdio>
int main() {
  const int N=1<<28;
  double T = (double)N;
  double s = 0.0;
  for (int i = 0; i < N; i++)
    s += (double)i/T;
  printf("Result=%e\n", s);
}
```

```
vega@lyra% icpc noconst.cc
vega@lyra% tmr ./a.out
Result=1.342117e+08
Time: 0.400 s
```

Optimized

```
#include <cstdio>
int main() {
  const int N=1<<28;
  const double T = (double)N;
  double s = 0.0;
  for (int i = 0; i < N; i++)
    s += (double)i/T;
  printf("Result=%e\n", s);
}
```

```
vega@lyra% icpc const.cc
vega@lyra% tmr ./a.out
Result=1.342117e+08
Time: 0.110 s
```

Listing 4.18: The sub-optimal code on the left takes 4x longer to compute than the optimized code on the right. The `const` qualifier on the variable `T` permits the compiler to precompute the reciprocal `1/T` and use multiplication instead of division in every iteration.

Analysis of assembly shows that in the optimized version with the `const` qualifier, the compiler replaced division by `T` with multiplication by the precomputed reciprocal `1/T`. With multiplication being a cheaper operation than division, performance was improved. Without the `const` qualifier, the compiler was not able to make this strength reduction optimization, even though floating-point semantics permits it at the default optimization level.

Array Reference by Index instead of Pointer Arithmetic

Intel Compilers are able to better optimize the code when memory access pattern in loops is clearly expressed. Specifically, addressing arrays by index is used instead of pointers is generally beneficial (i.e., `a[i]` instead of `*(a+i)`). Listing 4.19 Illustrates this recommendation.

Sub-optimal

```
#include <cstdio>
int main(){
 const long N = 800;
 float a[N*N], b[N*N], c[N*N];
 a[:] = b[:] = 1.0f;
 c[:] = 0.0f;
 for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++){
   float* cp = c + i*N + j;
    for(int k = 0; k < N; k++)
     *cp +=    a[i*N + k]
            * b[k*N + j];
   }
 printf("Result=%f\n", c[0]);
}
```

```
user@host% icpc array_ptr.cc
user@host% tmr ./a.out
Result=800.000000
Time: 0.733 s
```

Optimized

```
#include <cstdio>
int main(){
 const long N = 800;
 float a[N*N], b[N*N], c[N*N];
 a[:] = b[:] = 1.0f;
 c[:] = 0.0f;
 for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
   for (int k = 0; k < N; k++)
    c[i*N + j] +=   a[i*N + k]
                  * b[k*N + j];

 printf("Result=%f\n", c[0]);

}
```

```
user@host% icpc array_index.cc
user@host% tmr ./a.out
Result=800.000000
Time: 0.124 s
```

Listing 4.19: The intentionally crippled code on the left takes more than 5x longer to compute than the optimized code on the right. The only difference between the codes is the reference to the array element `c[i*N + j]`.

Optimization report explains why the example with pointer reference is slower. The compiler decided not to vectorize the inner loop and implemented it with scalar multiplications and additions. In contrast, with array notation, the compiler was able to interchange the loops in `j` and k, which opened up a unit-stride vectorization path, so the code was vectorized.

Overhead of Abstraction

Complex C++ classes may introduce significant overhead for operations that are expected to be fast for simple objects like C arrays. For example, data containers may perform some operations to update their internal state with every read or write operation. It may be possible to reduce the computational expense of manipulations with complex classes by outsourcing a part of the calculation to simpler objects.

For example, the calculation shown in Figure 4.20 was found in a scientific code employing the CERN ROOT library [28]. The code performs binning (i.e., hashing) of events marked by energy values into a histogram, in which the values of the bins represent the number of events with energies between the bin boundaries. The histogram is represented by the ROOT library's `class TH1F`, which offers additional histogram functionality required in the project. However, the binning process is a bottleneck of the project, because the method `Fill`, called over 10^9 times, involves unnecessary overhead.

```
// class TH1F contains a histogram with the number of bins equal to
// nBins, which span the values from energyMin to energyMax. nBins
// is of order 100
TH1F* energyHistogram = new TH1F("energyHistogram", "", nBins,
                                  energyMin, energyMax)

// Method TH1F::Fill adds an event with the value eventEnergy[i]
// to the histogram.

// nEvents is of order 1e9.
for (unsigned long i = 0; i < nEvents; i++)
    energyHistogram->Fill( eventEnergy[i] );
```

Listing 4.20: This code employs the ROOT library to construct a histogram implemented in `class TH1F`. The generation of the histogram is a bottleneck of the project.

The code was optimized by precomputing the values of the bins in the histogram using a more lightweight object, an array of `long` integers. The array was then loaded into the histogram using an overloaded `Fill` method, which takes the number of events in the bin, as opposed to adding a single event at a time. As a result, the execution time of the binning process was significantly reduced.

```
// array tempHistogram is used to prepare the histogram for loading
// into class TH1F
long tempHistogram[nBins]; tempHistogram[:] = 0;
const floatinvBinWidth = (energyMin - energyMax) / (float)nBins;
for (long i = 0; i < nEvents; i++) {
  bin = (int)floorf(eventEnergy[i] * invBinWidth);
  if ((0 <= bin) && (bin < nBins))
    tempHistogram[bin]++;
}
// Now the histogram class is filled, but this time
// only nBins=100 calls to TH1F::Fill are made, instead of
// nEvents=1e9 calls
TH1F* energyHistogram = new TH1F("energyHistogram", "", nBins,
                                  energyMin, energyMax);
for (int i = 0; i < nBins; i++)
  energyHistogram->Fill(((float)i+0.5f)*binWidth,
                          (float)tempHistogram[i]);
```

Listing 4.21: This code produces approximately the same results as the code in Listing 4.20, but works much faster, because the expensive method `Fill` is called fewer times.

The calculation of the histogram can be further accelerated through vectorization and thread parallelism. An example of these optimizations is discussed in Section 4.4.1.

While the example above is specific to the ROOT library, the principle illustrated here applies to other situations as well. When the overhead of operations in a library function or class is beyond the control of the developer, it may be possible to eliminate that overhead by precomputing some of the data in lightweight objects like C arrays.

4.2.5. Math Kernel Library for Scalar Arithmetic

Continuing the discussion of scalar optimization, it is natural to mention the possibilities offered by Intel Math Kernel Library (Intel MKL). Intel MKL is a comprehensive mathematical library for applications using Intel Xeon processor and Intel Xeon Phi coprocessors. Full scope of Intel MKL functionality is discussed in Section 5.1.

Here we summarize the section on scalar and general optimization by pointing out that many of the tasks requiring these kind of optimizations are implemented in Intel MKL. These implementations are tuned for Intel processors and coprocessors. Table 4.5 summarizes the Intel MKL functionality that programmers interested in this Section may find useful.

Functionality	Tools available in Intel MKL
Mathematical functions	Useful for applying to vectors larger than 40 elements. Support for `float`, `double`, `MKL_Complex8` and `MKL_Complex16` data types. Include: basic arithmetic, linear transformations, square and cube roots, power, hypotenuse, exponential, logarithmic, trigonometric, hyperbolic, error, gamma, rounding functions. Able to scatter/-gather non-unit stride vectors.
Statistical functions	Random number generators, convolution and correlation, summary statistics.
Data fitting functions	Linear, piece-wise polynomial interpolations of scalar-valued and vector-valued functions

Table 4.5: Intel MKL functionality related to scalar and general optimization.

Other facilities of MKL do not belong to this section as they are not just scalar functions: they leverage multi-threading and distributed computing. See Section 5.1 for the complete list that includes: BLAS, sparse BLAS, PBLAS, LAPACK, ScaLAPACK, sparse solvers, eigensolver routines, Fourier transforms, partial differential equations support, and nonlinear optimization problem solvers.

Examples in Listing 4.22 demonstrate one of the scalar functions at work: Intel MKL random number generator.

```
#include <cstdlib>
#include <cstdio>

int main(){
 const size_t N =1<<29L;
 const size_t F =sizeof(float);
 float* A =(float*)malloc(N*F);
 // Initialize RNG
 srand(0);
 for (int i = 0; i < N; i++){
  A[i]=  (float)rand()
        /(float)RAND_MAX;
 }

 printf("Generated %ld random \
#s.\nA[0]=%e\n", N, A[0]);
 free(A);
}
```

```
#include <cstdlib>
#include <cstdio>
#include <mkl_vsl.h>
int main() {
 const size_t N= 1<<29L;
 const size_t F= sizeof(float);
 float* A= (float*)malloc(N*F);
 VSLStreamStatePtr rnStream;
 // Initialize RNG
 vslNewStream( &rnStream,
   VSL_BRNG_MT19937, 1 );
 vsRngUniform( // Run RNG
   VSL_RNG_METHOD_UNIFORM_STD,
   rnStream, N, A, 0.0f, 1.0f);
 printf("Generated %ld random \
#s.\nA[0]=%e\n", N, A[0]);
 free(A);
}
```

```
vega@lyra% icpc -mmic rand.cc
vega@lyra%
vega@lyra% ssh mic0 \
>            tmr ${PWD}/a.out
Generated 536870912 random #s.
A[0]=8.401877e-01
Time: 49.108 s
```

```
vega@lyra% icpc -mkl \
>           -mmic  rand-mkl.cc
vega@lyra% ssh mic0 \
>                tmr ${PWD}/a.out
Generated 536870912 random #s.
A[0]=1.343642e-01
Time: 4.615 s
```

Listing 4.22: Comparison of random number generation on an Intel Xeon Phi coprocessor with the C Standard Library (left-hand side) and Intel MKL (right-hand side).

Both codes in Listing 4.22 perform the same task: the generation of a set of random numbers. However, the code in the left-hand side of the listing is based on the C Standard General Utilities Library, and in the right-hand side, the code is using the Intel MKL. The performance on the Intel Xeon Phi coprocessor is better with MKL by a factor of more than 10x. In addition, the MKL implementation is thread-safe and can be efficiently used by multiple threads in an application. That is not true of the random number generator implemented in the C Standard Library.

4.3. Optimizing Vectorization

This section presents optimization advice for automatic vectorization. The guidelines for developing applications suitable for auto-vectorization are outlined: unit-stride access, elimination of real and assumed vector dependence and data alignment, as well as some supplementary compiler functions, such as vectorization pragmas and vectorization report, and programming techniques for vectorization facilitation: strip-mining and regularizing the vectorization pattern.

4.3.1. Diagnosing the Utilization of Vector Instructions

When one begins to port and optimize an application for execution on Intel Xeon processors and/or Intel Xeon Phi coprocessors, it is important to determine whether the existing application takes advantage of vector instructions. There are multiple ways to do that:

1. When the performance-critical parts of the application are known, the programmer may use the compiler arguments `-qopt-report=n` and `-qopt-report-phase:vec` where `n` is an integer from 0 to 5 controlling the level of report verbosity. With this argument, the compiler generates the information about loops that are automatically vectorized or not vectorized, along with brief explanations.

2. It is also possible to use the Intel VTune Amplifier XE in order to diagnose the number of scalar and vector instructions issued by the code.

3. Finally, there is a simple and practical way to test the effect of automatic vectorization on the application performance. First, compile and benchmark the code with all the default compiler options. Then, compile the code with arguments `-no-vec -no-simd -no-openmp-simd` and benchmark again. With these options, automatic vectorization is disabled. If the difference in the performance is significant, it indicates that automatic vectorization already contributes to the performance. Note that this method works best when the application is benchmarked with only one thread. This reduces the impact of other factors (such as memory traffic and multi-threading overhead) on the execution time of the code.

4.3.2. Unit-Stride Access and Spatial Locality of Reference

Even though Intel compilers are able to vectorize loops with complex data access patterns, the best performance is achieved when data is accessed with unit stride. That is, in each iteration of a vectorized loop, scalar data elements packed into the vector register must be contiguous in memory. The rule of thumb for achieving unit-stride access is to use structures of arrays (SoA) instead of arrays of structures (AoS).

For an illustration, consider the following physical problem (this examples is based on Colfax Research publication [29] and available as Lab 4.02 – see Section 6.2). Suppose we have m particles identified by index i. Each particle is described by three Cartesian coordinates: $\vec{r} \equiv (r_{i,x},\ r_{i,y},\ r_{i,z})$ and charge q_i. We need to calculate the electric potential $\Phi(\vec{R})$ at multiple observation locations $\vec{R}_j \equiv (R_{j,x}, R_{j,y}, R_{j,z})$, where index j denotes one of n locations. The expression for that potential is given by Coulomb's law:

$$\Phi\left(\vec{R}_j\right) = -\sum_{i=1}^{m} \frac{q_i}{\sqrt{\left(r_{i,x} - R_{j,x}\right)^2 + \left(r_{i,y} - R_{j,y}\right)^2 + \left(r_{i,z} - R_{j,z}\right)^2}}. \quad (4.1)$$

Figure 4.1 is a visual illustration of the problem. In the left panel, $m = 512$ charges are distributed in a lattice-like pattern. Each of these particles contributes to the electric potential at every point in space. The right panel of the figure shows the electric potential at $m = 128 \times 128$ points in the xy-plane at $z = 0$ calculated using Coulomb's law (Equation 4.1).

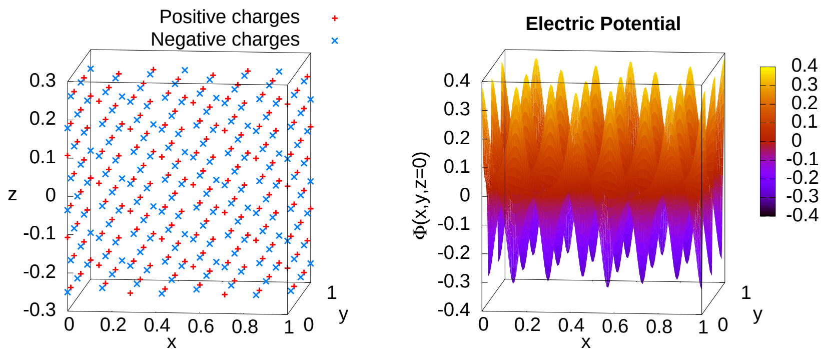


Figure 4.1: Left: charged particles. Right: electric potential $\Phi(z = 0)$ produced by the particles.

Elegant, but Inefficient Solution

To a physicist, it is natural to treat particles as the basis of physical models, and therefore it may seem reasonable to design an object-oriented code by defining the particle structure type as in Listing 4.23, and then declaring an array of structures (AoS) of this type.

```
struct ChargeType { // Elegant, but ineffective data layout
    float x, y, z, q; // Coordinates and value of this charge
};
ChargeType* particle; // Now declare an array of m point charges
particle = (ChargeType*)_mm_malloc(m*sizeof(ChargeType), 64);
```

Listing 4.23: Data organization as an array of structures (AoS), inefficient for vectorization.

Listing 4.24 demonstrates a function that calculates the electric potential at a point $\vec{R}$ defined by quantities `Rx`, `Ry` and `Rz` in the code.

```
// This version performs poorly because of strided access to data
void CalculateElectricPotential(
  const int m,       // Number of charges
  const ChargeType* particle, // Charge distribution (AoS)
  const float Rx, const float Ry, const float Rz, // Observation pt
  float & phi  // Output: electric potential
) {
  phi=0.0f;
  for (int i = 0; i < m; i++)  { // This loop is auto-vectorized
    // Non-unit stride between particle[i].x and particle[i+1].x
    const float dx = particle[i].x - Rx;
    const float dy = particle[i].y - Ry;
    const float dz = particle[i].z - Rz;
    phi -= particle[i].q/sqrtf(dx*dx+dy*dy+dz*dz); // Coulomb's law
  }
}
```

Listing 4.24: Inefficient solution for Coulomb's law application: access to quantities `x`, `y`, `z` and `q` has a stride of 4 rather than 1, because data is stored as AoS.

We used this code in a multi-threaded calculation where different threads compute the potential for different values of $\vec{R}$. Benchmarks show that, assuming 10 floating-point operations per charge per location, the performance of this code is 128 GFLOP/s on an Intel Xeon-based host, and 190 GFLOP/s on an Intel Xeon Phi coprocessor.

The performance of this application can be improved by code optimization. Indeed, consider the inner for-loop in line 9 of Listing 4.24. The variable `particle[i].x` in the i-th iteration is `4*sizeof(float)=16` bytes away in memory from `particle[i+1].x` used in the next iteration. This corresponds to a stride of `16/sizeof(float)=4` instead of 1, which will incur a performance hit when the data is loaded into the processor's vector registers. The same goes for members `y`, `z` and `q` of class `Charge`. Even though Intel Xeon Phi coprocessors support gather/scatter vector instructions, unit-stride access to vector data is almost always more efficient.

The left-hand side panel of Figure 4.2 illustrates the data layout in memory and the non-unit stride gather operation required to load a set of coordinates `x` into a vector register. In the next section we will optimize the code by changing the data layout to a structure of arrays in order to reproduce the unit-stride access illustrated in the right-hand side panel of Figure 4.2.

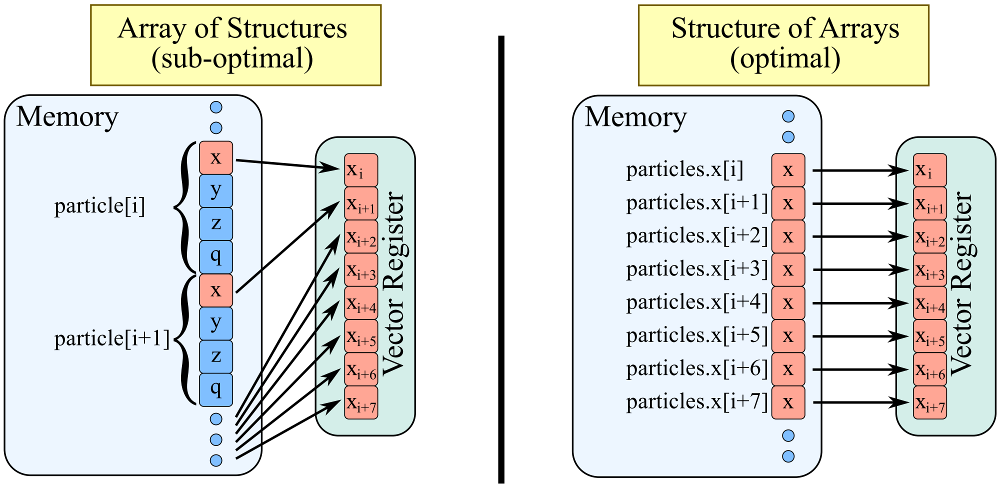


Figure 4.2: Difference in memory access between AoS and SoA

Optimized Solution with Unit-Stride Access

To achieve unit-stride data access in the for-loop of the performance-critical function `CalculateElectricPotential`, the data structure needs to be re-organized. Instead of the inefficient `struct ChargeType`, we can declare `struct ChargeDistributionType` containing the properties of charges as arrays (the SoA approach), as shown in Listing 4.25.

```
struct ChargeDistributionType { // Efficient layout: SoA
  const int m; // Number of charges
  float *x, *y, *z, *q; // Arrays of x, y, z and q of particles
};
ChargeDistributionType particles;
```

Listing 4.25: Data storage as a structure of arrays is usually beneficial for vectorization.

With this new data structure, the function calculating the electric potential takes the form shown in Listing 4.26.

```
1  // This version vectorizes better thanks to unit-stride data access
2  void CalculateElectricPotential(
3    const int m,     // Number of charges
4    const ChargeDistributionType &particles, // Charge distrib. (SoA)
5    const float Rx, const float Ry, const float Rz, // Observation pt
6    float & phi  // Output: electric potential
7  ) {
8    phi=0.0f;
9    for (int i = 0; i < particles.m; i++)  {
10     // Unit stride between particles.x[i] and particles.x[i+1]
11     const float dx = particles.x[i] - Rx;
12     const float dy = particles.y[i] - Ry;
13     const float dz = particles.z[i] - Rz;
14     phi -= particles.q[i]/sqrtf(dx*dx+dy*dy+dz*dz); //Coulomb's law
15   }
16 }
```

Listing 4.26: Efficient vectorization: unit-stride access to quantities `x`, `y`, `z` and `q`.

Inner for-loop in line 9 of Listing 4.26 has unit-stride data access, as `particles.x[i]` is immediately followed by `particles.x[i+1]` in memory. The same is true for all other quantities accessed via the array iterator `i`. The performance of this code is now 331 GFLOP/s on the host (2.6x faster than non-optimized) and 527 GFLOP/s (2.8x faster than non-optimized) on the coprocessor. The latter can be further increased to 764 GFLOP/s by compiling the code with `-fp-model fast=2` (see Section 4.2.2). Figure 4.3 summarizes the results.

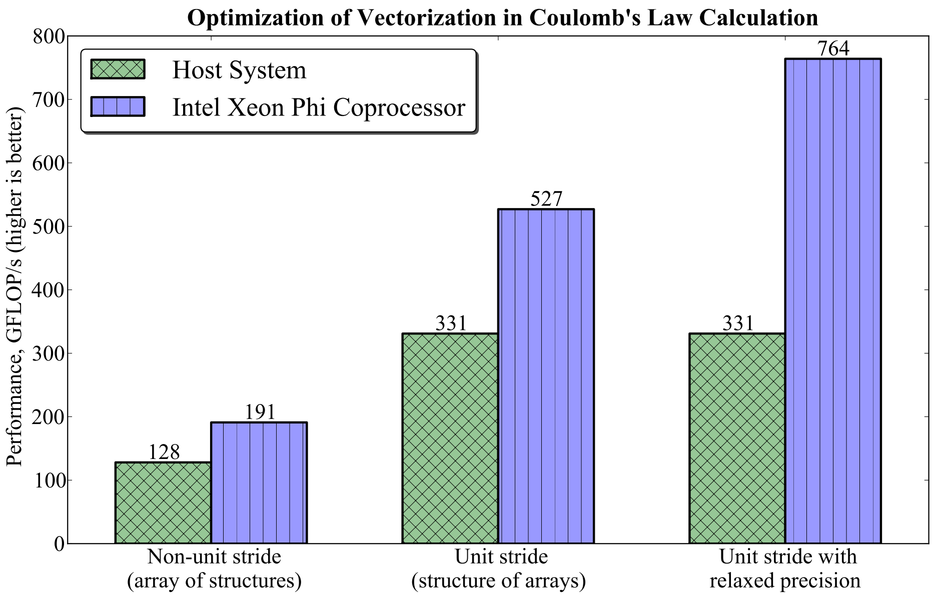


Figure 4.3: Performance of the electric potential calculation with Coulomb's law discussed in Section 4.3.2, before and after optimization.

The above example demonstrates how converting data layout from an array of structures to a structure of arrays results in unit-stride access to data, which leads to significant performance improvements. The necessity of unit-stride access is an inherent property of all computer architectures with cache-line granularity of access to memory. It is a manifestation of a more general prerequisite of high performance, *locality of reference* of data in *space*. See also Section 4.5 for an illustration of how improving locality of reference in *time* can improve performance by optimizing the cache traffic.

4.3.3. Regularizing Vectorization Pattern

VPUs of Intel Xeon Phi coprocessors can process vector instructions on vectors of exactly 16 or 8 elements in single and double precision, respectively. At the same time, the Intel C++ Compiler is able to vectorize loops for which loop count is not known at compilation time. Furthermore, load and store instructions in the KNC architecture can operate only on 64-byte aligned memory regions. However, this does not prevent the Intel C++ Compiler from vectorizing loops in which the alignment of the first element is not known at compilation time.

In cases where the loop count or alignment situation is known only at runtime, the compiler will implement a check at the beginning of the loop. Depending on this check, the code may peel off a few iterations at the beginning of the loop, or process the remainder (i.e., tail) of the loop with scalar or masked vector instructions. Naturally, a loop with beginning or remainder peeled off is less efficient than a loop that consists only of vector iterations. Peeling is illustrated in Figure 4.4 (cases "Code Path 1" and "Code Path 2" may be taken at runtime depending on the length of the loop and on the data alignment situation).

```
for (i = 0; i < n; i++)  A[i] = ...
```

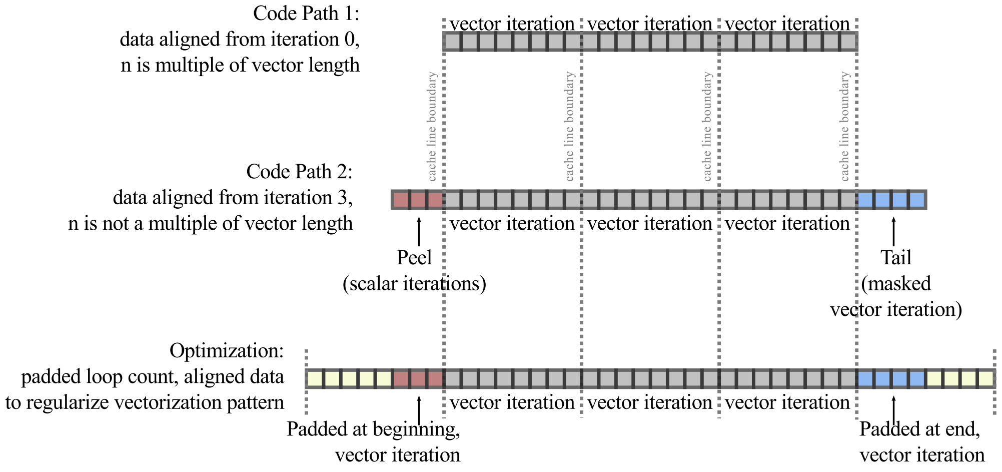


Figure 4.4: Compiler may peel an irregular loop. To prevent that, the programmer may regularize the vectorization pattern.

The programmer can alleviate the load on the platform by regularizing the pattern of vectorization, so that the operations are always vector and always on aligned data. This may require three measures:

1. Padding the loop count to a multiple of the vector length
2. Padding data to a multiple of vector length
3. Aligning data on the appropriate boundary

We refer to these optimizations that lead to the elimination of loop peel and remainder as "regularizing the vectorization pattern". The resulting loop may formally have more iterations, where in marginal iterations the processor crunches garbage data (see Figure 4.4, case "Optimization"). However, the performance of regularized loops on the Intel MIC architecture is generally better than that of irregular loops, because the peeled iterations take more clock cycles than a single vector iteration.

The compiler also supports semi-automated application of some of these application. For instance, vectorization of loop remainder and assumption of safe padding is described in the article "Utilizing Full Vectors and Use of Option -qopt-assume-safe-padding", which is a part of [30].

Example: LU Decomposition

To demonstrate this procedure, consider the Doolittle algorithm of LU decomposition (example presented here is based on a Colfax Research publication [31] and also available among Supplementary Code for Practical Exercises as Lab 4.03 – see Section 6.2). The purpose of this algorithm is to represent a square, non-degenerate matrix A as a product $A = LU$, where L is a unit lower triangular matrix, and U is an upper triangular matrix. Such a decomposition is commonly used to solve systems of linear algebraic equations.

The Doolittle algorithm applied to an $n \times n$ matrix performs $n-1$ iterations generally resembling the Gaussian elimination scheme. For iteration b, matrix row b is multiplied by a certain factor and added to matrix rows $b+1$ through $n-1$ so that the in the resulting matrix $A^{(b)}$, all elements in column b starting from $b+1$ are equal to zero. The coefficients of that multiplication are recorded in a separate matrix L:

$$A^{(0)} = A, \tag{4.2}$$

$$A^{(b+1)} = L^{(b)} A^{(b)}, \text{ where} \tag{4.3}$$

$$L^{(b)}_{ij} = \begin{cases} 1, \text{ if } i = j, \\ l_{i,b}, \text{ if } i > j \text{ and } j = b, \\ 0, \text{ otherwise}, \end{cases} \tag{4.4}$$

$$l_{i,b} = -\frac{A^{(b)}_{i,b}}{A^{(b)}_{b,b}}. \tag{4.5}$$

As a result, $U \equiv A^{(n-1)}$, and elements of L are

$$L_{i,j} = \begin{cases} 1, \text{ if } i = j, \\ -l_{i,j}, \text{ if } i < j, \\ 0, \text{ otherwise}. \end{cases} \tag{4.6}$$

Baseline

A simplified implementation of this procedure is shown in Listing 4.27. It is simplified because the algorithm does not involve pivoting (i.e., choosing the best row to eliminate elements in other rows) and is not optimized. The algorithm is single-threaded, and we are going to keep it this way: we assume that this function is called from a parallel region to process multiple independent matrices concurrently.

```
 1 void LU_decomp(const int n, float* const A) {
 2   // LU decomposition (Doolittle algorithm)
 3   // In-place decomposition of form A=LU
 4   // L is returned below main diagonal of A
 5   // U is returned at and above main diagonal
 6   for (int b = 0; b < n; b++) {
 7     // Strength reduction:
 8     const float recAbb = 1.0f/A[b*n + b];
 9     for (int i = b+1; i < n; i++) {
10       A[i*n + b] = A[i*n + b]*recAbb;
11       for (int j = b+1; j < n; j++)
12         A[i*n + j] -= A[i*n + b]*A[b*n + j];
13     }
14   }
15 }
```

Listing 4.27: LU decomposition, non-optimized, with irregular inner loop.

Note that we have already applied a scalar optimization technique, strength reduction, by precomputing the reciprocal of `A[b*n+b]` in line 8. This reciprocal is later used in the loop for multiplication in line 10.

However, the vectorization aspect of this code still remains to be optimized. Areas from optimization can be inferred from the optimization report produced by the compiler. Listing 4.28 shows the relevant parts of the report produced with the argument `-qopt-report=4`.

Indeed, the report indicates that the compiler implemented different versions of the loop depending on the alignment and aliasing situation at runtime. Also, loop peel and remainder processing were prepared for use at runtime.

We established the baseline performance of this code by running the

LU decomposition in multiple threads on a set of 10^4 matrices 128×128 elements in size. Each matrix was processed by only one of the active threads. To convert execution time into tangible performance figures, we assumed that each factorization requires $(2/3)n^3$ FLOPs (this is the asymptotic sum of the series $2n(n-1)+2(n-2)(n-1)+...+2\times2\times1$ for $n \to \infty$). Our Intel Xeon Phi coprocessor then yielded 97 GFLOP/s, and the host system 144 GFLOP/s. Apparently, the inefficiencies in this code hamper the Intel MIC architecture much more than the multi-core Intel Xeon CPU.

```
LOOP BEGIN at main.cc(11,7)
<Peeled, Multiversioned v1>
... PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at main.cc(11,7)
<Multiversioned v1>
... reference A has aligned access   [ main.cc(12,2) ]
... LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at main.cc(11,7)
<Remainder, Multiversioned v1>
... unaligned access used inside loop body
...   REMAINDER LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at main.cc(11,7)
<Multiversioned v2>
... non-vectorizable loop instance from multiversioning
LOOP END
```

Listing 4.28: Optimization report for Listing 4.27

Regularizing vectorization

Now we will optimize the code by regularizing the vectorization pattern in the inner loop. The inner loop in `j` has different loop counts in every iteration in `b`. Additionally, it sometimes begins on an aligned data element, other times on unaligned. To regularize this loop, instead of starting the inner loop from `j=b`, let's start it from `jmin`, which is the greatest multiple of 16 which does not exceed `b`. Our procedure also assumes that the matrix size, `n`, is also a multiple of the vector length.

The resulting code is shown in Listing 4.29. We also established empirically that on the CPU, a multiple of 8 works well, while on the coprocessor, multiples of 32 are better. Therefore, to optimize the code, we performed target-specific tuning using the preprocessor macro `__MIC__` (see Section 2.2.6). Furthermore, it always starts on an aligned data element as long as matrix `A` itself is aligned.

Naturally, the optimized version computes the addition and multiplication of a greater number of values, however, it does so in fewer processor instructions and fewer clock cycles.

While regularization of vectorization by padding the iteration count is good for performance, the programmer must control whether the additional arithmetic operations preserve correctness. In our code, we had to introduce a temporary matrix `L` in order to prevent the padded loop from overwriting the elements below the main diagonal of `A`. Matrix `L` preserves these elements. At the end of the calculation, `L` is copied into `A` and removed from memory. This optimization improved the performance on the coprocessor by 30% increasing it to 126 GFLOP/s. It also increased the CPU performance by 20% to 173 GFLOP/s.

Because there is still a significant gap between the host and coprocessor performance, we will continue with the optimization of this code in subsequent sections. One of the techniques related to regularization of vectorization pattern is informing the compiler about alignment guarantees. This is discussed in the next session.

```
void LU_decomp(const int n,
                  float* const A) {
  // LU decomposition (Doolittle algorithm)
  // In-place decomposition of form A=LU
  // L is returned below main diagonal of A
  // U is returned at and above main diagonal
#ifdef __MIC__
  const int tile=32;
#else
  const int tile=8;
#endif
  assert(n%tile==0);
  // Must store L separately from A
  float L[n*n] __attribute__((aligned(64)));
  for (int i = 0; i < n; i++) {
    L[i*n:n]=0.0f;
    L[i*n+i]=1.0f;
  }
  for (int b = 0; b < n; b++) {
    const int jMin = b - b%tile;
    // Strength reduction:
    const float recAbb = 1.0f/A[b*n + b];
    for (int i = b+1; i < n; i++) {
      L[i*n + b] = A[i*n + b]*recAbb;
      // Regularized pattern of vector loop:
      for (int j = jMin; j < n; j++)
        A[i*n + j] -= L[i*n+b]*A[b*n + j];
    }
  }
  // Copy temp matrix L into matrix A
  for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
      A[i*n + j] = L[i*n + j];
}
```

Listing 4.29: Gaussian elimination with regularized inner loop.

4.3.4. Compiler Hints: Aligned Data Notice

Data alignment on a 64-byte boundary is required for vector instructions in the Many Integrated Core architecture of Intel Xeon Phi coprocessors. The alignment of the first element in a pointer-based array is generally not known at compile time. Therefore, in automatically vectorized loops, the compiler must implement a check for alignment. Depending on the results of the check, the application may peel off several iterations at the beginning of the loop in order to reach the first aligned element. The alignment check may take a significant time, especially for short loops, and multiple versions of the code required for execution take up space in the L1 instruction cache.

If the programmer can guarantee that pointer-based arrays in a vectorized loop are aligned, it is beneficial to tell the compiler to assume alignment at the beginning of the loop. This may be done using `#pragma vector aligned` applied to the whole loop or specifier `__assume_aligned()` applied to individual arrays (see Section 3.1.10).

Applying this optimization to our LU decomposition code in Listing 4.29 results in the following (only the relevant snippet is shown below):

```
    for (int i = b+1; i < n; i++) {
      L[i*n + b] = A[i*n + b]*recAbb;
      // Aligned data hint:
#pragma vector aligned
      for (int j = jMin; j < n; j++)
        A[i*n + j] -= L[i*n+b]*A[b*n + j];
    }
```

Listing 4.30: Informing the compiler about alignment in the LU decomposition code.

Note that in order for the code in Listing 4.30 to work, array `A` must be aligned, and the length of its rows, `n`, should be a multiple of the alignment length. Otherwise, the application will crash with a segmentation fault. We guarantee alignment of `A` in the body of the application that uses the function `LU_Doolittle()`. This is done with methods discussed in Section 3.1.4. We also check that `n` is a multiple of 16, which amounts to 64 bytes in single precision. In practice, if a matrix inner dimension is not a multiple of 16, the programmer may pad the matrix row length.

With `#pragma vector aligned`, performance on the coprocessor increased by 13% to 142 GFLOP/s, and marginally decreased on the host.

4.3.5. Compiler Hints: Pointer Disambiguation

Another hint given to us by the optimization report of the LU decomposition code (Listing 4.28) is that the compiler instrumented multiversioning. This occurred because at compilation time, the compiler cannot be certain whether the value of `jMin` is between `0` and `n`. Therefore, it is not certain whether array references in the left-hand side and in the right-hand side of line 7 are aliased (i.e., pointing to the same memory regions) or not.

We can gain additional performance by eliminating multiversioning. This can be done as explained in Section 4.3.5, using `#pragma ivdep`. This pragma instructs the compiler to ignore assumed vector dependencies, and assume all pointers in the loop to be non-aliased. The same effect may be achieved using the keyword `restrict` (see Section 3.1.9).

The result of this optimization in shown in Listing 4.31. The new pragma can be combined with the pragma used in the previous step.

```
    for (int i = b+1; i < n; i++) {
      L[i*n + b] = A[i*n + b]*recAbb;
#pragma vector aligned
      // Pointer disambiguation:
#pragma ivdep
      for (int j = jMin; j < n; j++)
        A[i*n + j] -= L[i*n+b]*A[b*n + j];
    }
```

Listing 4.31: Informing the compiler about non-aliased pointers in the LU decomposition code.

Once pointers were disambiguated, the compiler dropped the assumption of vector dependence, and the performance on the CPU was bumped by around 6% to 174 GFLOP/s and on the coprocessor it increased by 10% to 156 GFLOP/s. Furthermore, the optimization report no longer includes information about multiversioning, which means that only one code path was implemented, with non-aliased `L` and `U`.

When `#pragma ivdep` is used, the penalty for supplying aliased pointers may be incorrect results, without a warning or error message. The programmer may manually implement a check for aliasing, but move it to the top of the function, rather than to the inner loop, and then use the pragma to allow the compiler to simplify the code.

Summary

The summary of the optimizations discussed in the last three sections is shown in Figure 4.5.

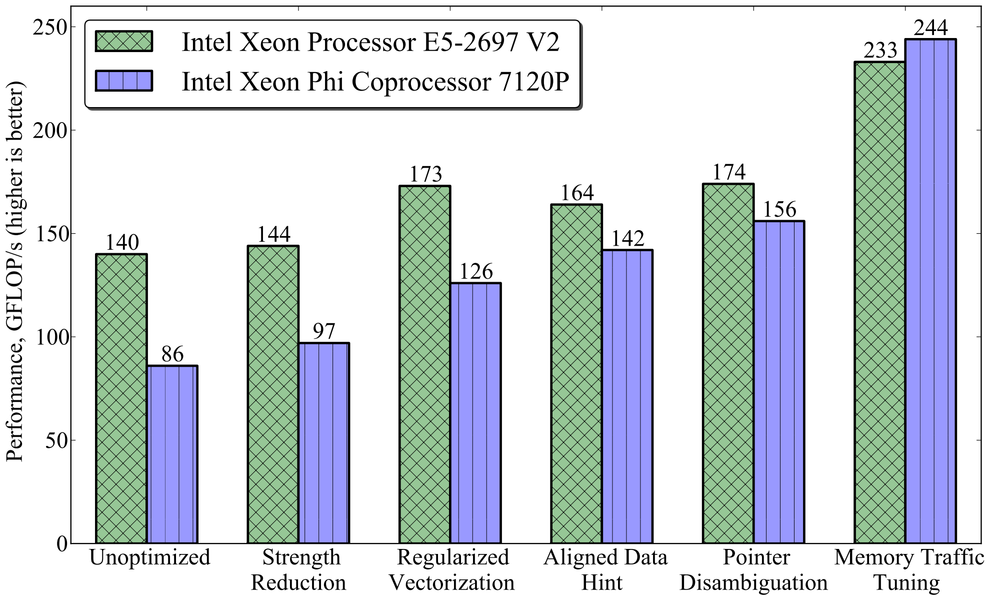
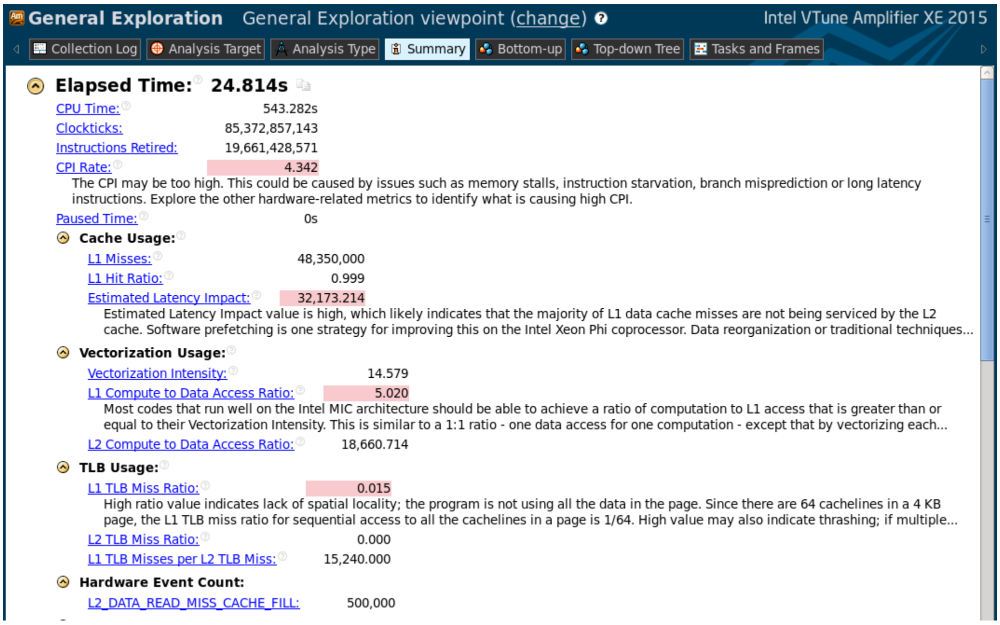


Figure 4.5: LU decomposition code performance optimization summary.

The first set of bars corresponds to an non-optimized code, which was not shown in the text, but which differs from Listing 4.27 only in that instead of precomputing the reciprocal of `A[b*n+b]`, there is a division by that element in the vector loop. After that, regularizing the vectorization pattern (Section 4.3.3) and using the compiler hints: aligned data notice and pointer disambiguation led to further improvements.

The resulting performance is greater on the host system than on the coprocessor. This is in part due to the fact that we are running a relatively small problem and comparing performance of a coprocessor with 0.5 MiB of L2 cache per core to the performance of a top of the line CPU with 2.5 MiB of cache per core.

At the same time, our performance result indicate that the optimization process is not complete. There is nothing left to be gained from vectorization tweaks. Minor performance gains may be tapped from tuning prefetching. However, the key to further optimization of this problem is memory traffic tuning. Indeed, matrices 128×128 in size occupy 64 KiB of memory. With 4 hardware threads per core, they fit in the L2 cache, but clearly spill over the L1 cache. This is confirmed by performance analysis in VTune (see Figure 4.6), which highlights the fields "estimated latency impact" and "L1 compute to data access ratio".



```
Elapsed Time: 24.814s
  CPU Time:                 543.282s
  Clockticks:               85,372,857,143
  Instructions Retired:     19,661,428,571
  CPI Rate:                 4.342
    The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency
    instructions. Explore the other hardware-related metrics to identify what is causing high CPI.
  Paused Time:              0s
  Cache Usage:
    L1 Misses:                  48,350,000
    L1 Hit Ratio:               0.999
    Estimated Latency Impact:   32,173.214
      Estimated Latency Impact value is high, which likely indicates that the majority of L1 data cache misses are not being serviced by the L2
      cache. Software prefetching is one strategy for improving this on the Intel Xeon Phi coprocessor. Data reorganization or traditional techniques...
  Vectorization Usage:
    Vectorization Intensity:            14.579
    L1 Compute to Data Access Ratio:    5.020
      Most codes that run well on the Intel MIC architecture should be able to achieve a ratio of computation to L1 access that is greater than or
      equal to their Vectorization Intensity. This is similar to a 1:1 ratio - one data access for one computation - except that by vectorizing each...
    L2 Compute to Data Access Ratio:    18,660.714
  TLB Usage:
    L1 TLB Miss Ratio:              0.015
      High ratio value indicates lack of spatial locality; the program is not using all the data in the page. Since there are 64 cachelines in a 4 KB
      page, the L1 TLB miss ratio for sequential access to all the cachelines in a page is 1/64. High value may also indicate thrashing; if multiple...
    L2 TLB Miss Ratio:              0.000
    L1 TLB Misses per L2 TLB Miss:  15,240.000
  Hardware Event Count:
    L2_DATA_READ_MISS_CACHE_FILL:   500,000
```

Figure 4.6: Summary of performance analysis in Intel VTune Amplifier.

Memory traffic tuning is a subject of a different conversation, which will resume in Section 4.5. However, as a motivational example, we implemented a version with optimized memory traffic, which runs 56% better than our last version, achieving 244 GFLOP/s on the coprocessor. On the host, this tuning increases performance to 233 GFLOP/s. We do not discuss this implementation in the text, however, the reader is welcome to study the corresponding code in the attachment to the book.

4.3.6. Strip-Mining for Vectorization

Principles: Strip-Mining

Strip-mining is a technique that transforms a single loop into two nested loops. The outer loop strides through "strips" or "tiles" of the iteration space, and the inner loop operates on the iterations inside the strip ("mining" it). Listing 4.32 demonstrates the strip-mining transformation.

```
// Original loop:
for (int i = 0; i < n; i++) {
  // ... do work
}

// Strip-mining converts the original loop into two nested loops:
const int TILE=80;
for (int ii = 0; ii < n; i += TILE)
  for (int i = ii; i < ii + TILE; i++) {
    // ... do work
  }
```

Listing 4.32: Strip-mining technique is usually implemented by the compiler "behind the scenes". However, it is easy to implement it manually, as shown in this listing.

This transformation can be used to allow vectorization to co-exist with multi-threading. An example of such application will be discussed in this section. The strip-mining technique can also be used to balance the parallelism in cores with parallelism in vectors as will be illustrated later in Section 4.4.4.

The size of the tile must usually be chosen as a multiple of the vector length in order to facilitate the vectorization of the inner loop. Furthermore, if the iteration count `n` is not a multiple of the tile size, then the programmer must peel off `n%TILE` iterations at the end of the loop.

Sometimes, strip-mining is used by the compiler "behind the scenes" in order to apply thread parallelism to vectorized loops. However, in some cases, explicit application of strip-mining may be necessary. An example of such a case is discussed below.

Example Problem: Computing a Histogram

We will illustrate the application of strip-mining with the problem of computing a histogram (code of this example is available as Lab 4.04 – see Section 6.2). Suppose array `age` contains floating-point numbers ranging from 0 to 100 representing the ages, in years, `n` of people. The task is to create array `hist` of size `m`, elements of which contain the number of people in age groups 0–20 years, 20–40 years, 40–60 years, etc. We will assume `m=5` (number of age groups covering the range 0-100), and `group_width=20.0f` (how many years each group spans) and `n=100000000` (large enough number so that `age` does not fit in the L2 cache).

This workload represents a set of problems where contiguous variables are interpolated onto a discrete grid. For example, in Monte Carlo particle transport simulations, coordinates and velocities of particles may need to be translated onto a spatial grid; in calculations involving piece-wise function interpolations, the code must find the interpolation bins corresponding to the values of function arguments.

A non-optimized serial C code that performs the histogram calculation is shown in Listing 4.33. This code is not protected from situations when one of the members of `age[]` is outside the range $[0.0 \ldots 100.0)$. We assume that the user of the function `Histogram()` is responsible for ensuring that the array `age` has only valid entries.

```
void Histogram(
  const float* age, // Ages, values from 0.0f to 100.0f
  const int n,      // Size of array age, n=100000000
  int* const hist,  // Output: counts in groups
  const int m,      // Size of array hist, m=5
  const float group_width // group_width=20.0f
  ) {
    for (int i = 0; i < n; i++) {
      const int idx = int(age[i]/group_width);
      hist[idx]++;
    }
}
```

Listing 4.33: Non-Optimized histogram calculation code.

Referring back to the earlier discussion on scalar optimization (Section 4.2.3), we can see opportunity for strength reduction in line 9. Indeed, the division by `group_width` can be replaced with multiplication by the reciprocal of this number. As we will see below (see Figure 4.7), this optimization improves performance on the CPU by 50% and on the coprocessor by 160%.

Now let's go back to the currently discussed optimization topic and asses this code from the point of view of vectorization. Code in Listing 4.33, even after strength reduction, is not optimal, because it cannot be auto-vectorized. The problem with vectorization is true vector dependence in the access to array `hist`. Indeed, consecutive iterations of the `i`-loop cause scattered writes to `hist`, which is not possible to express with vector instructions. However, the operation of computing the index `idx` does not have a vector dependence, and therefore this part of the calculation can be vectorized. Compiler optimization report, a fragment of which is shown in Listing 4.34, confirms that reasoning.

```
Begin optimization report for: Histogram(const float *,
                                            int, int *, int, float)
...
LOOP BEGIN at worker.cc(8,5)
   ...loop was not vectorized: vector dependence prevents
                                                   vectorization...
   ...vector dependence: assumed FLOW dependence between
                                               line 10 and  line 10
LOOP END
```

Listing 4.34: Vectorization report shows that histogram calculation is not vectorized.

To facilitate automatic vectorization, we can apply strip-mining. To "strip-mine" the `i`-loop, we express it as two nested loops: the outer loop with index `ii` and has a stride of `TILE = 16`, and an inner loop with index `i` that "mines" the strip of indexes between `ii` and `ii+TILE`. This modification will not help directly, however, it will allow the next step to happen. In the next step, we can split the inner loop in `i` into two consecutive loops: one for computing the index `idx` and the other incrementing the histogram values.

The first of these loops can be auto-vectorized, and the second will remain scalar.

Listing 4.35 demonstrates the code optimized with strip-mining and loop splitting. It produces the same results as the code in Listing 4.33, but faster, thanks to automatic vectorization. In addition to vectorization, we also implemented strength reduction by replacing the division by `group_width` with the multiplication by its reciprocal value.

```
void Histogram(const float* age, const int n, int* const hist,
               const int m, const float group_width) {

  const int TILE = 16; // Length of vectorized loop
  const float recWidth = 1.0f/group_width; // Precompute the
                                           // reciprocal

  // Strip-mining the loop in order to vectorize the inner short
  // loop. (Note: this algorithm works only if n%TILE == 0.)
  for (int ii = 0; ii < n; ii += TILE) {
    // Temporary storage for indices. Necessary for vectorization
    int idx[vecLen] __attribute__((aligned(64)));

    // Vectorize the multiplication and rounding
#pragma vector aligned
    for (int i = ii; i < ii + TILE; i++)
      idx[i-ii] = (int) ( age[i] * recWidth );

    // Stochastic memory access, does not get vectorized
    for (int c = 0; c < TILE; c++)
      hist[idx[c]]++;
  }
}
```

Listing 4.35: Vectorized histogram calculation code, with strength reduction and strip-mining.

We assume in this code that `n` is a multiple of `TILE`, i.e., `n%TILE==0`. This assumption is easy to lift by adding a peel loop in `i` from `n-n%TILE` to `n`. The choice of the value of `TILE=16` is dictated by the fact vector registers of Intel Xeon Phi coprocessors can fit 16 values of type `int`. Generally speaking, `TILE` must be a multiple of 16; however, experiments show that exactly 16 produces the best performance.

The performance of codes in Listing 4.33 and Listing 4.35 can be found in Figure 4.7. The scalar code performance is the baseline for this example. Vectorization made possible by strip-mining improves the performance on the CPU by 65% and on the coprocessor by 180%.

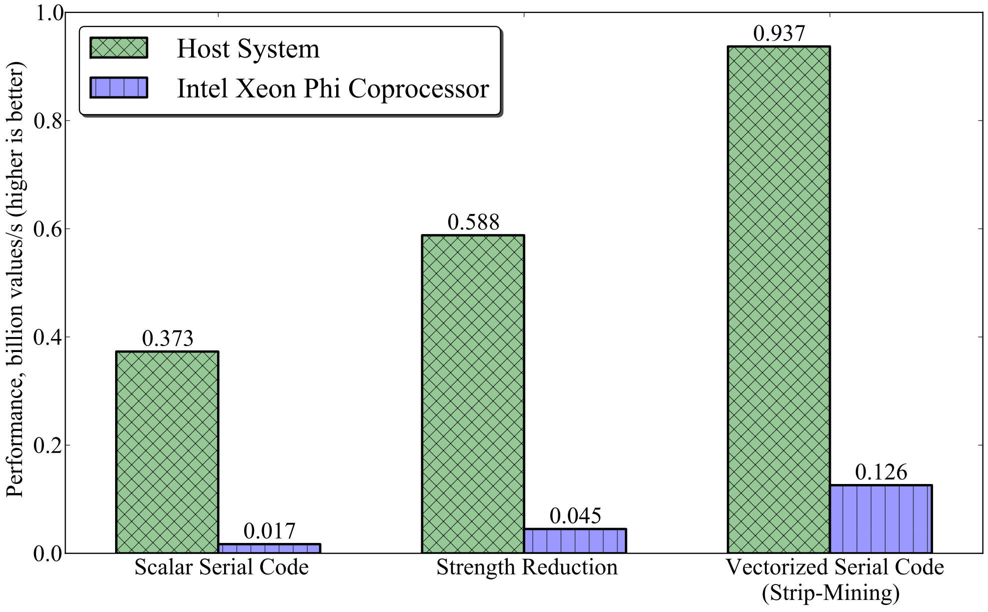


Figure 4.7: Performance of histogram calculation with strength reduction and strip-mining to assist automatic vectorization. Further optimization work is required.

The reason why the coprocessor is way behind the CPU in performance is that the code is single-threaded. Now that scalar optimization and vectorization have been implemented, we will proceed with the parallelization of this code in Section 4.4.

4.3.7. Additional "Tuning Knobs" for Vectorization

We have discussed compiler arguments and keywords (pragmas and qualifiers) that may be used to fine-tune automatically vectorized loops. Refer to the list in Section 3.1.10 for a more detailed summary of keywords and arguments for vectorization.

4.4. Optimization of Multi-Threading

Optimization advice for shared-memory parallel codes is presented in this section. This section addresses the most basic performance considerations for thread parallelism: reducing the synchronization overhead, exposing thread parallelism to the compiler, and load balancing.

4.4.1. Avoiding Synchronization through Parallel Reduction

In this section we continue the optimization of the histogram calculation introduced in Section 4.3.6. We will start where we left off with code shown in Listing 4.35 (code of this example is available as Lab 4.04 – see Section 6.2).

If we wanted to parallelize this code with OpenMP, a straightforward way to do it would be by putting the OpenMP parallel loop pragma before the outer loop as shown in Listing 4.36. The problem with this implementation is that it produces incorrect results due to a race condition in line 11. Race condition occurs when multiple threads concurrently increment the same element of `hist`. This will lead to unpredictable program behavior and incorrect results. See Section 3.2.5 for a refresher about race conditions.

```
 1 #pragma omp parallel for
 2   for (int ii = 0; ii < n; ii += TILE) {
 3     int idx[vecLen] __attribute__((aligned(64)));
 4
 5 #pragma vector aligned
 6     for (int i = ii; i < ii + TILE; i++)
 7       idx[i-ii] = (int) ( age[i] * recWidth );
 8
 9     // RACE CONDITION leads to incorrect results:
10     for (int c = 0; c < TILE; c++)
11       hist[idx[c]]++;
12   }
```

Listing 4.36: Incorrect parallel code for histogram calculation with a race condition.

To stabilize a program with a race condition, mutexes may be used, which generally incurs a performance penalty. This is often not the best way to resolve race conditions; let's see what happens when we try to use mutexes.

Bad Idea: Mutexes

Atomic operations in OpenMP (see Section 3.2.6) are lightweight mutexes suitable for protecting an increment operation as in our case. Listing 4.37 demonstrates an implementation of the histogram computation with atomic operations. With `#pragma omp atomic`, the result of the increment is guaranteed to be correct even if multiple threads access the same element of `hist` at the same time.

```
#pragma omp parallel for
  for (int ii = 0; ii < n; ii += TILE) {
    int idx[vecLen] __attribute__((aligned(64)));

#pragma vector aligned
    for (int i = ii; i < ii + TILE; i++)
      idx[i-ii] = (int) ( age[i] * recWidth );

    for (int c = 0; c < TILE; c++)
      // Protect the ++ operation with the atomic
      //  mutex (inefficient!)
#pragma omp atomic
      hist[idx[c]]++;
  }
```

Listing 4.37: Parallel code to compute the histogram protected with atomic operations.

The second set of bars in Figure 4.8 reports the performance result for this code. With multi-threading and atomic operations, performance on the CPU is 20 times worse than in the single-threaded case, and on the coprocessor it is 4 times worse than in the single-threaded case. The result shows that the use of atomic operations in this code is not a scalable solution. This is because every atomic operation partially serializes the execution (see Section 3.2.6). Because we execute an atomic operation in every loop iteration, this serialization, combined with the overhead of mutexes, results in worse performance than in the single-threaded case.

Mutexes are a viable solution only if they are used infrequently in an application; however, they are used too often in the histogram calculation. A different approach must be taken to parallelize this code, as discussed next.

Better Idea: Private Variables to Avoid True Sharing

As discussed in Section 3.2.7, an efficient method for avoiding synchronization between threads is parallel reduction. In our case, the code performs reduction into an array. Therefore, we cannot use the `reduction` clause of OpenMP. However, we can implement reduction by giving each thread an independent copy of array `hist` and then accumulating the results of all threads at the end of the calculation. Listing 4.38 illustrates this idea.

```
#pragma omp parallel
{ // Spawning threads before the for-loop in order to declare
  // private variables to hold a copy of histogram in each thread
  int hist_priv[m];     hist_priv[:] = 0;
  int idx[vecLen] __attribute__((aligned(64)));

  // Distribute work across threads
#pragma omp for
  for (int ii = 0; ii < n; ii += TILE) {

#pragma vector aligned
    for (int i = ii; i < ii + TILE; i++)
      idx[i-ii] = (int) ( age[i] * recWidth );

    for (int c = 0; c < TILE; c++)
      // This time, writing into the thread's private variable
      hist_priv[idx[c]]++;
  }

  // Reduce private copies into global histogram
  for (int c = 0; c < m; c++) {
    // Protect the += operation with the lightweight atomic mutex
#pragma omp atomic
    hist[c] += hist_priv[c];
  }
}
```

Listing 4.38: Parallel reduction in the histogram calculation code.

In Listing 4.38, threads are spawned with `#pragma omp parallel` before the loop begins. Variable `int hist_priv[m]` declared within the scope of that pragma is automatically considered private to each thread. In line 17, each thread writes to its own histogram copy, and no race conditions occur. Notice the absence of the word `parallel` in `#pragma omp for` in line 18: the loop is already inside of a parallel region.

After the loop in line 9 is over, the loop in line 21 is executed in each thread, accumulating the result of all calculations in the shared variable `hist`. Atomic operations are still used here. However, this time they do not incur a significant overhead. We have a total of `m`×`T` atomic increments, where `m=5`, and `T` is the number of threads. In contrast, in the flawed implementation in Listing 4.37, the code had to perform `n` atomic increments, where `n=100000000`.

The third set of bars in Listing 4.8 corresponds to the optimized version shown in Listing 4.38. The optimized parallel code performs 12 times faster than the serial code on the CPU and 110 times faster than the serial code on the coprocessor.

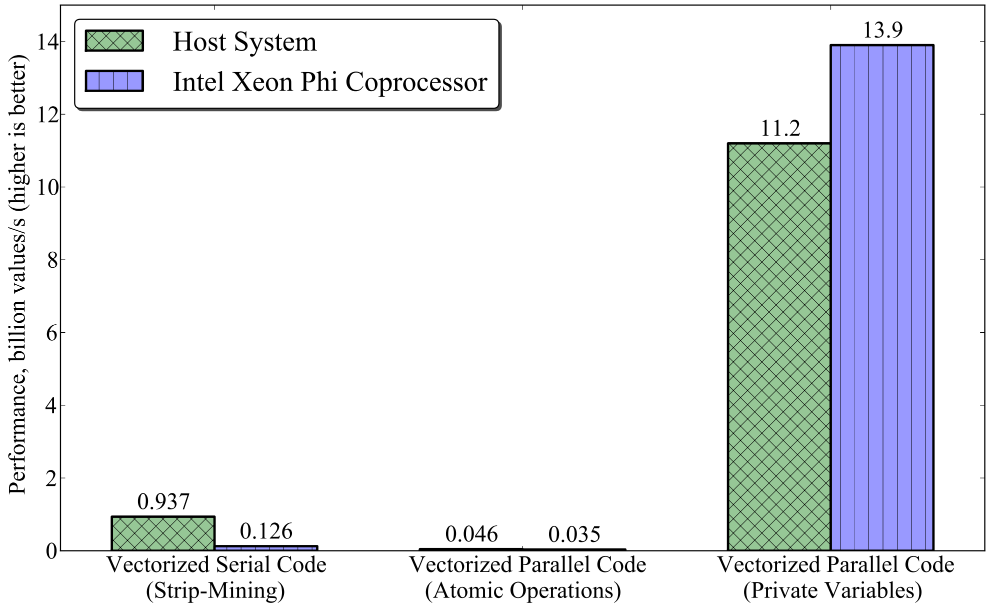


Figure 4.8: Performance of histogram calculation with thread parallelism. The first set of bars corresponds to Listing 4.35, the second to Listing 4.37 and the third to Listing 4.38

Before we finish this section, take another look at the way that thread-private storage is implemented in Listing 4.38. In this code, we make `hist_priv` private to each thread by declaring it inside the parallel region. There is an alternative way to make variables private, relying on clauses of `#pragma omp parallel`, which is illustrated in Listing 4.39.

```
int hist_priv[m];     hist_priv[:] = 0;
int idx[vecLen] __attribute__((aligned(64)));
#pragma omp parallel shared(age, hist) firstprivate(hist_priv, idx)
{
#pragma omp for
  for (int ii = 0; ii < n; ii += TILE) {

#pragma vector aligned
    for (int i = ii; i < ii + TILE; i++)
      idx[i-ii] = (int) ( age[i] * recWidth );

    for (int c = 0; c < TILE; c++)
      // This time, writing into the thread's private variable
      hist_priv[idx[c]]++;
  }

  // Reduce private copies into global histogram
  for (int c = 0; c < m; c++) {
    // Protect the += operation with the lightweight atomic mutex
#pragma omp atomic
    hist[c] += hist_priv[c];
  }
}
```

Listing 4.39: Parallel reduction (alternative implementation) in the histogram calculation code.

Approach shown in Listing 4.39 yields the same results as Listing 4.38, and both methods are legitimate in C and C++. However, in Fortran, only the method with clauses `shared`, `private` and `firstprivate` is possible.

We have achieved success with the optimization of histogram calculation, however, we will re-use this example in Section 4.4.2 to illustrate another optimization issue in multi-threaded applications known as false sharing.

4.4.2. Elimination of False Sharing with Padding

False sharing is a situation similar to a race condition, except that it occurs when two or more threads access the same *cache line* or the same block of cache in a coherent cache system (as opposed to accessing the same data element), and one of those accesses is a write. False sharing does not result in a race condition, however, it negatively impact performance.

Performance degradation occurs because the x86 architecture processors, as well as Intel Xeon Phi coprocessors, have *coherent caches*. When two or more threads access different elements in the same cache line, the processor must lock the whole cache line until the write operation is complete, and coherence is enforced. The cache line size in most modern Intel architectures is 64 bytes, and cache lines are mapped to memory on 64-byte boundaries. Therefore, if one thread is writing to memory address `A`, and another thread is reading from or writing to memory address `B`, which is within 64 bytes of `A`, false sharing may occur.

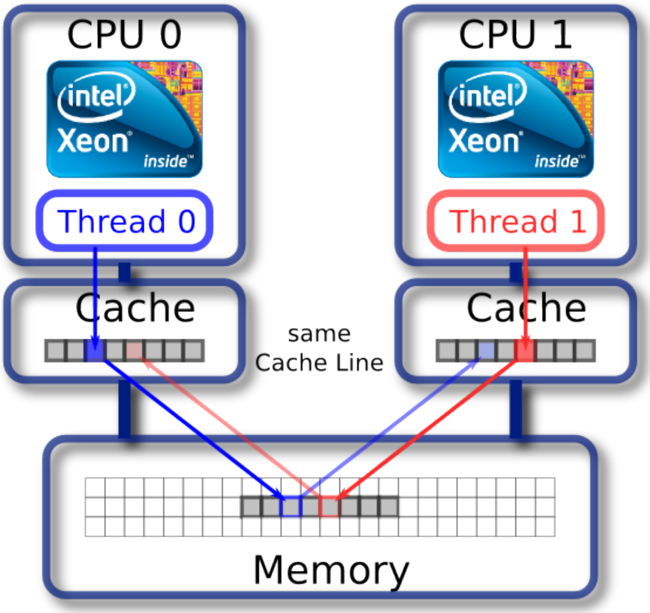


Figure 4.9: Illustration of false sharing in parallel architectures with cache coherency.

Example of False Sharing

To demonstrate false sharing, consider the histogram calculation problem discussed in Section 4.4.1. In that section, we resolved race conditions using thread-private variables (Listing 4.38). Now let's try a different method. Instead of declaring the variable `hist_priv` private to each thread, we will declare a shared array of histograms, with a separate histogram container for each thread. The code that implements this method is shown in Listing 4.40.

```
const int nThreads = omp_get_max_threads(); // Count threads
int hist_containers[nThreads][m]; // Array of histogram containers
hist_containers[:][:] = 0;
#pragma omp parallel
{
  const int thr = omp_get_thread_num(); // Query current thread
  int idx[vecLen] __attribute__((aligned(64)));

#pragma omp for
  for (int ii = 0; ii < n; ii += TILE) {
#pragma vector aligned
    for (int i = ii; i < ii + TILE; i++)
      idx[i-ii] = (int) ( age[i] * recWidth );

    for (int c = 0; c < TILE; c++)
      // Writing into current thread's own container
      hist_containers[thr][idx[c]]++; // FALSE SHARING HERE
  }
}

// Parallel region has ended, now we are back in initial thread
hist[:]=0;
for (int thr = 0; thr < nThreads; thr++)
  // Reduce own copies into global histogram; no race condition
  for (int c = 0; c < m; c++)
    hist[c] += hist_containers[thr][c];
```

Listing 4.40: Histogram calculation with a shared array of histogram containers. False sharing occurs if `m` is not a multiple of 16.

The code in Listing 4.40 may look like it should work similarly to the code in Listing 4.38, because each thread accesses its own region of memory. At the first glance, this method is even better than the method with private variables illustrated in Listing 4.38, because there are no mutexes at all in this implementation. However, in practice, code in Listing 4.40 exhibits poor performance on the host system (see the second set of bars in Figure 4.11) and on the coprocessor.

The cause of performance degradation is false sharing. The value of `m=5` is rather small, so all histogram bins in the container for thread 0, `hist_containers[0][:]`, are within `m*sizeof(int)=20` bytes of the bins in the container for thread 1, `hist_containers[1][:]`. When thread 0 and thread 1 are accessing their histogram containers simultaneously, there is a chance of hitting the same cache line or the same block of the coherent L1 cache, which results in one of the threads having to wait until the other thread unlocks that cache line. In other words, false sharing occurs because the data containers for different threads are too closely packed. The data layout in this case is illustrated in the top panel of Figure 4.10.

Padding to Avoid False Sharing

If data must be written to adjacent memory locations by different threads, false sharing situations may be avoided by padding. The amount of padding must be at least equal to size of the cache line, which is 64 bytes. Listing 4.41 shows how padding can be done in the case considered above.

```
// Padding for inner dimension of hist_containers[][]
const int paddingBytes = 64;
const int paddingElements = paddingBytes / sizeof(int);
const int mPadded = m + (paddingElements-m%paddingElements);
int hist_containers[nThreads][mPadded]; // Padded containers

// ...The rest of the code without change
```

Listing 4.41: Padding the inner dimension of the array `hist_thr` eliminates false sharing.

The only difference between Listing 4.40 and Listing 4.41 is that the inner dimension of `hist_containers` is now `mPadded` instead of `m`. Increasing the size of the inner dimension from `m` to `mPadded` separates the memory regions in which different threads operate. This reduces the penalty paid for cache coherence, which the processor cache must enforce. As a result, false sharing is eliminated.

The padded layout is illustrated in the bottom panel of Figure 4.10. In this layout, each thread-private entry for `hist_containers` is located in its own cache line. Furthermore, if the array `hist_containers` is allocated on a 64-byte boundary, then each thread-private entry begins at the start of a cache line.

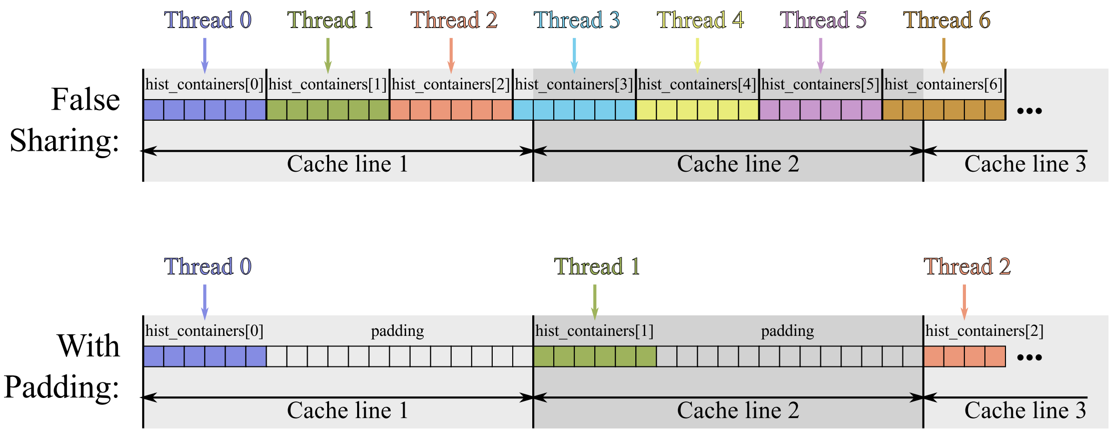


Figure 4.10: Data layout for a shared container with thread-private sections. *Top*: no padding, false sharing occurs when multple threads access the same cache line. *Bottom*: padded layout. Each private entry occupies its own cache line.

Figure 4.11 summarizes the performance results of the code in Listing 4.40 and Listing 4.41. The latter code was compiled and benchmarked in three variations: with `paddingBytes=64`, `128` and `256`. For the last case, the performance of the code is restored to the performance of the baseline code.

Note that false sharing in multi-dimensional arrays can be be avoided by padding the inner dimension if only the outer dimension index is distributed across threads. In C and C++, multi-dimensional arrays are stored in row-major format, so the inner dimension of `A[i][j]` is in `j`. In Fortran, column-major format is used, so the inner dimension of `A(i,j)` is `i`.

More information on false sharing can be found, for example, in [32].

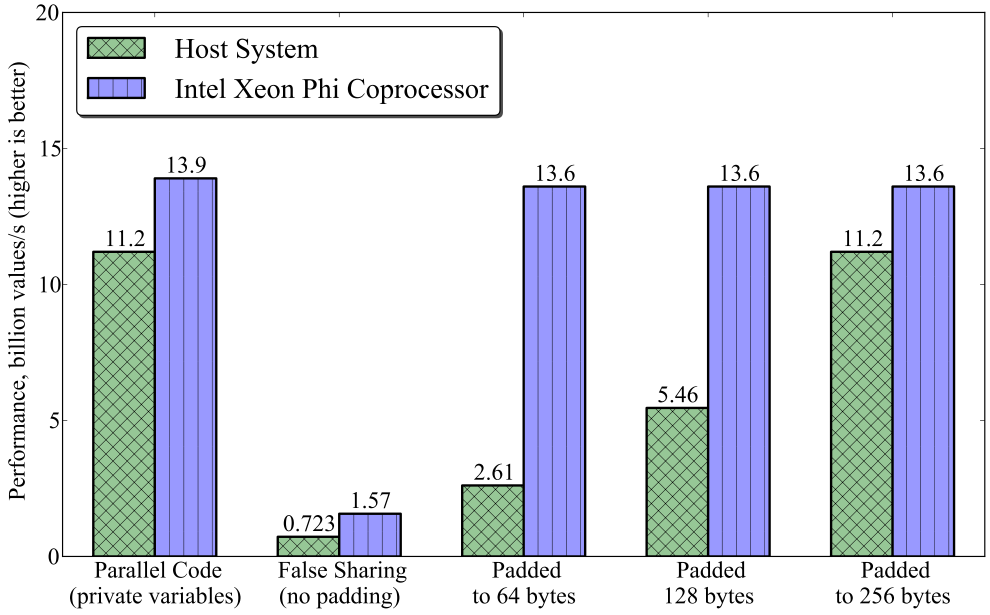


Figure 4.11: The performance of histogram calculation for n=2^{30} and m=5 using codes in Listing 4.38 ("Baseline: Parallel Code"), Listing 4.40 ("Poor Performance: False Sharing") and Listing 4.41 ("Padding to 64/128/256 bytes").

The example of histogram calculation that we have been using since Section 4.3.6 has served us well: we learned how strip-mining can facilitate automatic vectorization, how parallel reduction alleviates the overhead of synchronization, and how to eliminate false sharing with padding. Further minor performance improvements can be extracted out of this example with techniques discussed in subsequent sections. For example, refer to the Practical Exercises (Section 6.2) to see how parallel first touch allocation may improve the host performance of this code. However, to better illustrate the techniques that we discuss in the remainder of Section 4.4, let's put the histogram calculation to rest. From this point on, we will be using other example applications in our discussion.

4.4.3. Resolving Load Imbalance with Scheduling Control

In Section 3.2.3 we discussed how parallel loops can be executed in different scheduling modes. Specifically, in OpenMP, `static`, `dynamic` and `guided` modes are available, and scheduling granularity can also be specified. Choosing a scheduling mode is often a trade-off. Lightweight, coarse-grained scheduling modes incur little overhead, but may lead to load imbalance. On the other hand, complex, fine-grained scheduling modes can improve load balance, but may introduce significant scheduling overhead.

Consider a parallel for-loop that calls a thread-safe serial function in every iteration shown in Listing 4.42.

```
#pragma omp parallel for
  for (int i = 0; i < n; i++)
    BlackboxFunction(i, data[i]);
```

Listing 4.42: Sample parallel loop calling a serial function with variable execution time.

Suppose that the execution time of the function varies significantly from call to call. Such an application is prone to load imbalance because some of the parallel threads may be "lucky" to get a quick workload while other threads may struggle with a more expensive task. "Lucky" threads will have to wait for all other threads, however, the application cannot proceed further until all of the loop iterations are processed. To improve the performance, we can specify a scheduling mode and a grain size, as in Listing 4.43.

```
#pragma omp parallel for schedule(dynamic, 4)
  // ...
```

Listing 4.43: The `schedule` clause for OpenMP parallel loop may improve load balance.

Here, the `dynamic` scheduling mode indicates that the iteration space must be split into chunks of length 4, and these chunks must be distributed across available threads. As threads finish with their task, they will receive another chunk of the problem to work on. Other scheduling modes are `static`, where iterations are distributed across threads before the calculations begin, and `guided`, which is analogous to `dynamic`, except that the chunk size starts large and is gradually reduced later toward the end of the calculation.

The grain size of scheduling in Listing 4.42 is chosen as 4. Choosing the grain size is a trade-off:

- with too small a grain size, too much communication between threads and the scheduler may occur, and the application may be slowed down by the task of scheduling;
- with too large a grain size, there are few chunks, so load balancing opportunities may be limited.

In order to be effective, the grain size must be between 1 and `n/T`, where `n` is the number of loop iterations and `T` is the number of parallel threads.

Because the optimal scheduling strategy depends on the number of threads, performance portability may be an issue if the tuning parameters are hard-coded. Porting an application from multi-core Intel Xeon-based host to manycore Intel Xeon Phi coprocessor may require an adjustment in the settings of scheduling. In imbalanced problems, guided scheduling mode may often prove to be the most portable solution because of its self-adjusting nature. In well-balanced problems, static scheduling with its more predictable memory access pattern and low overhead may result in better portability.

Figure 4.12 illustrates the distribution of iterations between threads in different scheduling modes. In this hypothetical example, `n=32` iterations are distributed across `T=4` threads.

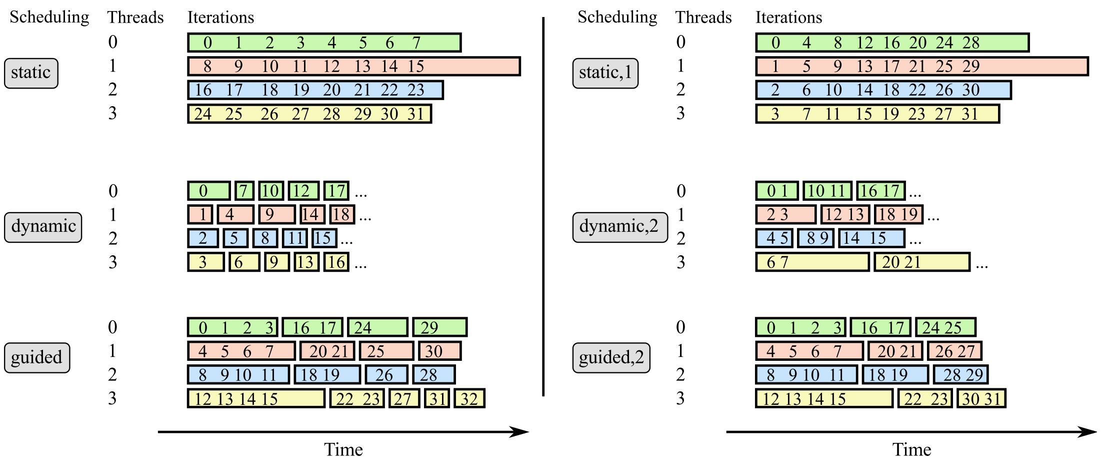


Figure 4.12: Loop scheduling modes in OpenMP.

Example of Load Imbalance Resolution

As an example, we study a function that solves a non-homogeneous system of linear algebraic equations $M\vec{x} = \vec{b}$ using the iterative Jacobi method (code of this example is available as Lab 4.06 – see Section 6.2). We construct a parallel loop that calls the Jacobi solver, and in every iteration, a different vector $\vec{b}$ is used for the problem (see Figure 4.44).

```
#pragma omp parallel for schedule(guided,4)
    for (int c = 0; c < nBVectors; c++)
      IterativeSolver(n, M, &b[c*n], &x[c*n], accuracy[c]);
```

Listing 4.44: Parallel loop that calls the Jacobi solver with different vectors $\vec{b}$ and requests a different accuracy for every call.

The iterative Jacobi method (see Listing 4.45) is inherently variable in the runtime, because for different vectors $\vec{b}$, it may take different numbers of iterations to obtain the solution $\vec{c}$ of the required accuracy. This by itself may cause load imbalance, because some lucky threads will get workload that requires few iterations, while others may get workload requiring more iterations. For the sake of demonstration, we further exacerbate the load imbalance by requesting different accuracy of solution for different loop iterations. This causes the number of Jacobi iterations to fluctuate greatly from one call of the solver to another.

We benchmarked the Jacobi solver code with various settings for the loop scheduling mode. This was done by adding the clause `schedule` to the OpenMP for-loop pragma as shown in Listing 4.44. Matrix size 256×256 was used, and a total of 20000 matrices were processed. The results can be found in Figure 4.13. On the host system, the runtime varies from 0.213 s to 0.401 s, with the default scheduling mode resulting in the worst performance. On the coprocessor, the variation in runtime is from 0.189 s to 0.557 s, with the default scheduling resulting in 0.315 s.

The "static" scheduling mode does not achieve optimum performance for the tested chunk sizes. This is not surprising because the problem has poor load balance. Notably, the case with chunk size 256 has almost double the runtime of other cases. This is because there are a total of $20000/256 \approx 78$

chunks, which is not enough to load all cores in coprocessor.

The "dynamic" and "guided" modes have runtime scheduling. They both get close to the optimum performance for some chunk sizes. However, some tuning is required to find the optimum. For low chunk sizes, there are many chunks, so load can be balanced evenly at the cost of scheduling overhead. In contrast, for large chunk sizes, there are few chunks, so load is not balanced evenly, but the scheduling overhead is low. For dynamic scheduling, there is a "sweet spot" in chunk size at the value of 4, while, with guided scheduling, a grain size of 1 works as well as a chunk size of 4. That means that guided scheduling requires less strict tuning due to its adaptive nature.

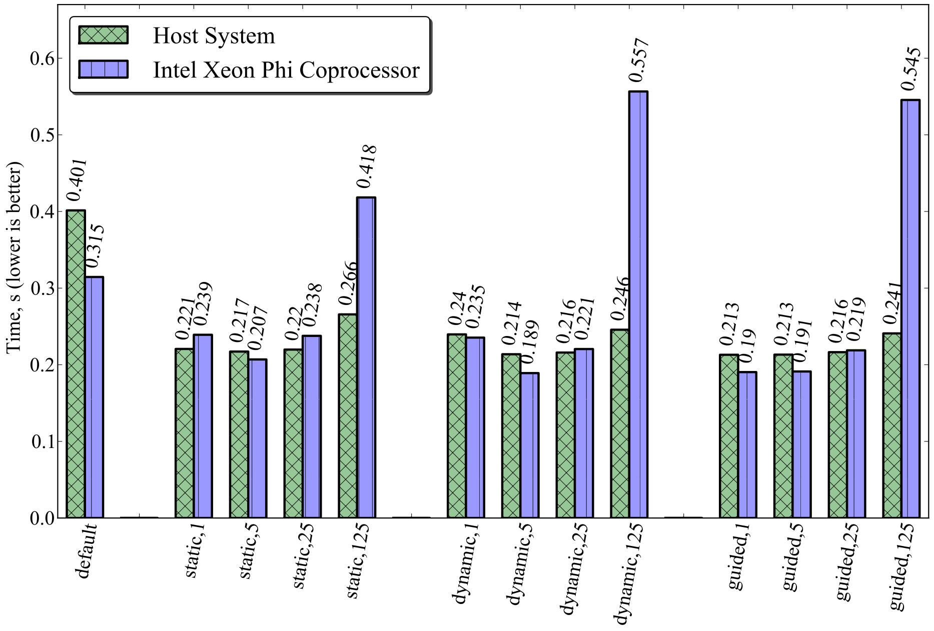


Figure 4.13: Performance of the parallel loop executing the Jacobi solver (Listing 4.45 and Listing 4.44) for a set of vectors $\vec{b}$ with various OpenMP loop scheduling modes.

For each application, the tradeoff between load balance and scheduling overhead will be achieved with different scheduling modes and chunk sizes. To test them all, the programmer may either use an OpenMP clause `schedule()` as in Listing 4.44, or by leaving the scheduling mode unspecified in the code, but setting the environment variable `OMP_SCHEDULE` to the corresponding scheduling value.

```
double RelativeNormOfDifference(const int n, const double* v1,
                                const double* v2) {
  double norm2 = 0.0;  double v1sq = 0.0;  double v2sq = 0.0;
#pragma vector aligned
  for (int i = 0; i < n; i++) {
    norm2 += (v1[i] - v2[i])*(v1[i] - v2[i]);
    v1sq += v1[i]*v1[i];  v2sq += v2[i]*v2[i];
  }
  return sqrt(norm2/(v1sq+v2sq)); // ||v1 - v2||/(||v1|| + ||v2||)
}
int IterativeSolver(const int n, const double* M, const double* b,
                    double* x, const double minAccuracy) {
  // Iteratively solves the equation Mx=b with accuracy of at
  // least minAccuracy using the Jacobi method
  double accuracy;
  double bTrial[n] __attribute__((align(64)));
  x[0:n] = 0.0; // Initial guess
  int iterations = 0;
  do {
    iterations++;
    // Jacobi method
    for (int i = 0; i < n; i++) {
      double c = 0.0;
#pragma vector aligned
      for (int j = 0; j < n; j++) {
        c += M[i*n+j]*x[j];
        x[i] = x[i] + (b[i] - c)/M[i*n+i];
      }
    }
    bTrial[:] = 0.0; // Verification
    for (int i = 0; i < n; i++) {
#pragma vector aligned
      for (int j = 0; j < n; j++)
        bTrial[i] += M[i*n+j]*x[j];
    }
    accuracy = RelativeNormOfDifference(n, b, bTrial);
  } while (accuracy > minAccuracy);
  return iterations;
}
```

Listing 4.45: Iterative Jacobi solver for non-homogeneous systems of linear algebraic equations.

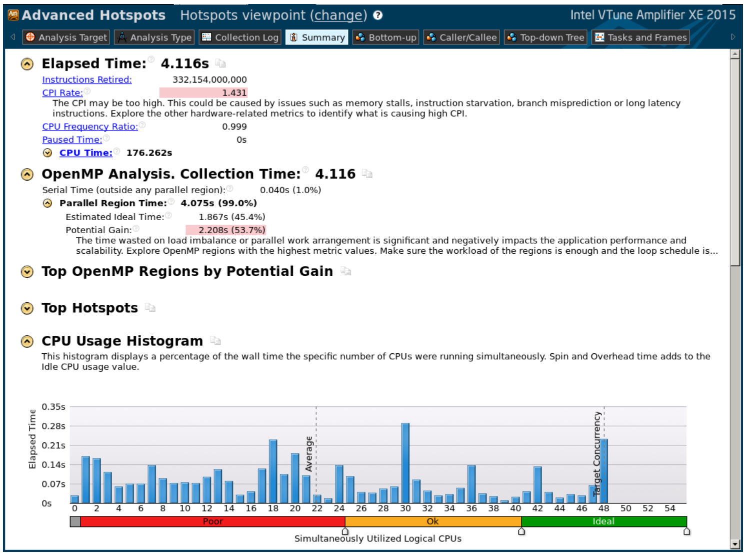
Diagnosing Load Imbalance with VTune

It is possible to detect situations where load imbalance causes performance loss in a multi-threaded application. Intel VTune Amplifier XE can be used for that (see Section 5.2 for more information about this tool). We created a VTune project for the Jacobi solver running on the host. The type of analysis used for this benchmark is called "Advanced hotspots", we ran the performance analysis on the host CPU. The benchmark was run for two versions of the code: default loop scheduling and `schedule(guided,4)`. For each code version, the parallel loop was run 10 times. Results are shown in Figure 4.14 (default scheduling) and Figure 4.15 (guided scheduling).

The top panel of Figure 4.14 is the summary information compiled by VTune. In this panel, two metrics are highlighted with red: the CPI rate and the potential gain in OpenMP region after elimination of idling time. Also, this panel contains a histogram showing the elapsed time that the code spent with different numbers of cores utilized. The data in this concurrency histogram is spread across all core counts, indicating poor concurrency.

The bottom panel of Figure 4.14 contains the bottom-up view of the analysis. Here, the prime hotspot is identified as function `IterativeSolver`, and `__kmp_wait_template` is shown to have significant imbalance or serial spinning time. At the bottom, the screenshot shows the timeline of the load in different cores. Brown color indicates running, and orange shows spin time. The orange gaps in the timelines are the telltale sign of load imbalance. They show that while some cores were done with their workload, they had to wait for other cores.

Figure 4.15 has the same information, but for optimized code with guided scheduling. In the summary screenshot the concurrency histogram has most of the measurements in the ideal range (almost all cores utilized). In the bottom-up view, `__kmp_wait_template` is gone from the top hotspots list, and the timeline is mostly all brown with tiny orange streaks at the synchronization points at the end of every iteration. This is what a well-balanced application profile should look like.



Elapsed Time: 4.116s

Instructions Retired:	332,154,000,000
CPI Rate:	1.431

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

CPU Frequency Ratio:	0.999
Paused Time:	0s
CPU Time:	176.262s

OpenMP Analysis. Collection Time: 4.116

Serial Time (outside any parallel region):	0.040s (1.0%)
Parallel Region Time:	4.075s (99.0%)
Estimated Ideal Time:	1.867s (45.4%)
Potential Gain:	2.208s (53.7%)

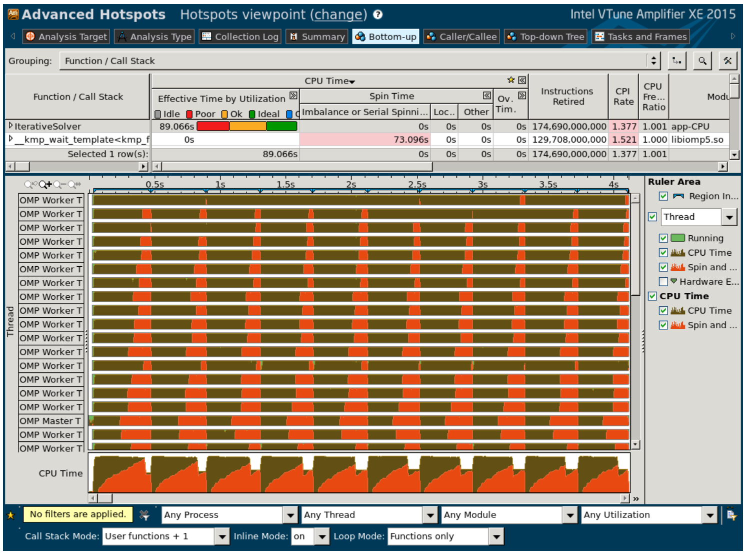
The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is...

Top OpenMP Regions by Potential Gain

Top Hotspots

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

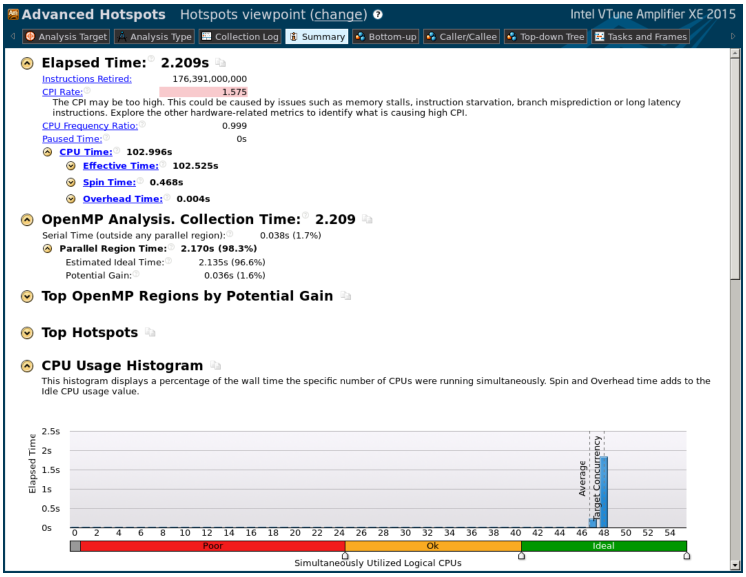


Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal)	Spin Time: Imbalance or Serial Spinni...	Loc..	Other	Ov. Tim.	Instructions Retired	CPI Rate	CPU Fre... Ratio	Modu
IterativeSolver	89.066s	0s	0s	0s	0s	174,690,000,000	1.377	1.001	app-CPU
__kmp_wait_template<kmp_f	0s	73.096s	0s	0s	0s	129,708,000,000	1.521	1.000	libiomp5.so
Selected 1 row(s):	89.066s	0s	0s	0s	0s	174,690,000,000	1.377	1.001	

Call Stack Mode: User functions + 1 Inline Mode: on Loop Mode: Functions only

Figure 4.14: *Top panel*: Summary view for VTune analysis of the Jacobi solver (Listing 4.45 and Listing 4.44) on the host system with default scheduling.
Bottom panel: bottom-up view of the same analysis.



Elapsed Time: 2.209s

Instructions Retired:	176,391,000,000
CPI Rate:	1.575

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

CPU Frequency Ratio:	0.999
Paused Time:	0s

CPU Time: 102.996s
- **Effective Time: 102.525s**
- **Spin Time: 0.468s**
- **Overhead Time: 0.004s**

OpenMP Analysis. Collection Time: 2.209

Serial Time (outside any parallel region): 0.038s (1.7%)

Parallel Region Time: 2.170s (98.3%)

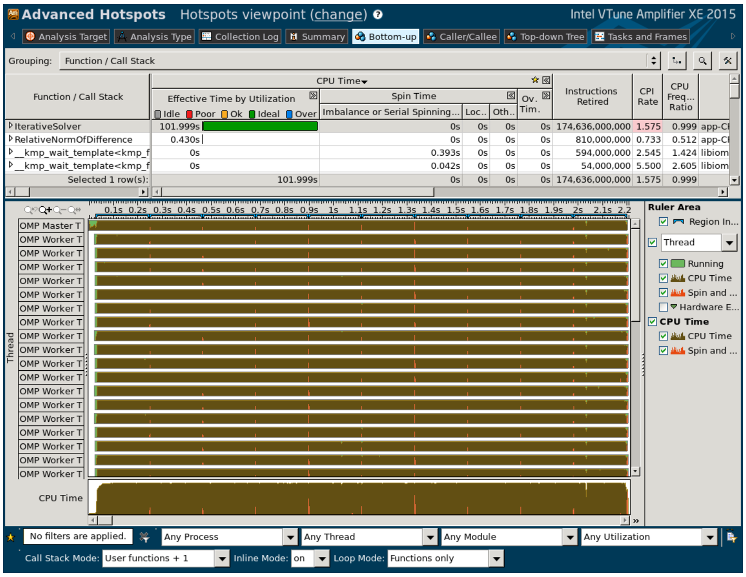
Estimated Ideal Time: 2.135s (96.6%)
Potential Gain: 0.036s (1.6%)

Top OpenMP Regions by Potential Gain

Top Hotspots

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time: Imbalance or Serial Spinning...	Spin Time: Loc..	Spin Time: Oth..	Ov. Tim.	Instructions Retired	CPI Rate	CPU Freq... Ratio	
IterativeSolver	101.999s	0s	0s	0s	0s	174,636,000,000	1.575	0.999	app-CI
RelativeNormOfDifference	0.430s	0s	0s	0s	0s	810,000,000	0.733	0.512	app-CI
__kmp_wait_template<kmp_f	0s	0.393s	0s	0s	0s	594,000,000	2.545	1.424	libiom
__kmp_wait_template<kmp_f	0s	0.042s	0s	0s	0s	54,000,000	5.500	2.605	libiom
Selected 1 row(s):	101.999s	0s	0s	0s	0s	174,636,000,000	1.575	0.999	

No filters are applied. Any Process | Any Thread | Any Module | Any Utilization

Call Stack Mode: User functions + 1 | Inline Mode: on | Loop Mode: Functions only

Figure 4.15: *Top panel*: Summary view for VTune analysis of the Jacobi solver (Listing 4.45 and Listing 4.44) on the host system with guided scheduling.
Bottom panel: bottom-up view of the same analysis.

4.4.4. Dealing with Insufficient Parallelism

Intel Xeon Phi coprocessors feature up to 61 cores each supporting 4 hardware threads in each, a total of up to 244 logical processors. Such degree of hardware parallelism may be difficult to utilize for some algorithms. For instance, algorithms in which parallel loop count is smaller than the number of logical processors will not perform well on the coprocessor. A possible solution to this problem is a modification to the parallelization strategy that increases the iteration space exposed to thread parallelism. For instance, parallelism can be re-distributed outward, from vectors to threads, or inward, from MPI processes to threads.

In this section we demonstrate two simple, yet efficient techniques for increasing available thread parallelism in an application: strip-mining and loop collapse. Both of them allow to expose more parallelism at the cost of reducing per-thread workload.

Principles: Loop Collapse

Loop collapse is a technique that converts two nested loops into a single loop. This technique can be applied either automatically (for example, using the `collapse` clause of `#pragma omp for`), or explicitly. Loop collapse is demonstrated in Listing 4.46.

```
#pragma omp parallel for
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    // m iterations are distributed across threads

#pragma omp parallel for collapse(2)
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    // m*n iterations are distributed across threads
```

Listing 4.46: Loop collapse exposes more thread parallelism in nested loops. The first piece of code does not use loop collapse; the second relies on the automatic loop collapse functionality.

It is also possible to explicitly instrument loop collapse as shown in Figure 4.47. This approach may allow tweaks that favorably interact with the scheduling modes (Section 4.4.3) and with data locality.

```
// The example below demonstrates explicit loop collapse.
// A total of m*n iterations are distributed across threads
#pragma omp parallel for
for (int c = 0; c < m*n; c++) {
  const int i = c / n;
  const int j = c % n;
  // ... do work
}
```

Listing 4.47: Explicit implementation of loop collapse.

Example: Sweep along a Short, Wide Matrix

To illustrate the usage of loop collapse and strip-mining for problems with insufficient parallelism, we consider the problem of performing a reduction (sum, average, or another cumulative characteristic) along the rows of a matrix `M[m][n]` (code for this example is available as Lab 4.05 – see Section 6.2). The principles demonstrated here are also applicable to certain stencil codes and other applications with little thread parallelism but a large amount of vector parallelism.

The task is to use `M` compute `m` scalar values defined by this equation:

$$S_i = \sum_{j=0}^{n} M_{ij}, \ i = 0 \ldots m. \tag{4.7}$$

Assume that `m` is small (smaller than the number of threads in the system), and `n` is large (large enough so that the matrix does not fit into cache). For specificity, we will use `m=4` and `n=100000000`. A straightforward implementation of summing the elements of each row is shown in Listing 4.48.

This implementation suffers from insufficient parallelism, because `m` is too small to keep all cores occupied. In fact, this is a bandwidth-bound problem, because memory access has a regular pattern, and the arithmetic intensity is equal to 1. Therefore, the performance concern is utilizing all *memory controllers*, rather than all *cores*. There are 16 memory controllers in the Knights Corner architecture. The performance of this code, expressed in GB/s (the amount of data in matrix `M` processed per second), is 44 GB/s on the host system and 10 GB/s on an Intel Xeon Phi coprocessor.

```
void SumColumns(const int m, const int n, long* M, long* s){
#pragma omp parallel for
  for (int i = 0; i < m; i++) { // m=4
    long sum = 0;
    for (int j = 0; j < n; j++) // n=100000000
      sum += M[i*n + j];
    s[i] = sum;
  }
}
```

Listing 4.48: Non-Optimized function `SumColumns()` calculates the sum of the elements in each row of matrix `M`. When the number of rows, `m`, is smaller than the number of threads in the system, the performance of this loop suffers from a low degree of parallelism.

To improve the performance of this application, the amount of exploitable parallelism in the code must be expanded. In the remainder of this section, we will implement three optimization techniques:

1. First, we will try to move the parallel pragma into the inner loop, which has more iterations.
2. Second, we will attempt to use the loop collapse functionality of OpenMP (this could work, but does not).
3. Finally, we will apply the strip-mining technique and loop collapse.

Optimization Option 1: Parallelize Inner Loop

To improve parallelism, one may try to parallelize the inner loop, which has more iterations, as shown in Listing 4.49. In this case, the number of iterations for thread parallelism is huge, so there is sufficient parallelism. Furthermore, according to the optimization report (Listing 4.50), the compiler is able to make multi-threading in the `j`-loop to co-exist with vectorization.

```
void SumColumns(const int m, const int n, long* M, long* s){
  for (int i = 0; i < m; i++) { // m=4
    long sum = 0;
    // Parallelizing the inner loop to have more thread parallelism
#pragma omp parallel for reduction(+: sum)
    for (int j = 0; j < n; j++) // n=100000000
      sum += M[i*n+j];
    s[i] = sum;
  }
}
```

Listing 4.49: Function `SumColumns()` with thread parallelism applied to inner loop instead the outer loop (compare to Listing 4.48).

```
...
OpenMP Construct at worker.cc(5,1)
   remark #16200: OpenMP DEFINED LOOP WAS PARALLELIZED
...
LOOP BEGIN at worker.cc(5,1)
   remark #25084: Preprocess Loopnests: Moving Out Store
                                                    [ worker.cc(7,7) ]
   remark #15300: LOOP WAS VECTORIZED
...
LOOP END
...
```

Listing 4.50: Vectorization report for Listing 4.49. Thread parallelism in the `j`-loop co-exists with vectorization.

Note that in order to properly perform reduction (summation) along the row, we had to use the clause `reduction(+: sum)` for variable `sum` in the OpenMP pragma. This is because with parallelized `i`-loop, different threads will be contributing to the total value of `s[i]`.

With parallel inner loop, the performance on the coprocessor jumps up to 162 GB/s. It also marginally improves on the host to 45 GB/s.

Even though we are observing good acceleration with an Intel Xeon Phi coprocessor, there is an indication that this is not the optimum performance. The value of performance on the host, 45 GB/s, is considerably lower than the host system bandwidth measured by the STREAM benchmark. STREAM achieves 64 GB/s for the "copy" test and 85 GB/s for the "triad" test on our host system.

There are two reasons for sub-optimal performance:

1. The OpenMP threads are spawned and terminated in every `i`-iteration, which incurs parallelization overhead. This may be a minor effect for our parameters of choice (`m=4` and `n=100000000`), however, the larger the value of `m` and the smaller the value of `n`, the greater performance penalty will this inefficiency cause.

2. When the inner loop is parallelized, the OpenMP library does not see the whole scope of the data processed by the problem, and has less freedom for optimal load scheduling.

Even though with parallelized inner loop, we observed a performance increase on the coprocessor, we will mark this method as sub-optimal for this problem because of the problems stated above. Let's see if we can resolve the issue of insufficient parallelism in this code without re-spawning the parallel region multiple times, and still exposing the whole problem to thread and vector parallelism.

Optimization Option 2 (Inefficient): Loop Collapse

In an attempt to instrument a better algorithm, the programmer can use the OpenMP functionality of loop collapse, as shown in Listing 4.51.

```
void SumColumns(const int m, const int n, long* M, long* s){
  s[0:m] = 0;
#pragma omp parallel
  {
    // Each thread will need a private container
    long sum[m];    sum[:] = 0;
    // Loop collapse expands iteration space:
    // distributing m*n iterations across all threads
#pragma omp for collapse(2)
    for (int i = 0; i < m; i++) // m=4
      for (int j = 0; j < n; j++) // n=100000000
        sum[i] += M[i*n+j];
    // Reducing from thread containers to the output array
    for (int i = 0; i < m; i++)
#pragma omp atomic
      s[i] += sum[i];
  }
}
```

Listing 4.51: Row-wise matrix reduction with an attempt to expand the iteration space by collapsing nested loops.

```
OpenMP Construct at worker.cc(3,1)
   remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED
...
LOOP BEGIN at worker.cc(9,5)
...loop was not vectorized: vector dependence prevents vectorization
LOOP END
```

Listing 4.52: Vectorization report for Listing 4.51. Loop collapse precludes vectorization.

With loop collapse, having a scalar variable `sum` to avoid race conditions is no longer sufficient. That is because in Listing 4.49, `sum` had a separate instance for each value of `i`. Now, with collapsed loops in Listing 4.51, different threads may be operating on the same value of `i`. To avoid data races, we gave each thread a private container `long sum[m]`. After the parallel run, all thread-private containers are reduced into the shared container `s` using atomic operations. Such a procedure for reduction was demonstrated in Section 4.4.1.

While the `collapse(2)` directive makes OpenMP expand the iteration space into two loops, the code works slowly on both the host system and the coprocessor. On the host, we observer 46 GB/s and on the coprocessor only 8 GB/s.

Even though we did not achieve optimal performance with this optimization, we are on the right track, because we expose the most parallelism to the compiler.

The failure of the compiler to vectorize collapsed thread-parallel loops is not due to a fundamental property of this algorithm, but due to the complicated nature of code analysis in such case. In the next optimization step, we will assist the compiler by increasing the amount of nesting with the strip-mining technique. This will allow us to retain vectorization in the inner loop, at the same time exposing the whole iteration space to thread parallelism. Strip-mining was previously discussed in Section 4.3.6.

Optimization Option 3 (Optimal): Strip-Mining and Loop Collapse

Finally, consider the code in Listing 4.53, which employs strip-mining in order to transform the `j`-loop into two nested loops, and utilizes the loop collapse directive on the outer `i`-loop and the new `jj`-loops.

```
1  void SumColumns(const int m, const int n, long* M, long* s){
2    const int tile = 10000;
3    assert(n%tile == 0);
4    s[0:m] = 0;
5  #pragma omp parallel
6    {
7      // Each thread will need a private container
8      long sum[m];    sum[:] = 0;
9      // Loop collapse expands iteration space:
10     // distributing m*(n/tile) iterations across threads
11 #pragma omp for collapse(2)
12     for (int i = 0; i < m; i++)
13       for (int jj = 0; jj < n; jj+=tile)
14         for (int j = jj; j < jj+tile; j++)
15           sum[i] += M[i*n+j];
16     // Reducing from thread containers to the output array
17     for (int i = 0; i < m; i++)
18 #pragma omp atomic
19       s[i] += sum[i];
20   }
21 }
```

Listing 4.53: This code improves on the version in Listing 4.51 by strip-mining the inner loop. This allows OpenMP to balance the load across available threads, while automatic vectorization succeeds in the inner loop.

This code retains the structure designed in Listing 4.51 and adds to it another level of nesting. The compilation report now indicates that the loop in line 14 is vectorized. The length of the inner loop, expressed with variable `tile`, is chosen empirically. It has to be long enough to allow sufficient workload in each thread, and it should also be a multiple of the cache line to ensure aligned access to data in every instance of the `j`-loop. We assume that `n` is a multiple of `tile`. If it is not, the programmer must modify the bounds of the `jj`-loop and add a peel loop at the end of the code (see Listing 4.58).

Note that even though we have an atomic construct in line 18, we do not have the kind of penalty for it that we observer in Section 4.4.1. That is because in Listing 4.53, the atomic operation is performed outside the innermost loop in line 14. We will have a total of `m × T` atomic additions, where `T` is the number of threads, which amounts to approximately 1000. In contrast, the innermost loop in line 14 performs `m × n` additions, which amounts to hundreds of millions of operations. Having an atomic operation in that loop would have been disastrous for performance.

Figure 4.16 contains a summary of the performance of all versions of the row-wise matrix reduction algorithm considered in this section.

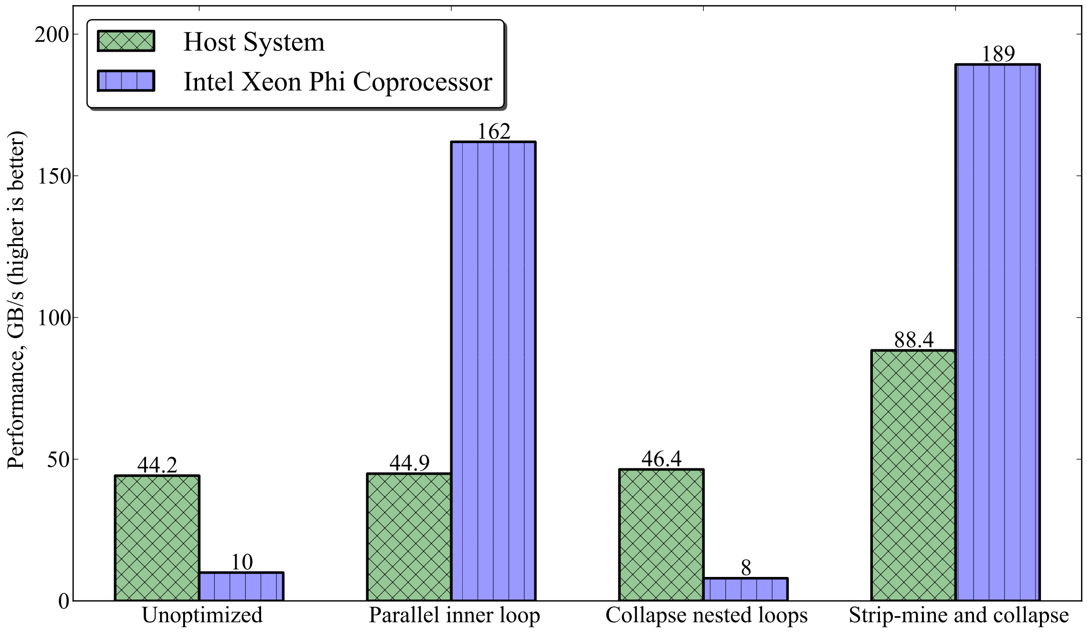


Figure 4.16: Performance of all versions of the row-wise matrix reduction code (Listing 4.48, Listing 4.49, Listing 4.51 and Listing 4.53) on the host system and on the Intel Xeon Phi coprocessor.

Evidently, our last optimization attempt is the best of all cases we had considered so far. It achieves 88 GFLOP/s on the host and 189 GFLOP/s on the coprocessor. Both values are greater than the values achieved by the STREAM benchmark on the respective platforms in any of its four tests. This indicates that we are very near the optimum performance.

Note that the STREAM benchmark performs both reads and writes, and our code performs only reads. This is why it was able to achieve greater bandwidth than STREAM.

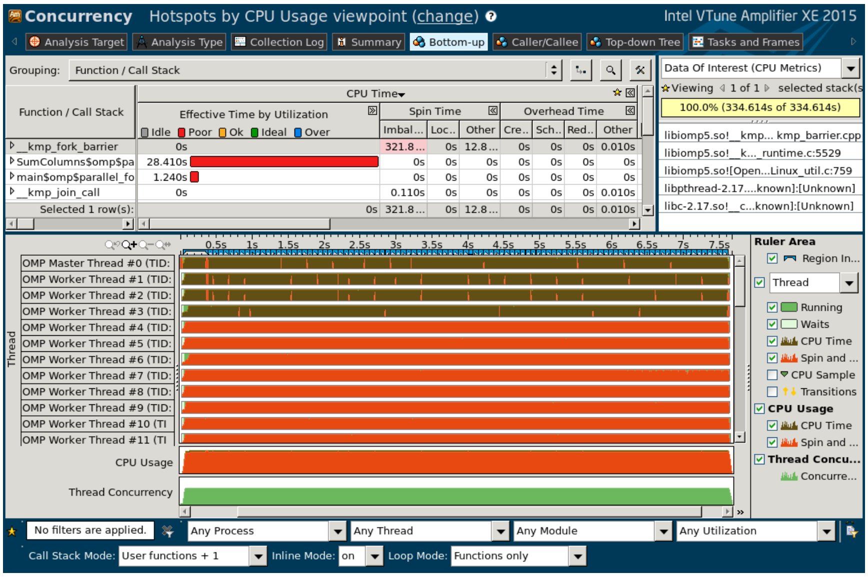
Diagnosing Insufficient Parallelism with VTune

Insufficient parallelism may be seen in the VTune profile of the application. We captured screenshots of VTune analysis of the non-optimized (Listing 4.48) and optimized (Listing 4.53) matrix sweep codes in Figure 4.17. These analyses were performed on the host with the analysis type "Concurrency" (however, "Advanced Hotspots" would have yielded the same results).

The bottom-up list of hotspots in the non-optimized version (top panel of Figure 4.17) has function `__kmp_fork_barrier` on top, taking almost 10 more CPU time than the main function `SumColumns()`. Having an OpenMP function at the top of the hotspots list is a red flag in itself pointing to issues with thread parallelism. The red band in front of `SumColumns` in the column "Effective Time by Utilization" indicates that most of the time, thread concurrency in this function was in the "Poor" range. The timeline of the non-optimized code confirms these indicators of multi-threading inefficiency, showing that only 4 threads were working (brown stripes) while all other threads were spinning (orange stripes).

After optimization (bottom panel of Figure 4.17), the OpenMP function `__kmp_fork_barrier` is still among the top hotspots, however, now it takes just over 10% of the CPU time. The main function, `SumColumns()`, is now the top hotspot, with the majority of its runtime spent in the "Ideal" range of thread concurrency (this is shown by the extent of the green band in the "Effective Time by Utilization" column). The timeline in this screenshot also shows good thread utilization: all bands are mostly brown with intermittent minor orange streaks.

The orange streaks in the optimized code timeline point to spin time. This is not unusual in memory bandwidth-bound applications, which this one is. Metrics designed for compute-bound applications, such as CPI (cycles per instructions) ratio and spin time, may appear abnormally high in VTune, even though the achieved bandwidth is optimal.



Concurrency Hotspots by CPU Usage viewpoint (change) — Intel VTune Amplifier XE 2015

Grouping: Function / Call Stack

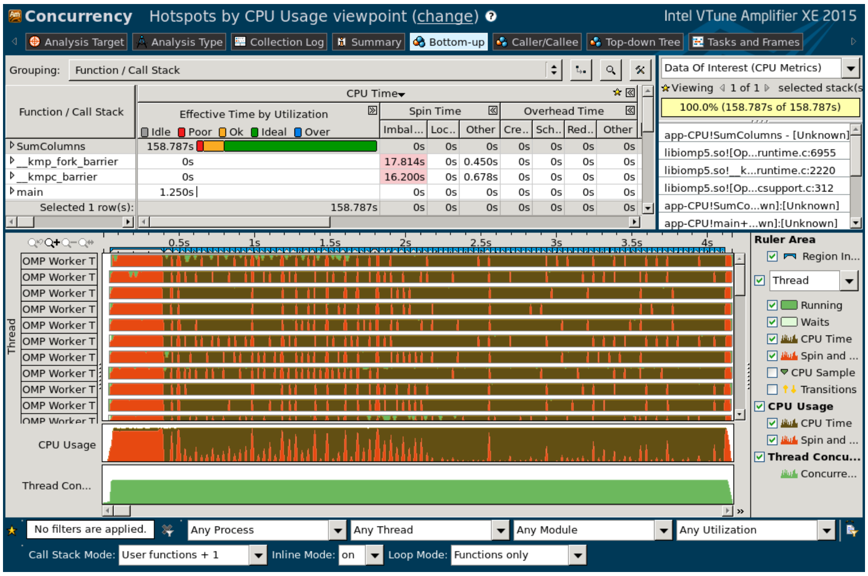
Function / Call Stack	Effective Time by Utilization	Spin Time: Imbal...	Loc..	Other	Overhead Time: Cre..	Sch..	Red..	Other
__kmp_fork_barrier	0s	321.8...	0s	12.8...	0s	0s	0s	0.010s
SumColumns$omp$pa	28.410s	0s	0s	0s	0s	0s	0s	0s
main$omp$parallel_fo	1.240s	0s	0s	0s	0s	0s	0s	0s
__kmp_join_call	0s	0.110s	0s	0s	0s	0s	0s	0.010s
Selected 1 row(s):	0s	321.8...	0s	12.8...	0s	0s	0s	0.010s

Data Of Interest (CPU Metrics)
Viewing 1 of 1 selected stack(s)
100.0% (334.614s of 334.614s)

libiomp5.so!__kmp... kmp_barrier.cpp
libiomp5.so!__k..._runtime.c:5529
libiomp5.so![Open...Linux_util.c:759
libpthread-2.17....known]:[Unknown]
libc-2.17.so!__c...known]:[Unknown]

OMP Master Thread #0 (TID:
OMP Worker Thread #1 (TID:
OMP Worker Thread #2 (TID:
OMP Worker Thread #3 (TID:
OMP Worker Thread #4 (TID:
OMP Worker Thread #5 (TID:
OMP Worker Thread #6 (TID:
OMP Worker Thread #7 (TID:
OMP Worker Thread #8 (TID:
OMP Worker Thread #9 (TID:
OMP Worker Thread #10 (TI
OMP Worker Thread #11 (TI
CPU Usage
Thread Concurrency

No filters are applied. Any Process, Any Thread, Any Module, Any Utilization
Call Stack Mode: User functions + 1; Inline Mode: on; Loop Mode: Functions only



Concurrency Hotspots by CPU Usage viewpoint (change) — Intel VTune Amplifier XE 2015

Grouping: Function / Call Stack

Function / Call Stack	Effective Time by Utilization	Spin Time: Imbal...	Loc..	Other	Overhead Time: Cre..	Sch..	Red..	Other
SumColumns	158.787s	0s	0s	0s	0s	0s	0s	0s
__kmp_fork_barrier	0s	17.814s	0s	0.450s	0s	0s	0s	0s
__kmpc_barrier	0s	16.200s	0s	0.678s	0s	0s	0s	0s
main	1.250s	0s	0s	0s	0s	0s	0s	0s
Selected 1 row(s):	158.787s	0s	0s	0s	0s	0s	0s	0s

Data Of Interest (CPU Metrics)
Viewing 1 of 1 selected stack(s)
100.0% (158.787s of 158.787s)

app-CPU!SumColumns - [Unknown]
libiomp5.so![Op...runtime.c:6955
libiomp5.so!__k...runtime.c:2220
libiomp5.so![Op...csupport.c:312
app-CPU!SumCo...wn]:[Unknown]
app-CPU!main+...wn]:[Unknown]

No filters are applied. Any Process, Any Thread, Any Module, Any Utilization
Call Stack Mode: User functions + 1; Inline Mode: on; Loop Mode: Functions only

Figure 4.17: Thread concurrency profile of host implementation of the row-wise matrix reduction code. *Top panel*: non-optimized code (Listing 4.48). *Bottom panel*: optimized code with strip-mining and loop collapse (Listing 4.53).

4.4.5. Thread Affinity Optimization

The Intel OpenMP library has the ability to bind OpenMP threads to a set of logical or physical cores. This functionality is available in both Intel Xeon processors and Intel Xeon Phi coprocessors. Such binding, known as *thread affinity*, may improve application performance. For example, it makes sense to use thread affinity in the following cases:

1. **In HPC applications that utilize the whole system**, OpenMP threads may migrate from core to another according to OS decisions. This leads to performance penalties because the migrated thread must fetch the cache contents into the new core's L1 cache. Using thread affinity, the programmer can forbid migration and thus improve performance as well as dramatically boost power efficiency (see [33]);
2. **For memory bandwidth-bound applications**, best performance is usually achieved with one software thread per physical core. In this case, thread affinity must place consecutive software threads on different physical cores, skipping additional hyper-threads (in Intel Xeon CPU) or hardware threads (in Intel Xeon Phi coprocessor) in that core;
3. **For compute-bound applications**, the best performance is usually achieved with multiple threads per core. Thread affinity for these applications must bind software threads to logical processors (hyper-threads in CPU or hardware threads in MIC architecture). Ordering may be important: threads with adjacent numbers are likely to work on adjacent data subsets, so they should be placed on the same physical core to share the cache (L1 cache in Intel Xeon or L1 and L2 caches in Intel Xeon Phi architecture);
4. **In offload applications for Intel Xeon Phi coprocessors**, core 0 manages offload tasks, so calculations should not be scheduled on it. Intel OpenMP excludes core 0 from thread affinity mask on Xeon Phi;
5. **In Non-Uniform Memory Access (NUMA) systems** (platforms with two- or four-way Intel Xeon processors), thread affinity can be used to partition the system between independent processes, sharing the compute node's resources. For example, in a two-way NUMA system with two processes, thread affinity must bind threads of one process to the first socket and threads of the other process to the second socket.

The KMP_AFFINITY Environment Variable

Thread affinity in OpenMP applications can be controlled at the application level by setting the environment variable KMP_AFFINITY. A detailed explanation of the format of the variable is given in Listing 4.54. Table 4.6 explains the meaning of the arguments.

```
KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

Listing 4.54: Syntax of thread affinity settings with the environment variable KMP_AFFINITY.

In the value of KMP_AFFINITY, the key parameter is `type`. It determines the pattern of thread distribution across cores. Usually, it is set to `compact` or `scatter`. `compact` means that each thread is placed as close as possible to to the previous one, and `scatter` means that each thread is placed as far as possible from the previous one. Additionally, on an Intel Xeon Phi coprocessor, type `balanced` is supported, which is useful for 2 or 3 threads per core. Finally, type my be set to `explicit`, which means that instead of a pattern, threads will be bound according to an explicit proclist.

Other arguments of KMP_AFFINITY are `<modifier>`, of which the most useful are `verbose` (OpenMP will print on the screen the affinity mask) and `granularity=fine` or `granularity=core`. `fine` granularity binds threads to specific hyper-threads/hardware threads, and `core` binds threads to physical cores, permitting migration between hyper-threads/hardware threads.

Finally, `permute` and `offset` are integer arguments. The former indicates that the in the affinity pattern, the order of levels in the machine topology map must be permuted (useful, for example, for binding consecutive threads to consecutive cores instead of consecutive hyper-threads). The latter indicates the starting position for thread assignment (useful for partitioning the system between different processes).

As of Intel Parallel Studio XE 2015 Update 1, the default affinity type on Intel Xeon processors is `KMP_AFFINITY=none`, and on Intel Xeon Phi coprocessors `KMP_AFFINITY=scatter`.

Argument	Default	Description
`modifier`	`nonverbose,` `respect,` `granularity=core`	`verbose/nonverbose` control whether the application must print information about the supported affinity upon OpenMP library initialization: number of packages (i.e., processors), number of cores in each package, number of thread contexts in each core, and OpenMP thread bindings to physical thread contexts. `respect/norespect` control whether to respect the affinity mask in place for the thread that initializes the OpenMP run-time library `warnings/nowarnings` control whether to print warning messages for the affinity interface `granularity=<specifier>` describes the lowest level that OpenMP threads are allowed to float within a topology map. The values of `<specifier>` are `core` or `fine` (the latter is equivalent to `thread`). With `granularity=core`, threads bound to a core are allowed to float between the different thread contexts (logical processors). With `granularity=fine`, each thread is bound to a specific thread context. `proclist=[<proc_list>]` specifies an explicit mapping of OpenMP threads to OS procs. The format of `<proc_list>` is a comma-separated string containing the numerical identifiers of OS procs or their ranges, and float lists enclosed in brackets `{}`. Example: `proclist=[7,4-6,{0,1,2,3}]` maps OpenMP thread 0 to OS proc 7, threads 1, 2 and 3 to procs 4, 5 and 6, respectively, and thread 4 is allowed to float between procs 0, 1, 2 and 3.
`type`	`none`	`type=compact` assigns each OpenMP thread to a thread context as close as possible to the previous thread. This type is beneficial for compute-intensive calculations. `type=scatter` is the opposite of `compact`: OpenMP threads are distributed as evenly as possible across the system. This type is beneficial for bandwidth-bound applications. `type=balanced` is supported only in the MIC architecture and is a compromise between `scatter` and `balanced`. `type=explicit` assigns thread affinity according to the list specified in the `proclist=` modifier. `type=disabled` completely disables affinity interface and forces the OpenMP library to behave as if the affinity interface was not supported by the operating system `type=none` does not bind OpenMP threads to particular thread contexts. Compiler still uses the OpenMP thread affinity interface to determine machine topology, unlike with `type=disabled`.
`permute`	`0`	For `compact` and `scatter` affinity maps, controls which levels are most significant when sorting the machine topology map. A value for permute forces the mappings to make the specified number of most significant levels of the sort the least significant, and it inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.
`offset`	`0`	indicates the starting position (proc ID) for thread assignment.

Table 4.6: Arguments of the `KMP_AFFINITY` environment variable. This summary table is based on the complete description in the Intel C++ Compiler Reference Guide.

Low-level Affinity Interface

In addition to using the variable KMP_AFFINITY, the programmer may use the low-level affinity interface supported by Intel Compilers. That interface allows to set affinity using function calls from the application code. Functions implementing this interface support only explicit processor lists, and do not support patterns similar to scatter and compact. Low-level affinity interface has precedence over the high-level method with the environment variable KMP_AFFINITY. See the Compiler Reference for more information.

OS Procs

Using KMP_AFFINITY=verbose can help to determine the runtime mapping of threads to sockets and cores. Diagnostic output generated by this variable also reveals that there is an intermediate mapping used by OpenMP for affinity settings, which uses OS procs to enumerate threads. OpenMP maps software threads to OS procs, and the operating system maps OS procs to logical processors. The mapping of OS procs can be queried by viewing the Linux system file /proc/cpuinfo. The lines beginning with the word "processor" in this file contain the OS procs numbers.

Affinity on Coprocessors and KMP_PLACE_THREADS

To set the number of OpenMP threads in an application, the variable OMP_NUM_THREADS may be used. On coprocessors, a more descriptive variable KMP_PLACE_THREADS can be used to specify not only the total number of threads, but also the number of threads per core. The syntax is

```
KMP_PLACE_THREADS=<N>C,<M>t
```

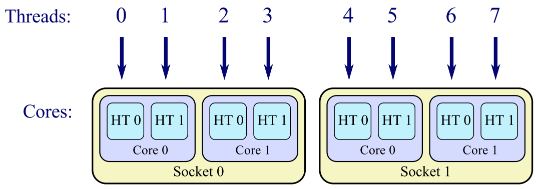
where <N> is an integer specifying the number of cores and <M> is the number of threads per core. For example, KMP_PLACE_THREADS=60C,3t creates 180 threads, distributing 3 threads per core across 60 cores. Usage of KMP_PLACE_THREADS does not replace KMP_AFFINITY, but complements it. One of the advantages of using it is that when the user requests one core less than the number of physical cores, core 0 on coprocessor will be automatically excluded from the affinity mask. This is beneficial for offload applications, because core 0 is used for data movement tasks.

KMP_AFFINITY Usage Examples

Examples of usage are given below. We assume a two-way processor with 2 cores per socket and enabled hyper-threading (a total of 4 physical cores or 8 logical processors):

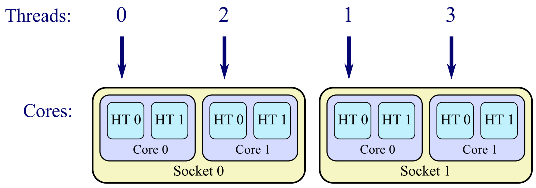
1) To set a compute-optimal affinity for a single process:

```
export OMP_NUM_THREADS=8
export KMP_AFFINITY=compact,granularity=fine
```



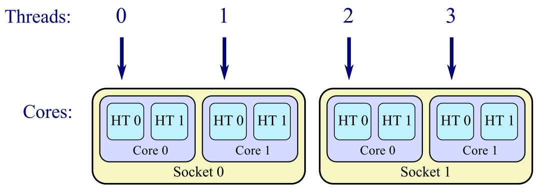
2) To set a bandwidth-optimal affinity for a single process:

```
export OMP_NUM_THREADS=4
export KMP_AFFINITY=scatter,granularity=fine
```



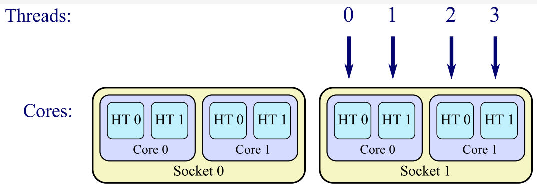
3) To bind threads consecutively to physical cores:

```
export OMP_NUM_THREADS=4
export KMP_AFFINITY=compact,granularity=fine,1,0
```



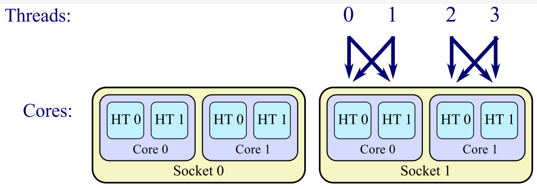
4) To bind threads to the second CPU socket:

```
export OMP_NUM_THREADS=4
export KMP_AFFINITY=compact,granularity=fine,0,4
```



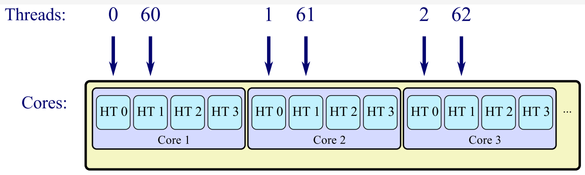
5) Same as previous, but allow migration within a core:

```
export OMP_NUM_THREADS=4
export KMP_AFFINITY=compact,granularity=fine,1,0
```



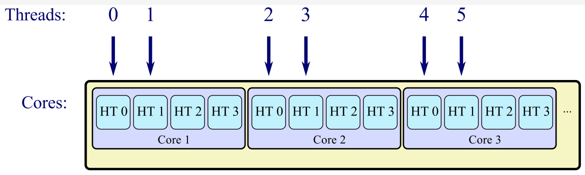
6) In offload, bind 2 threads per core on a 61-core coprocessor with pattern "scatter", excluding core 0 from affinity mask:

```
export MIC_ENV_PREFIX=XEONPHI
export XEONPHI_KMP_PLACE_THREADS=60C,2t
export XEONPHI_KMP_AFFINITY=scatter,granularity=fine
```



7) Same as previous, but with pattern "compact":

```
export MIC_ENV_PREFIX=XEONPHI
export XEONPHI_KMP_PLACE_THREADS=60C,2t
export XEONPHI_KMP_AFFINITY=compact,granularity=fine
```



Example: Compute-Bound Application Tuning, DGEMM

For applications in which cache utilization and arithmetic performance are more important than memory bandwidth, it is beneficial to place threads close to each other on the processors. Affinity of type `compact` usually works best for compute-bound application. Failing to set affinity can be expensive, because by default, on processors, affinity is not set, and on coprocessors it defaults to `scatter`.

As an illustration, we benchmarked the Intel MKL implementation of Double-precision General Matrix-Matrix Multiply (DGEMM) on matrices of size 8000×8000. The application was run on the host and, independently, on the coprocessor as a native executable. Code for this example is available in Lab 4.07 – see Section 6.2. Figure 4.18 demonstrates our results.

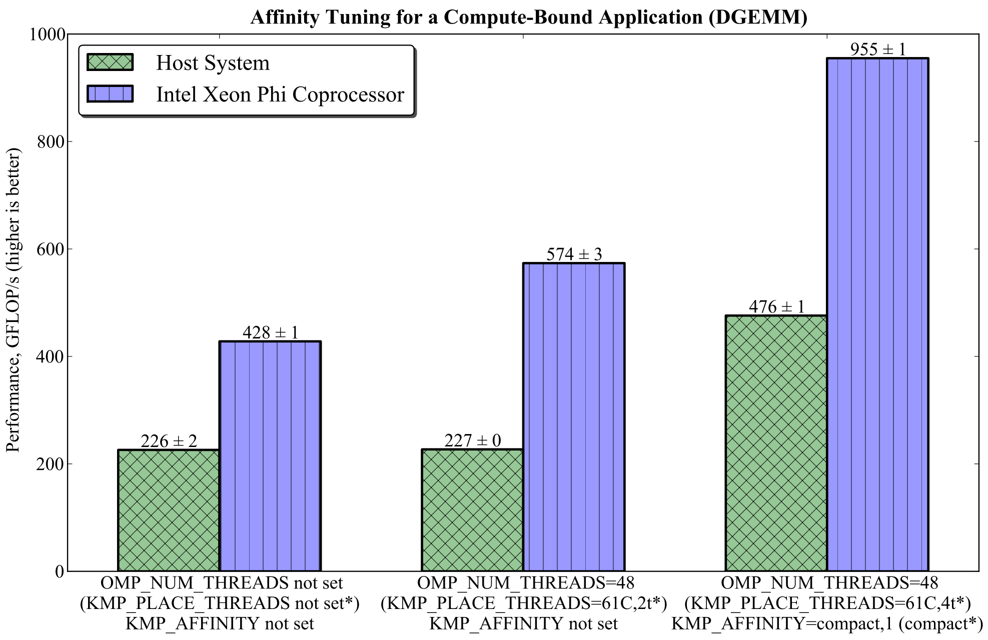


Figure 4.18: Performance of DGEMM with and without affinity settings. This example illustrates the effect of thread affinity on compute-bound applications on processors and coprocessors. * Values with an asterisk apply to configuration on the Intel Xeon Phi coprocessor.

First, we ran the application without setting affinity or the number of threads. In this case, thread assignment was set by the OpenMP library to the default values: 48 threads on the host with no affinity, and 244 threads on the coprocessor with affinity of type `scatter`. With these settings, the host yielded 226 ± 2 GFLOP/s and the coprocessor 428 ± 1 GFLOP/s.

The second step was tuning the number of threads. On our 24-core host with 2-way hyper-threading, the natural choice is between 48 and 24 threads. We have found that setting `OMP_NUM_THREADS=24` and `48` yields the same performance. On the 61-core coprocessor, we tested different numbers of threads using `KMP_PLACE_THREADS` instead of `OMP_NUM_THREADS`. Reasonable values to try are `61C,4t`, `61C,3t`, `61C,2t` and `61C,1t`. The best results were obtained with `61C,2t` (using all cores, 2 threads per core), and DGEMM achieved 574 ± 3 GFLOP/s.

Finally, we tuned the thread affinity type together with the number of threads. On the host, the best result is obtained by using either the combination: `OMP_NUM_THREADS=24`, `KMP_AFFINITY=compact` or the combination `OMP_NUM_THREADS=48`, `KMP_AFFINITY=compact,1`. This result is 476 ± 1 GFLOP/s. On the coprocessor, the best result was achieved with thread affinity `compact` and 4 threads per core (`61C,4t`), and this result is 955 ± 1 GFLOP/s, which is close to the theoretical peak performance of the 7120P coprocessor.

Example: Bandwidth Tuning and KMP_AFFINITY=scatter

In applications bound by memory bandwidth, it is usually beneficial to use 1 thread per core on processors and 1-2 threads per core on the coprocessor. This reduces thread contention on memory controllers. Additionally, affinity type scatter improves the effective bandwidth because all memory controllers are utilized uniformly.

Tuning thread affinity for bandwidth typically yields smaller speedup than tuning for compute intensity (see previous example). That is because on coprocessors, the default affinity is scatter, so even without setting KMP_AFFINITY, the application uses optimal affinity type. On host processor, by default, affinity is not set, however, its hardware-rich architecture is able sustain good memory traffic even without thread affinity.

To test affinity tuning, we implemented a code that copies one array into another using an OpenMP parallel loop (Listing 4.55). This workload is similar to the "copy" test of the STREAM benchmark [34]. Code for this example is available in Lab 4.07 – see Section 6.2.

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
  B[i] = A[i];
```

Listing 4.55: Microkernel for testing memory bandwidth with array copy.

We tested array copy for array size n=320,000,000 and element type double. The application first ran the test on the host. Then the same application performed offload and ran the benchmark on the coprocessor. Results are shown in Listing 4.19.

The first attempt to run the application, without affinity settings, yields good result "out of the box": 69 ± 1 GB/s on the host and 151 ± 1 GB/s on the coprocessor. This is not surprising, because, as mentioned above, the coprocessor uses a favorable affinity type scatter by default, and the bandwidth on the host is not very sensitive to affinity settings. These results are close to the highly-tuned STREAM benchmark values [35].

However, we can do 10% better on the host by setting one thread per core (OMP_NUM_THREADS=24) and KMP_AFFINITY=scatter for this application. As the second set of bars in Figure 4.19 shows, this increases host bandwidth to 77 ± 1 GB/s. However, at the same time, the coprocessor bandwidth drops to 98 ± 2 GB/s. This is an inadvertent effect of environment variable forwarding to the offloaded code: variable `OMP_NUM_THREADS` propagated to the coprocessor and resulted in only 24 threads operating there.

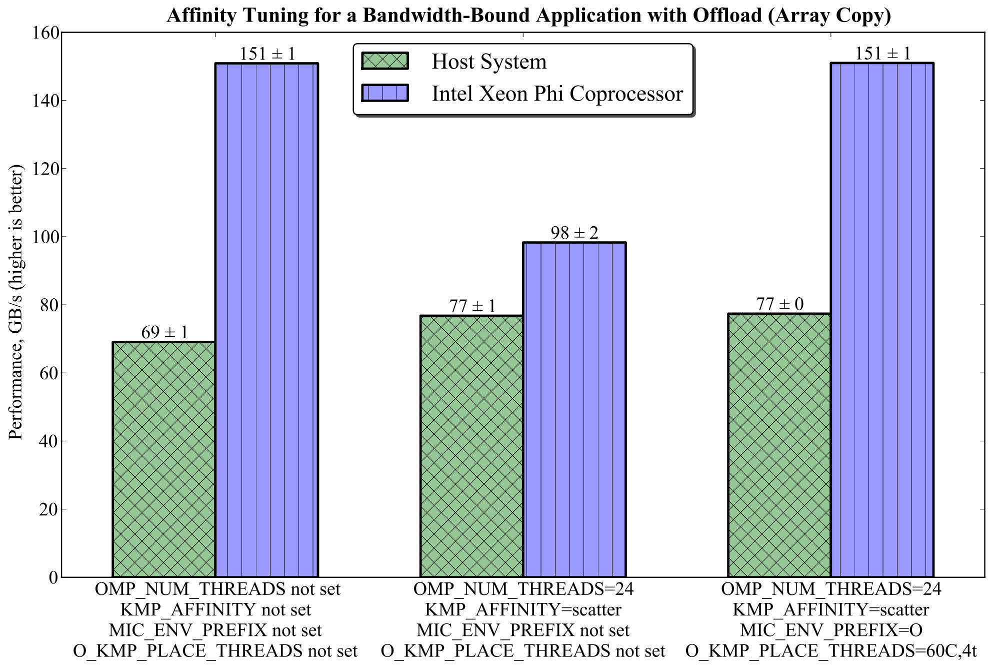


Figure 4.19: Performance of array copy with and without affinity settings. This example illustrates how to tune affinity settings in offload applications and for bandwidth-bound workloads.

We know how to resolve environment variable name collision for offload application from Section 2.2.9. In the third attempt, we added two environment variables: `MIC_ENV_PREFIX=O` (the letter "O" is for offload) and `O_KMP_PLACE_THREADS=60C,4t`. Note that on a 61-core processor, our offload application is restricted to 60 cores because core 0 is reserved for data movement. With these settings, performance on the host remained at the optimized value of 77 GB/s, and on the coprocessor it went back to the high value 151 ± 1 GB/s.

Example: Partitioning System Between Multiple Processes

Consider situations when multiple independent computing processes operate on a processor or a coprocessor. Such situations may occur:

a) in batch processing tasks. The user may set up a queue of independent jobs, and a scheduler application assigns jobs to free "slots" in a cluster or a batch farm. Several "slots" per compute node or per coprocessor may be used;
b) when one Intel Xeon Phi coprocessor is shared between multiple host processes, and
c) in native MPI applications with multiple ranks placed on one coprocessor.

We will not discuss the last case, because Intel MPI has its own way of process pinning (see Section 4.7.2). However, for the first two cases, there are situations where partitioning the processor or coprocessor between multiple processes yields better performance than running one process. These situations are:

a) the application has poor thread scalability, i.e., one T-threaded process delivers less performance than the cumulative performance of N processes with T/N threads each.
b) the application is running in a NUMA system based on a two- or four-way Intel Xeon processor; in this case binding one process to each CPU socket guarantees that it will operate on its NUMA-local memory.

As an example of such a case, we benchmarked a Discrete Fast Fourier Transform (DFFT) of a large one-dimensional array using the FFT functionality of Intel MKL. The array size is 2^{26} in double precision. We ran the benchmark on our two-way host system and on an Intel Xeon Phi coprocessor. Code for this example is available in Lab 4.07 – see Section 6.2. Results are shown in Figure 4.20.

We began by running one process that occupies all 48 threads on the host, and, similarly, one process that uses all 240 threads on the coprocessor. The variable `KMP_AFFINITY=none` was not set, so threads were allowed to migrate between cores. In this test, the host delivered 31.2 GFLOP/s, and the coprocessor 5.7 GFLOP/s.

After that, in an effort to improve the *cumulative* performance, we ran two 24-threaded processes on the host and two 120-threaded processes on the coprocessor. Each of them delivered less than one process using the whole system, but the *sum* of their performances was 41.5 GFLOP/s on the host and 11.2 GFLOP/s on the coprocessor.

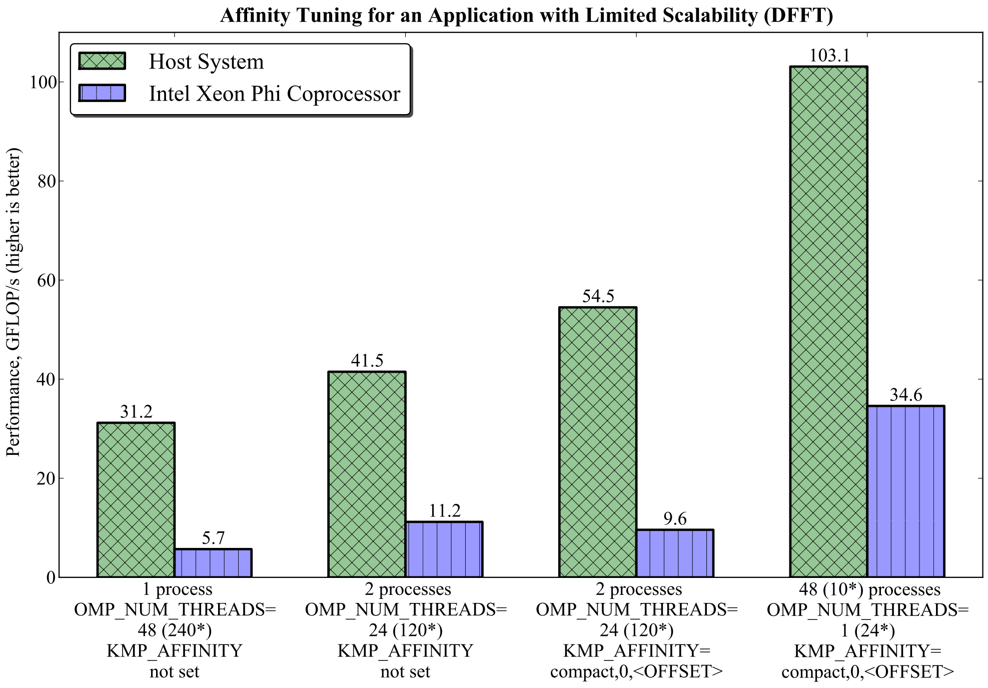


Figure 4.20: Performance of DFFT with and without affinity settings. This example illustrates how affinity settings help to partition a platform between multiple independent processes.
* Values with an asterisk apply to configuration on the Intel Xeon Phi coprocessor.

Clearly, as the second test shows, using fewer threads per process yields better performance. However, we expect that, especially on our two-way host, binding each process to the respective CPU socket should perform even better, because only NUMA-local memory will be used. This is why in the next test, we set thread affinity for each process. Each process had a different affinity setting:

- On the host, both processes used `OMP_NUM_THREADS=24`, but the first

  process had affinity set to `KMP_AFFINITY=compact,0,0` and the second to `KMP_AFFINITY=compact,0,24`.
- On the coprocessor, both processes used `OMP_NUM_THREADS=120`, but the first process had affinity set to `KMP_AFFINITY=compact,0,0` and the second to `KMP_AFFINITY=compact,0,120`.

The affinity settings shown above use used the parameter `offset` of variable `KMP_AFFINITY`. The first process populated 24 (or 120 threads) starting from offset 0, and the second populated the same number of threads starting from offset 24 (120). The result of this setting was that, as expected, the *net* performance of the two processes on the host increased to 54.5 GFLOP/s, but changed little on the coprocessor.

An offset of 0 maps to different OS procs on the host and on the coprocessor. On the host, it maps to OS proc 0, which corresponds to socket 0, core 0, thread 0. On the coprocessor, it maps to OS proc 1, which corresponds to core 1, thread 0. Subsequent OS procs map to cores and treads in a different way on the host and on the coprocessor. However, the bottom-line effect of the above affinity settings is the optimal partitioning of the system. This non-uniform mapping is done in the OpenMP library to relieve the programmer from dealing with the complexity of the architecture (specifically, with the fact that on a 61-core coprocessor, OS proc 0, 241, 242 and 243 correspond to core 0, threads 0-3, which are reserved for offload tasks).

Finally, in order to take full advantage of partitioning the system, we ran the last test with as few threads per process as possible. On the host, we were able to run 48 single-threaded DFFT processes, and on the coprocessor, memory limitation allowed us to run only 10 processes with 24 threads each. Affinity was set using the `offset` parameter. For example, on the host, single-threaded processes {0; 1; 2; etc.}, had affinity set to { `compact,0,0`; `compact,0,1`; `compact,0,2`; etc.}; on the coprocessor, the respective 24-threaded processes had `compact,0,0`, `compact,0,24`, `compact,0,48`, etc.

With that last effort, we nearly doubled the performance on the host to 103.1 GFLOP/s and tripled on the coprocessor to 34.6. GFLOP/s.

Affinity optimizations for the coprocessor discussed here are applicable to situations where multiple offload processes share cores on a single coprocessor. In this case, `MIC_ENV_PREFIX` may need to be used.

4.4.6. Diagnosing Parallel Efficiency, Scalability Tests

In the process of porting and optimizing applications on Intel Xeon processors and Intel Xeon Phi coprocessors, the programmer must ensure good parallel scalability of the application. On a multi-core CPU host, applications must efficiently scale to tens of threads in order to harness the task parallelism of Intel Xeon processors. On the Intel MIC architecture, the application must have a good scaling up hundreds of threads. Excessive synchronization, insufficient exposed parallelism and false sharing may limit the parallel scalability and prevent performance gains on the Intel Xeon Phi architecture.

A simple scalability test may help to assess the need for shared-memory algorithm optimization. The methodology of this test may be designed as below:

Step 1: Set thread affinity to `KMP_AFFINITY=scatter` for bandwidth-bound or `KMP_AFFINITY=compact,1` for compute-bound applications (drop the `,1` on the host if hyper-threading is disabled). Benchmark the code with one thread. On the host, set one thread with `OMP_NUM_THREADS=1` and on the coprocessor, use the syntax `KMP_PLACE_THREADS=1C,1t`.

Step 2: Proceed benchmarking the code with more threads, keeping only one thread per core (i.e., on the host, increase `OMP_NUM_THREADS` to the number of physical cores on the CPU; on the coprocessor, use `2C,1t`, `3C,1t`, ..., `60C,1t`).

Step 3: After that, start increasing the number of threads per core (on the host, test twice the number of physical cores if hyper-threading is enabled; on the coprocessor, try `60C,2t`, `60C,3t` and `60C,4t`).

The performance of a well-optimized compute-bound application must scale linearly in Step 2, and also for going from `60C,1t` to `60C,2t`. Further performance increase in Step 3 for such an application should be marginal. If scalability is not linear in Step 2, and if performance drops in Step 3, the application may be bandwidth-bound. If significant performance increase is observed in Step 3, the application likely has a significant latency-bound component.

To illustrate this test with an example, we performed this procedure for the compute-bound applications discussed in Section 4.3.2 and for the bandwidth-bound application discussed in Section 4.4.4. Raw performance results are displayed in Figure 4.21 and the parallel efficiency (ratio of raw performance to linear extrapolation of single-threaded performance) in Figure 4.22.

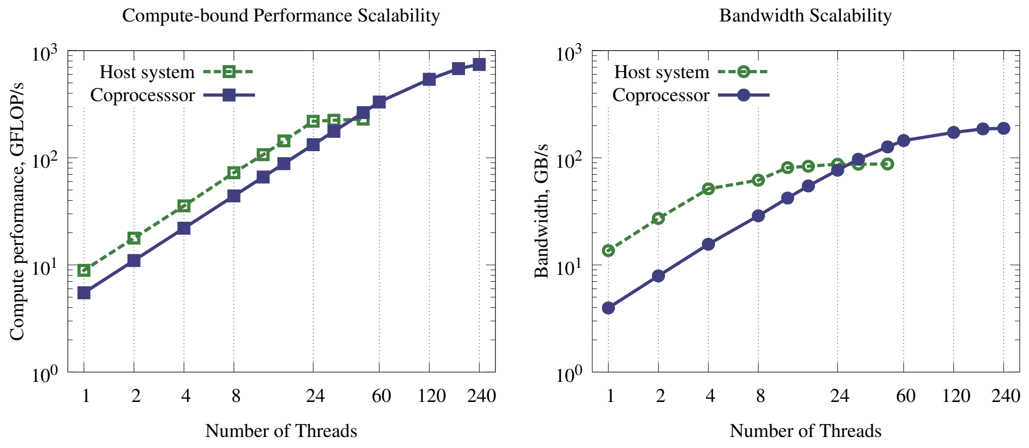


Figure 4.21: Scalability tests: raw performance results.

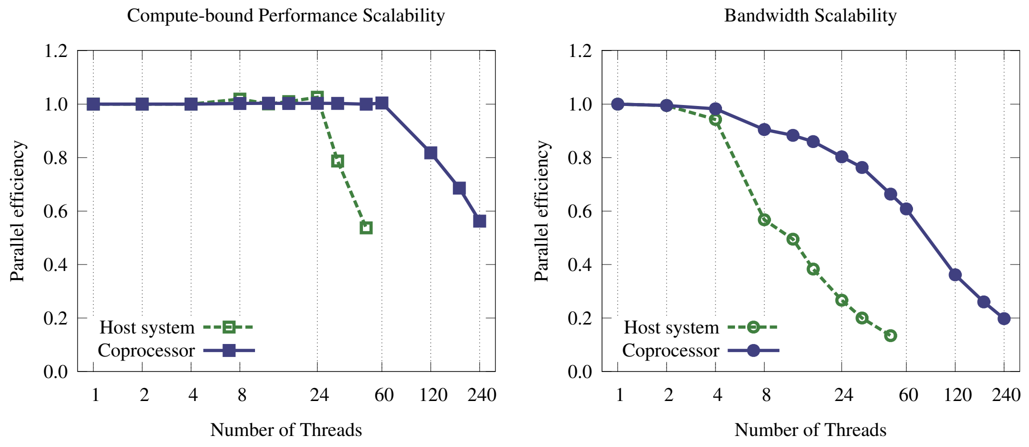


Figure 4.22: Scalability tests: parallel efficiency.

4.5. Memory Access Optimization

4.5.1. General Considerations

Computing applications can be classified by the efficiency of their memory access as:

i) **Compute-bound**. That means that the application saturates the throughput of ALUs or VPUs, so that memory traffic takes an insignificant fraction of CPU cycles.

ii) **Bandwidth-bound**. Such applications are limited not by the arithmetic performance, but by the rate at which memory controllers can deliver data from the main memory to processor cores.

iii) **Latency-limited** workloads are also limited by memory traffic, but by the latency of memory access rather than the rate of data transfer (bandwidth).

Latency-limited code segments may not perform well on the Intel MIC architecture. Generally, they would need to be changed to take advantage of caches in a better fashion (for example, using data transformations or data relayout techniques) as well as to employ efficient latency-hiding techniques (such as prefetching). Such algorithmic changes should be applicable to general-purpose CPUs as well, but usually have a bigger performance effect on the many-core architecture.

Both compute-bound and bandwidth-bound workloads can run more efficiently on the many-core platform than on multi-core general-purpose processors. In some cases, the programmer may optimize a bandwidth-bound application by changing the order of compute and memory access operations, so the application gets closer to being compute-bound and works faster both on multi-core and manycore platforms (again, this is not always possible).

Usually, for applications with data size N and arithmetic complexity $O(N)$, the best case scenario is working in the bandwidth-bound regime. However, for stronger complexity scaling (e.g., $O(N \log N)$ or $O(N^\alpha)$ for $\alpha > 1$), there is a single parameter of the algorithm which determines whether the application is compute- or bandwidth-bound. This parameter is arithmetic intensity, the number of arithmetic operations performed on each data element fetched from memory while it is still in the processor's registers or cache. Better understanding of the impact of this parameter is provided by an analytical tool called the roofline model discussed below.

Arithmetic Intensity

The estimate of the theoretical peak performance in double precision (64-bit floating-point numbers) for a 60-core Intel Xeon Phi coprocessor clocked at 1.0 GHz and utilizing 512-bit vector registers is

$$\text{Arithm. Performance } = 60 \times 1.0 \times (512/64) \times 2 = 960 \text{ GFLOP/s.} \quad (4.8)$$

Here, the factor 2 assumes that the fused multiply-add operation is employed, performing two floating-point operations per cycle. At the same time, the peak memory bandwidth of this system performing 6.0 GT/s using 8 memory controllers with 2 channels in each, working with 4 bytes per channel, is

$$\text{Memory Bandwidth } = 6.0 \times 8 \times 2 \times 4 = 384 \text{ GB/s.} \quad (4.9)$$

This amounts to $384/8 = 48$ billion floating-point numbers per second (in double precision). Therefore, in order to sustain optimal load on the arithmetic units of an Intel Xeon Phi coprocessor, the code must be tuned to perform no less than $960/48 = 20$ floating-point operations on every number fetched from the main memory. Arithmetic intensity greater than 20 makes the code compute-bound; much less than 20 makes it bandwidth-bound.

In comparison, a system based on two twelve-core Intel Xeon E5 processors clocked at 2.7 GHz (see Table 1.2) delivers up to

$$\text{Arithm. Perf. } = 2 \ \times 12 \times 2.7 \times (256/64) \times 2 = 518 \text{ GFLOP/s} \quad (4.10)$$

with a memory bandwidth

$$\text{Memory Bandwidth } = 2 \ \times 59.7 = 119 \text{ GB/s,} \quad (4.11)$$

where the additional factor of 2 in the estimate of performance reflects the presence of two ALUs (Arithmetic Logic Units) in each Ivy Bridge processor. Even though this processor does not have an FMA instruction, xAXPY-like algorithms may favorably utilize the processor's pipeline and employ both ALUs. The threshold arithmetic intensity for transitioning from bandwidth-bound to compute-bound workload is $518/(119/8) \approx 35$.

Roofline Model

Generally, the more arithmetic operations per memory access a code performs, the easier it is to fully utilize the arithmetic capabilities of the processor. That is, high arithmetic intensity applications tend to be compute-bound. In contrast, low arithmetic intensity applications are bandwidth-bound, if they access memory in a streaming manner, or latency-bound if their memory access pattern is irregular.

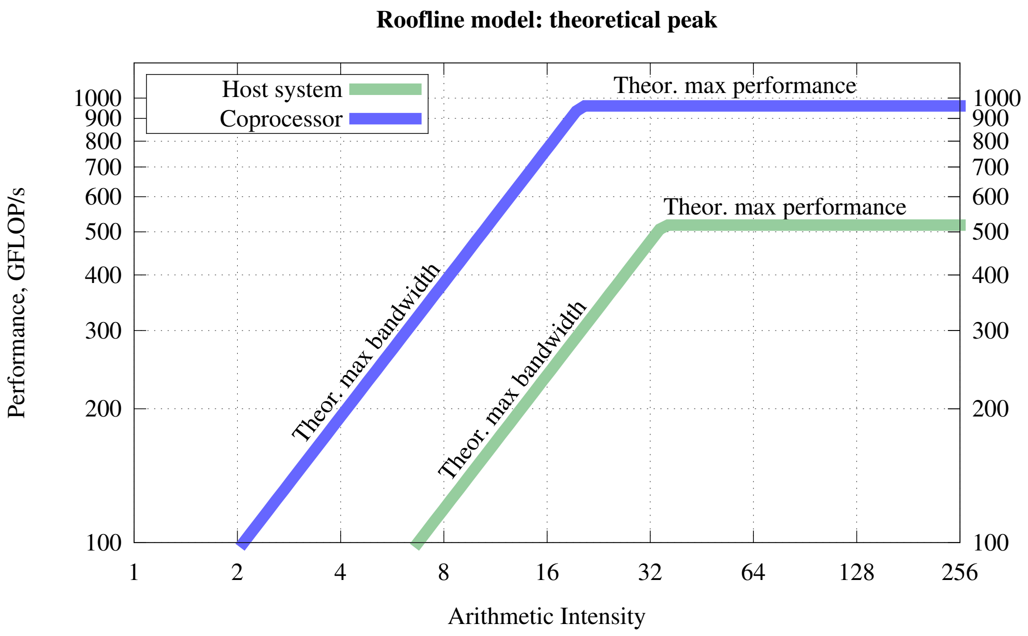


Figure 4.23: Basic roofline model for a host with two Intel Xeon E5 processors and for an Intel Xeon Phi coprocessor.

The relationship between the arithmetic intensity and the resource limitation of an application can be better understood with the help of the roofline model. Roofline model is a theoretical tool for assessing the optimization options of HPC applications. This model was suggested by Williams, Waterman & Patterson [36]. To build the roofline model for a specific architecture, one plots two lines on a log-log graph where the arithmetic intensity is plotted along the horizontal axis, and performance (in GFLOP/s) on the vertical axis. The two lines correspond to the bandwidth (a line with a unit slope normalized to the expected bandwidth) and to the peak arithmetic performance (a horizontal line normalized to the expected peak performance). The point where these two lines intersect corresponds to the maximum bandwidth and maximum arithmetic performance. An example of the roofline model is shown in Figure 4.23.

The utility of the roofline model plot is in its predictive power in the selection of code optimization options. Any application can be thought of as a column in this plot positioned at the arithmetic intensity of the application and extending upwards until it hits the "roof" represented by the model. If the column hits the sloping part of the roof (the bandwidth line), then the application is bandwidth-bound. Such an application may be optimized by improving the memory access pattern to boost the bandwidth or by increasing the arithmetic intensity. If the column hits the horizontal part of the roof (the performance line), then the application is compute-bound. Therefore, it may be optimized by improving the arithmetic performance by vectorization, utilization of specialized arithmetic instructions, or other arithmetic-related methods.

The roofline model can be extended by adding ceilings to the model. Figure 4.24 demonstrates an extended roofline model for the host system with two Intel Xeon E5 processors and for a single Intel Xeon Phi coprocessor. In this figure, an additional model is produced by introducing a realistic memory bandwidth efficiency η=50%. Additionally, we introduced a ceiling "without FMA" for the coprocessor and "one ALU" for the host. One of these ceiling correspond to applications that do no employ the fused multiply-add operation on the coprocessor, or do not fill the host processor pipeline in a fashion that utilizes both arithmetic logic units (ALUs) of Ivy Bridge processors. This assumption reduces the maximum arithmetic performance by approximately a factor of 2. Another ceiling additionally assumes that the application is scalar, i.e., does not use vector instructions. In double precision, this reduces the theoretical peak performance on the host by a factor of 4 and on the coprocessor by a factor of 8 (see Section 1.3.2 for additional discussion on this subject).

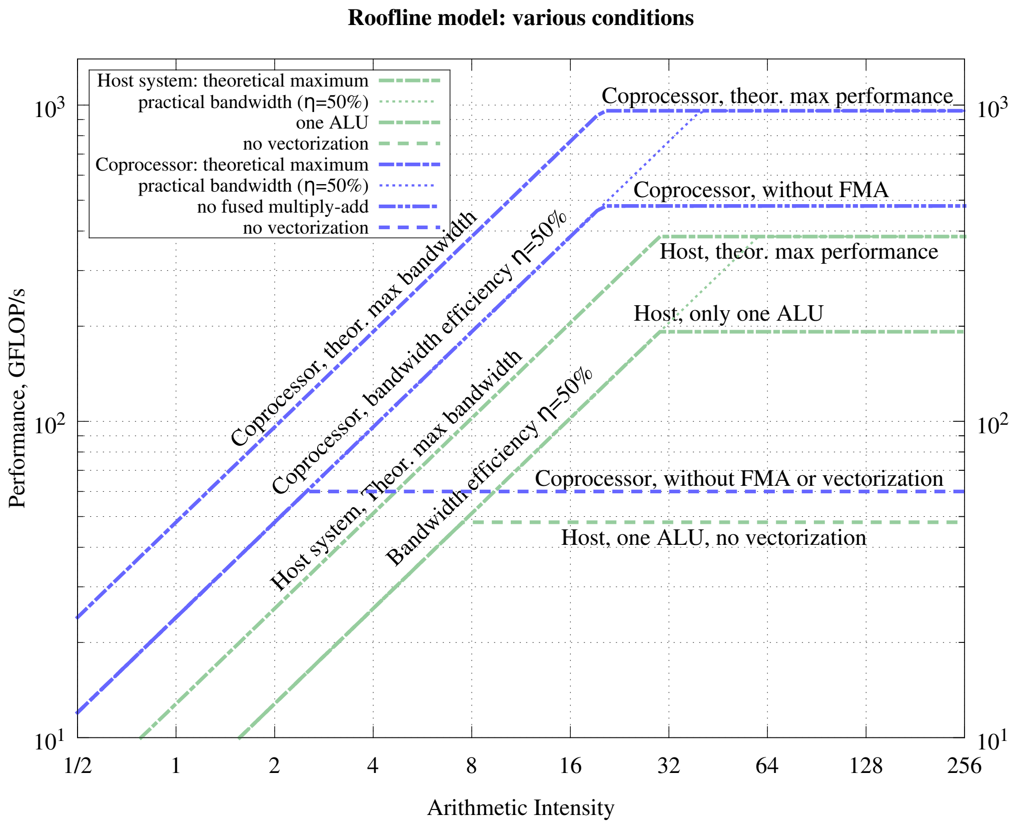


Figure 4.24: Extended roofline model for a host with two Intel Xeon E5 processors and for an Intel Xeon Phi coprocessor with a realistic bandwidth efficiency factor and additional ceilings.

The information in the roofline model plot can be used to preduct which optimizations are likely to benefit a given application. It also indicates the threshold arithmetic intensity at which the workload transitions from bandwidth-bound to compute-bound. The arithmetic intensity is a property of the numerical algorithm and can be varied for algorithms more expensive than $O(N)$. Code optimizations that improve the memory performance and increase the arithmetic intensity are presented in the current section.

Cache Misses

Intel processors and coprocessors have similar memory hierarchy: a large, but relatively slow RAM is cached with a smaller, but faster L2 cache, which, in turn, is cached by an even smaller and even faster L1 cache, residing in direct proximity of the core registers. See Figure 1.9 for KNC core topology and Table 1.3 for cache properties.

One aspect of cache organization distinguishes Intel Xeon processors from Intel Xeon Phi coprocessors. Intel Xeon processors have the L2 cache symmetrically shared between all cores, while in Intel Xeon Phi coprocessors, the L2 cache can be viewed as slices local to every core and connected via the CRI (Figure 1.8 illustrates the die layout).

Any algorithm that operates on data in RAM incurs cache misses when the data is loaded from RAM into all levels of cache hierarchy for the first time. Additionally, if the data set does not fit in the cache, the algorithm will incur additional cache misses as it processes the data, because the same data may be evicted from cache and fetched from RAM or lower-level cache multiple times. Every cache miss on a *read* operation makes the thread stall until the data requested by the thread is fetched from memory. A cache miss on a *write* does not necessarily stall the thread, because it may not need to wait until the data is written.

The latencies of communication with caches can be masked (i.e., overlapped with calculations). In Intel Xeon processors, hyper-threads and out-of-order execution are used on the hardware level to mask latencies. In Intel Xeon Phi coprocessors, 4 hardware threads per core play this role.

Sometimes it is possible to use special techniques to reduce the occurrence of cache misses in an algorithm:

1) permuting nested loops when it improves the locality of data access;
2) loop tiling (also known as loop blocking) for algorithms with nested loops acting on multi-dimensional arrays;
3) recursive cache-oblivious algorithms,
4) loop fusion and inter-procedural optimization.

These methods are described in Sections 4.5.2, 4.5.3 and 4.5.5. Additionally, for multi-socket Intel Xeon processors, first touch memory allocation is an important optimization (see Section 4.5.4).

4.5.2. Loop Tiling

Loop tiling is a technique for memory traffic optimization in algorithms that involve nested loops, multidimensional arrays, and regular access patterns. This technique is also known as “strip-mine and permute”, because to tile a loop, the programmer strip-mines (see Section 4.4.4) the inner or the outer loop and permutes some of the loops in the resulting code.

Cache Blocking

A loop tiling pattern known as “cache blocking” is shown in Listing 4.56. In this pattern, the inner (presumably, unit-stride) loop gets strip-mined, and in the resulting three nested loops, the outer two are permuted.

```
1 for (int i = 0; i < m; i++) // Plain nested loops
2   for (int j = 0; j < n; j++)
3      compute(a[i], b[j]); // Memory access is unit-stride in j
4
5 for (int jj = 0; jj < n; j+=TILE) // Tiled loops (cache blocking)
6   for (int i = 0; i < m; i++)
7     for (int j = jj; j < jj + TILE; j++) // Return to b[jj] sooner
8       compute(a[i], b[j]); // Memory access still unit-stride in j
```

Listing 4.56: Schematic organization of loop tiling: cache blocking.

To analyze this optimization, let us assume that the array `b` does not fit in cache. Then, in the non-optimized version (lines 1 through 3 in Listing 4.56), for every iteration in `i`, all the data of `b` will have to be read from memory into cache, evicted from cache and then fetched again in the next `i`-iteration. Re-organization of the loops with tiling (lines 5-8 in Listing 4.56) ensures that the code re-uses the value of `b[jj]` after only `TILE` iterations while it is still in a cache.

Ideally, the data spanned by the innermost loops in a cache-blocked algorithm should utilize the whole cache. This means that the size of the tile must be tuned to the specifics of computer architecture.

The loop in `i` can be tiled in a similar manner (see Listing 4.57), improving the locality of access to array `a[i]`.

```
for (int ii = 0; ii < m; ii += TILE) // Tiling in both i and j
  for (int jj = 0; jj < n; jj += TILE)
    for (int i = ii; i < ii + TILE; i++)
      for (int j = jj; j < jj + TILE; j++)
        compute(a[i], b[j]); // Memory access is unit-stride in j
```

Listing 4.57: Double loop tiling.

Special precautions should be taken with loop tiling. It is best to make array termination conditions as simple as possible, and use constants for tile sizes, in order to facilitate automatic vectorization. Specifically, when the values of m and n are not multiples of the tile size TILE, it is better to peel off some iterations as shown in the first part of Listing 4.58 than to compose a compact code with poorly known vector loop length (as shown in the commented out section ofn Listing 4.58).

```
// Efficient approach with redundant code for processing remainder:
// (m - m%TILE) is always a multiple of TILE
for (int ii = 0; ii < m - m%TILE; ii+=TILE)
  for (int j = 0; j < n; j++)
    for (int i = ii; i < ii + TILE; i++)
      compute(a[i], b[j]);

for (int i = m - m%TILE; i<m; i++) // Remaining iterations
  for (int j = 0; j < n; j++)
      compute(a[i], b[j]);

/* Inefficient approach below! Two problems:
   1) iMax is not a compile-time constant
   2) Loop count varies from iteration to iteration
for (int ii = 0; ii < m; ii+=TILE) {
  const int iMax = (ii+TILE > m ? m : ii+TILE);
  for (int j = 0; j < n; j++)
    for (int i = ii; i < iMax; i++)
      compute(a[i], b[j]);
} */
```

Listing 4.58: Peeling the tiled loop when m is not a multiple of TILE.

Register Blocking (Unroll and Jam)

Another pattern of loop tiling, known as register blocking or “unroll and jam”, is when the outer loop gets strip-mined, and in the resulting three nested loops, the inner two are permuted, however, vectorization is retained in the same loop as the in original code. Listing 4.60 shows this transformation. The outer loop is automatically vectorized using `#pragma simd`, which works best when the value of `TILE` a compile-time constant (see Listing 4.59) and is not large.

```
for (int ii = 0; ii < m; ii += TILE) // Tiling (unroll and jam):
#pragma simd
  for (int j = 0; j < n; j++) // Vectorization in j
    for (int i = ii; i < ii + TILE; i++)
      compute(a[i], b[j]);
```

Listing 4.59: Loop tiling for the purpose of register blocking (unroll and jam) with `#pragma simd` to vectorize the second innermost `j`-loop.

Unlike cache blocking, which targets data reuse in caches, the unroll-and-jam technique targets data reuse in processor registers.

With multiple nesting and tiling levels, the vector loop may be the *third* innermost. In this case, `#pragma simd` may fail, however, manually unrolling an inner loop re-enables automatic vectorization (see Listing 4.60).

```
for (int jj = 0; jj < n; jj += 4) // Explicit: j-tile is 4
  for (int ii = 0; ii < m; ii += 4) // Explicit: i-tile is 4
#pragma simd
    for (int k = 0; k < p; k++)
      for (int i = ii; i < ii + 4; i++) {
        compute(a[i], b[jj + 0], c[k]); // Unrolling in i in to
        compute(a[i], b[jj + 1], c[k]); // vectorize in k
        compute(a[i], b[jj + 2], c[k]);
        compute(a[i], b[jj + 3], c[k]);
    }
```

Listing 4.60: Double “unroll and jam” with manual unrolling to retain vectorization in `k`.

Tuning Loop Tiling

Tuning the tiled code requires trying different values for tile size. Furthermore, optimal values for tile size may be different for different loop dimensions. Additionally, optimal values may be different on the CPU-based host and on a MIC architecture coprocessor. Listing 4.61 demonstrates how to deal with these situations.

```
#ifdef __MIC__
  // Tuned values of tile sizes for the MIC architecture
  const int iTILE = 4;
  const int jTILE = 1024;
#else
  // Tuned values of tile sizes for the CPU architecture
  const int iTILE = 8;
  const int jTILE = 2048;
#endif

   // ...
```

Listing 4.61: Tiled loops may require tuning of the tile size for different loop dimensions and for different architectures.

Finally, tuning of tiled codes generally requires trying most of all the possible loop permutations. Generally, it is possible to reason why one order of loop nesting may be beneficial to another. Considerations for this reasoning include: unit-stride access, amount of data in containers traversed by different loop indices, sharing of variables between threads, and having sufficient parallelism in the outer loops. However, the interaction between these factors is often difficult to predict, while benchmarking up to $(n!)$ possible orders of n nested loops is usually a feasible task.

Examples of loop tiling in non-trivial situations may be found, for example, in [18]. A simpler educational example of memory traffic optimization with loop tiling is given in this section.

Example: Loop Tiling in Matrix-Vector Multiplication

Consider the problem of multiplying matrix `A` of size `m×n` by vector `b` of length `n`. For specificity, consider `m=1024` and `n=524288`. The resulting vector `c` of size `m` is calculated as

$$c_i = \sum_{j=0}^{m} A_{ij} b_j, \quad i = 0,\ 1,\ \ldots,\ (n-1). \tag{4.12}$$

A simple parallel code for this operation can be written as in Listing 4.62. See Lab 4.08 in Section 6.2 for complete code of this example.

```
void Multiply(const double* const A, const double* const b,
              double* const c, const long n, const long m){
  assert(n%8 == 0);
#pragma omp parallel for
  for (long i = 0; i < m; i++)
#pragma vector aligned
    for (long j = 0; j < n; j++)
      c[i] += A[i*n+j] * b[j];
}
```

Listing 4.62: Non-optimized code for matrix-vector multiplication.

This code uses OpenMP for thread parallelism in the outer loop and relies on automatic vectorization in the inner loop. We also assumed that the length of the rows of matrix `A` is a multiple of 64 bytes (this can be achieved by padding the rows if necessary), and that the matrix itself as well as vector `b` are allocated on 64-byte aligned boundaries (see Section 3.1.4). With these assumptions, the data alignment hint `#pragma vector aligned` (see Section 4.3.4) is used for additional performance. It is evident that the scalar, vector and thread aspects of this code's performance are optimized.

To establish the baseline performance, we ran this code with affinity setting `KMP_AFFINITY=scatter` on the host and on an Intel Xeon Phi coprocessor in the native mode. The performance was estimated in FLOP/s by dividing the number of operations, $2 \times m \times n$, by the calculation time. The result, shown in Figure 4.25, is 16.0 ± 0.1 GFLOP/s on the host and 30.7 ± 0.7 GFLOP/s on the coprocessor.

It is easy to see that the achieved performance is sub-optimal. In the course of the calculation, each element of `A` is used only once. Therefore, neglecting the traffic of vector `b`, the amount of time required for matrix-vector multiplication equals the amount of time required to read matrix `A`. Assuming memory bandwidth of around 80 GB/s on the host and 160 GB/s on the coprocessor, one can estimate that the optimal performance of the application is around 20 GFLOP/s on the host and 40 GFLOP/s on the coprocessor, which is 25-30% greater than we observed.

The problem is that the amount of time required for reading `b` is not negligible, even though this vector occupies far less memory than the matrix `A`. That is because each element of `b` is used `m=1024` times. With `n=524288`, vector `b` occupies 4 MiB, which exceeds the amount of L1 cache per core on Intel Xeon processors and Intel Xeon Phi coprocessors. It also exceeds the size of the L2 cache per core on coprocessors. Because `b` does not fit in the cache, it will have to be fetched from the main memory multiple times.

Loop tiling can help to avoid fetching `b` multiple times by re-using its elements several times (in other words, by increasing the arithmetic intensity of the calculation). We will perform loop tiling next.

Applying Tiling

Listing 4.63 shows the first version of optimized matrix-vector multiplication code with loop tiling. To arrive at this code, we had to experiment to decide whether to tile in `i` or in `j`, in which order to nest the loops, and what tile size to use.

```
void Multiply(const double* const A, const double* const b,
              double* const c, const long n, const long m){
  const long jTile = 4096L;
  assert(n%jTile == 0);
#pragma omp parallel
  {
    double temp_c[m] __attribute__((aligned(64)));
    temp_c[:] =0;

#pragma omp for
    for (long jj =0; jj < n; jj+=jTile)
      for (long i = 0; i < m; i++)
#pragma vector aligned
        for (long j =jj; j < jj+jTile; j++)
          temp_c[i] += A[i*n+j] * b[j];

    for(long i = 0; i < m; i++) {
#pragma omp atomic
      c[i]+= temp_c[i];
    }
  }
}
```

Listing 4.63: Matrix-vector multiplication code with loop tiling.

The performance of this code is 19.2 ± 0.1 GFLOP/s on the host and 32.5 ± 0.2 GFLOP/s on the coprocessor, which is 20% and 6% better than before tiling, respectively. Even though these values are better than before optimization, we are still short of the expected performance of around 40 GFLOP/s on the coprocessor. Therefore, we will continue optimization in the next step.

Tiling + Strip-Mining

One may notice that after tiling, the outer loop is the loop in `jj`. For our parameters, it has `n/jTile`=128 iterations. This should be alarming, because it is not enough to keep the 244 hardware threads on the coprocessor busy. We have already studied a similar case in Section 4.4.4 and found that strip-mining and loop collapse can help performance by expanding the parallel iteration space. This naturally leads us to the next optimization step, where we will strip-mine the loop in `i`. After that, the outer loops in `ii` and `jj` may be collapsed to expand the parallel iteration space as shown in Listing 4.64. This increases the performance on the host by 6% to 20.3 ± 0.2 GFLOP/s and on the coprocessor by 10% to 35.6 ± 0.1 GFLOP/s.

```
void Multiply(const double* const A, const double* const b,
              double* const c, const long n, const long m){
  const long iTile = 64L;    assert(m%iTile == 0);
  const long jTile = 4096L; assert(n%jTile == 0);
#pragma omp parallel
  {
    double temp_c[m] __attribute__((aligned(64)));
    temp_c[:] =0;
#pragma omp for collapse(2)
    for (long jj =0; jj < n; jj+=jTile)
      for (long ii = 0; ii < m; ii+=iTile)
        for (long i = ii; i < ii+iTile; i++)
#pragma vector aligned
          for (long j =jj; j < jj+jTile; j++)
            temp_c[i] += A[i*n+j] * b[j];
    for(long i = 0; i < m; i++) {
#pragma omp atomic
      c[i]+= temp_c[i];
    }
  }
}
```

Listing 4.64: Matrix-vector multiplication code with tiling, strip-mining and loop collapse.

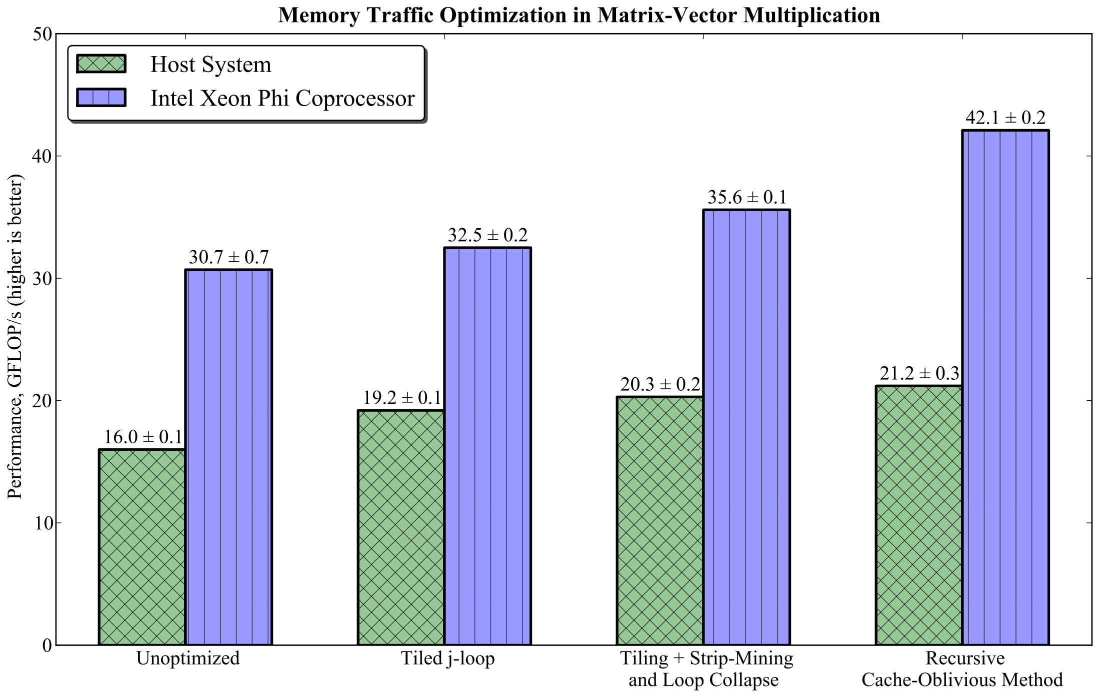


Figure 4.25: Performance of matrix-vector multiplication with loop tiling and cache-oblivious recursion applied for memory traffic optimization.

We have achieved speedup by a factor of 1.2 on the host and on the coprocessor using loop tiling to improve memory traffic. Doing so increased the arithmetic intensity of the usage of vector `b` and, thus, shifted the application to the right in the roofline model (see Section 4.5.1). In practice, depending on the problem, the impact of memory traffic optimization may be far greater, especially in applications for Intel Xeon Phi coprocessors.

Even though we achieved success here, we will proceed to learn in the next section an alternative to loop tiling known as cache-oblivious methods. As a motivation for this new method, find the last set of bars in Figure 4.25 titled "Recursive Cache-Oblivious Method". That last result takes the performance up one more notch, finally meeting our expected performance goal based on bandwidth estimates. The method used for this case is discussed in Section 4.5.3.

4.5.3. Cache-Oblivious Recursive Methods

An alternative approach to memory traffic optimization, known as cache-oblivious algorithms, may yield better performance than tiled algorithms, at the same time not requiring strict tuning to the cache size.

Principles

Cache-oblivious algorithms introduced by Prokop [37] and subsequently elaborated by Frigo et al. [38] recursively divide the data set into smaller and smaller chunks. Regardless of the cache size of the system, recursion will eventually reach a small enough data subset that fits into the cache. Listing 4.65 illustrates this approach.

```
// Non-Optimized algorithm
void CalculationNonOptimized(void* data, const int size)  {
  for (int i = 0; i < n; i++) {
   // ... perform work;
  }
}

// Optimized recursive cache-oblivious algorithm
void CalculationOptimized(void* data, const int size)  {
  if (size < recursionThreshold) {
    for (int i = 0; i < size; i++) {
      // ... perform work sequentially
    }
  } else {
    // Recursively split the data set
    CalculationRecurse(&data[0],      size/2);
    CalculationRecurse(&data[size/2], size/2);
  }
}
```

Listing 4.65: Schematic recursive cache-oblivious algorithm.

In practice, continuing recursion until the problem size is 1 operations is not optimal, as the overhead of function calls may outweigh the benefit of cache-efficient data handling. In addition, having just 1 operation per function precludes vectorization. For this reason, a threshold is introduced at which the recursion stops, and a sequential algorithm is applied.

For truly parallel problems, the recursive algorithm shown in Listing 4.65 is straightforward to parallelize using the fork-join model of parallelism. Listing 4.66 illustrates how this may be done in OpenMP.

```
// Optimized recursive cache-oblivious algorithm
void CalculationOptimized(void* data, const int size)  {
  if (size < recursionThreshold) {
    // ...
  } else {
    // Recursively split the data set and use
    // OpenMP tasks to parallelize the recursion
#pragma omp task
    {
      CalculationRecurse(&data[0],      size/2);
    }
    CalculationRecurse  (&data[size/2], size/2);
#pragma omp taskwait
  }
}

// The function must be called from a single thread
// in a parallel region as shown below.
#pragma omp parallel
{
#pragma omp single
  {
    CalculationOptimized(myData, originalSize);
  }
}
```

Listing 4.66: Schematic recursive cache-oblivious algorithm.

Implementation of parallel recursion in Intel Cilk Plus is even simpler, with `_Cilk_spawn` and `_Cilk_sync` used instead of `#pragma omp task` and `#pragma omp taskwait`, respectively.

Naturally, if the calculation must return results, the programmer must take care of avoiding race conditions. This may be done using thread-private containers, similarly to the case shown in Section 4.4.1.

Cache-Oblivious Matrix-Vector Multiplication

Listing 4.67 demonstrates a recursive implementation of matrix-vector multiplication which we started to optimize in Section 4.5.2. See Lab 4.08 in Section 6.2 for complete code of this example.

```
void recursMultiply(const double* const A, const double* const b,
      double* const c, const long n, const long m, const long lda){
  const long jThreshold = 8192L; assert(n%jThreshold == 0);
  const long iThreshold = 64L;   assert(m%iThreshold == 0);
  if ((m<=iThreshold) && (n<=jThreshold)) { // Recursion threshold
    for (long i = 0; i < m; i++)
#pragma vector aligned
      for (long j = 0; j < n; j++)
        c[i] = A[i*lda+j] * b[j]; // Matrix-vector multiplication
  } else { // Recursive divide-and-conquer
    if (m*jThreshold > n*iThreshold) { // Split i-wise
      double c1[m/2] __attribute__((aligned(64)));
#pragma omp task
      { recursMultiply(&A[0*lda + 0], &b[0], c1, n, m/2, lda); }
      double c2[m/2] __attribute__((aligned(64)));
      recursMultiply(&A[(m/2)*lda + 0], &b[m/2], c2, n, m/2, lda);
#pragma omp taskwait
      c[0:m/2] += c1[0:m/2]; c[m/2:m/2] += c2[0:m/2]; // Reduction
    } else { // Split j-wise
      double c1[m] __attribute__((aligned(64)));
#pragma omp task
      { recursMultiply(&A[0*lda + 0], &b[0], c1, n/2, m, lda); }
      double c2[m] __attribute__((aligned(64)));
      recursMultiply(&A[0*lda + n/2], &b[0], c2, n/2, m, lda);
#pragma omp taskwait
      c[0:m] += c1[0:m]; c[0:m] += c2[0:m]; // Reduction
} } }

void Multiply(const double* const A, const double* const b,
              double* const c, const long n, const long m){
#pragma omp parallel
#pragma omp single
    { recursMultiply(A, b, c, n, m, n); }  }
```

Listing 4.67: Matrix-vector multiplication with cache-oblivious parallel recursion.

Performance result with the cache-oblivious algorithm is shown in Figure 4.25. The application achieves 21.2 ± 0.3 GFLOP/s on the host and 42.1 ± 0.2 GFLOP/s on the coprocessor, respectively. This is 4% and 18% greater than the best case with the tiled algorithm. These results meet and even slightly exceed our theoretical performance expectation based on bandwidth estimates, which are 20 GFLOP/s and 40 GFLOP/s on the host and on the coprocessor, respectively.

Figure 4.26 explains why the recursive algorithm achieves high performance, and why data locality in this algorithm may be better than in the tiled implementation.

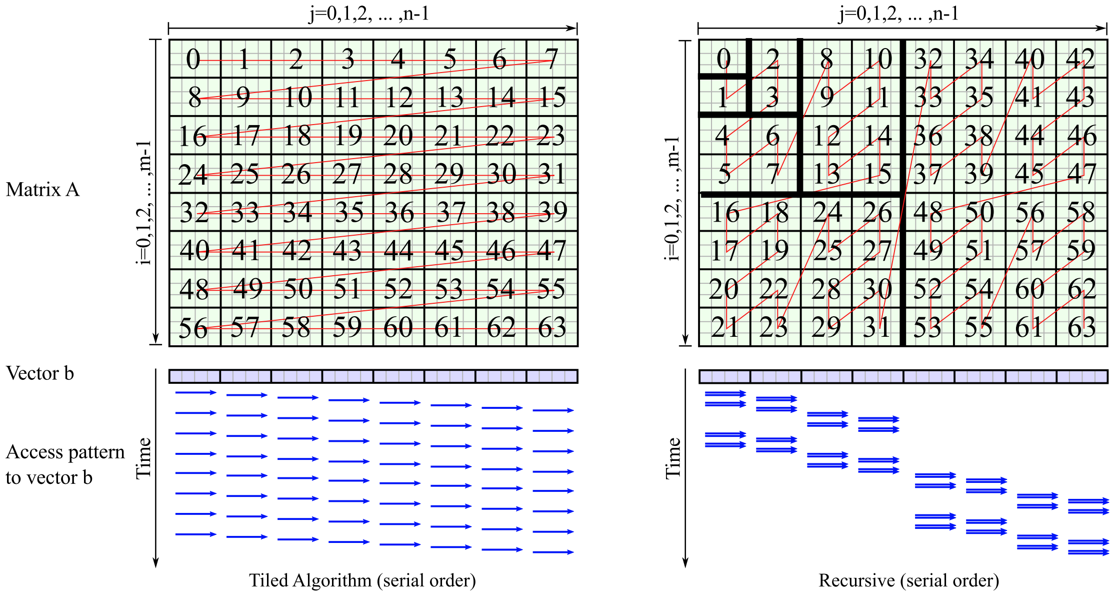


Figure 4.26: Order of matrix tile traversal in serialized tiled algorithm (Listing 4.64) and recursive algorithm (Listing 4.67).

The blue arrows indicate the pattern of memory access to the respective strips in vector `b` in a serial (i.e., single-threaded) implementation. Within each strip, the data of `b` is re-used a total of 64 times. At the next level of caching, locality of access is determined by how soon the algorithm re-visits any given strip. The tiled algorithm moves away from the original strip and re-visits it only after traversing the entire length of `b`. In contrast, the recursive algorithm stays in the vicinity of any given strip, re-visiting it immediately, then after 2 more strips, then after 10 strips. This behavior translates to the parallel algorithm, because strips are assigned to threads roughly in the same order as in the serial algorithm.

Generally speaking, translation of the memory access pattern of the sequential algorithm to that of the parallel algorithm is not trivial and depends on the scheduling mode. For instance, we have found an implementation of the tiled algorithm (see Section 4.5.2), which achieves a slightly higher performance on the host than the recursive algorithm (23 GFLOP/s on our system). However, the same implementation works slightly worse than the recursive method on the coprocessor (achieving 37 GFLOP/s). Thus, the recursive solution appears to be more portable. Readers wishing to experiment more in this area may want to know that this implementation of tiled algorithm requires tuning

i) the loop scheduling mode (`static,1` works best) and
ii) the first touch allocation pattern (the same exact pattern as in the usage of matrix `A` works best).

The technique of bandwidth optimization by choosing the first touch pattern is discussed in Section 4.5.4.

4.5.4. First Touch Allocation and NUMA Policy

NUMA, or Non-Uniform Memory Architecture, is a general term for shared-memory computing systems in which cores and memory are partitioned into nodes, and the latency and bandwidth of access between different nodes is different.

An example of a NUMA system is a multi-socket computer based on Intel Xeon processors (see Figure 4.27). In such a solution, access by a core to a memory bank is faster if that memory bank is controlled by the socket containing that core. This applies to latency and to bandwidth. Memory banks controlled by a socket are often referred to as "local" or "NUMA-local" memory with respect to that socket.

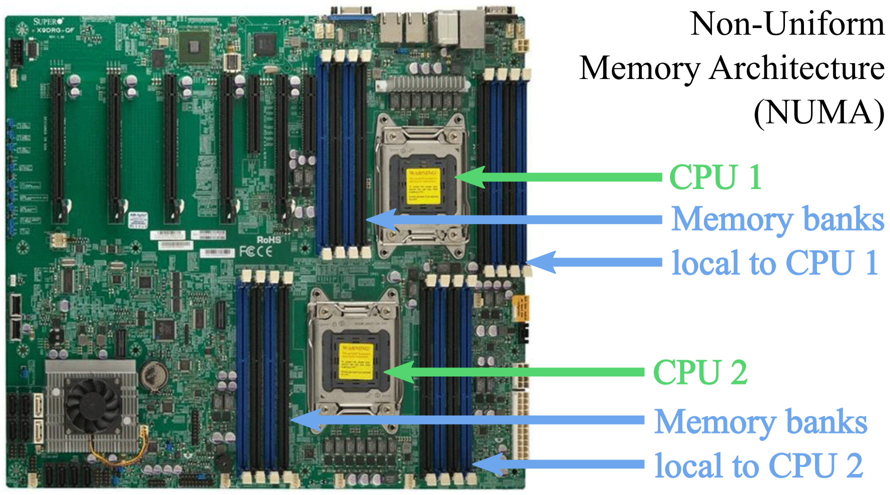


Figure 4.27: Example of a NUMA architecture: server board of our SXP8600 workstation based on a two-way (i.e., dual-socket) Intel Xeon processor.

Two memory behaviors in Linux may be important for optimization of applications in NUMA systems:

1) Memory pages for data are assigned when they are first touched by a thread, rather than during the call to `malloc`.
2) In NUMA systems, the default page allocation policy is local, i.e., pages are allocated close to the thread that touched them.

The consequence of these properties is that for parallel applications, the programmer must ensure favorable first touch pattern. That is, arrays must be first touched with the same parallel pattern as the pattern with which they will be used. For instance, arrays used in parallel regions must be initialized also from parallel regions (see, e.g., Listing 4.68).

```
float* matrix = (float*) _mm_malloc(N*ld*sizeof(float));

// Bad practice: first touch with a single thread
// for (int c = 0; c < N*ld; c++)
//   matrix[c] = 0.0f;

// Bad first touch: with the wrong parallel pattern
// #pragma omp parallel for schedule(dynamic,1)
// for (int c = 0; c < N*ld; c++)
//   matrix[c] = 0.0f;

// Good first touch: same parallel pattern as in usage
#pragma omp parallel for
for (int i = 0; i < N; i++)
  for(int j = 0; j < ld; j++)
    matrix[i*ld + j] = 0.0f;

// Scenario of usage of the data container "matrix"
#pragma omp parallel for
for (int i = 0; i < N; i++)
  for(int j = 0; j < ld; j++)
    b[i] += matrix[i*ld + j]*x[j];
```

Listing 4.68: Memory allocation on first touch: good and bad practices.

Sometimes, the pattern in which an array is used is so complex that just touching an array by from a parallel region is not sufficient to ensure a favorable memory allocation. In these cases, first touch by a “dry run” of the calculation may help to improve performance.

Additionally, some applications may use an array in more than one parallel calculation, and a memory allocation favorable for one phase of the application may be unfavorable for another. In these cases, a tradeoff must be sought, for example, performing first touch by the more computationally demanding phase. An example of such case may be found in [21].

Example:

To illustrate the performance impact of first touch, we will re-visit the matrix-vector multiplication code developed in Section 4.5.3. First touch in this problem is important only for matrix A, because our cache optimizations rendered cache traffic in b and c insignificant.

First, we benchmarked the same application as in Section 4.5.3, but with matrix A first touched with a serial loop instead of a parallel loop (see Listing 4.69). The resulting performance on the two-way CPU was reduced by almost a factor of 2, to 13.0 ± 0.4 GFLOP/s. However, on the coprocessor, performance was not affected, because it is not a NUMA system.

```
  double * A = (double*) _mm_malloc(sizeof(double)*n*m, 64);
  for (long i = 0; i < m; i++)
    for (long j = 0; j < n; j++)
      A[i*n+j] = i;
```

Listing 4.69: Memory allocation on first touch with a serial region.

After that, we restored the initialization of matrix A to what it was in Section 4.5.3, where the matrix is initialized with a parallel loop (Listing 4.70). This, indeed, recovered the performance observed in Section 4.5.3: 21.2 ± 0.2 GFLOP/s on the host and 42.0 ± 0.3 GFLOP/s on the coprocessor.

```
  double * A = (double*) _mm_malloc(sizeof(double)*n*m, 64);
#pragma omp parallel for
  for (long i = 0; i < m; i++)
    for (long j = 0; j < n; j++)
      A[i*n+j] = i;
```

Listing 4.70: Memory allocation on first touch with a parallel region.

Finally, after some experimentation, we established that a 5% performance increase is possible by optimizing the first touch pattern as shown in Listing 4.71. In this case, it rows of A are given one by one to threads 0, 1, 2, 3, etc. Because we used KMP_AFFINITY=scatter, it means that every even-numbered row is touched by CPU 1, and every odd-numbered row by CPU 2. Apparently, such placement of matrix `A` avoids a load skew observed in the previous case, and achieves 22.2 ± 0.3 GFLOP/s on the host and remains unchanged at 42.1 ± 0.3 on the coprocessor.

```
  double * A = (double*) _mm_malloc(sizeof(double)*n*m, 64);
#pragma omp parallel for schedule(static,1)
  for (long i = 0; i < m; i++)
    for (long j = 0; j < n; j++)
      A[i*n+j] = i;
```

Listing 4.71: Memory allocation on first touch with a parallel region and scheduling tweak.

Measurements discussed above are summarized in Figure 4.28.

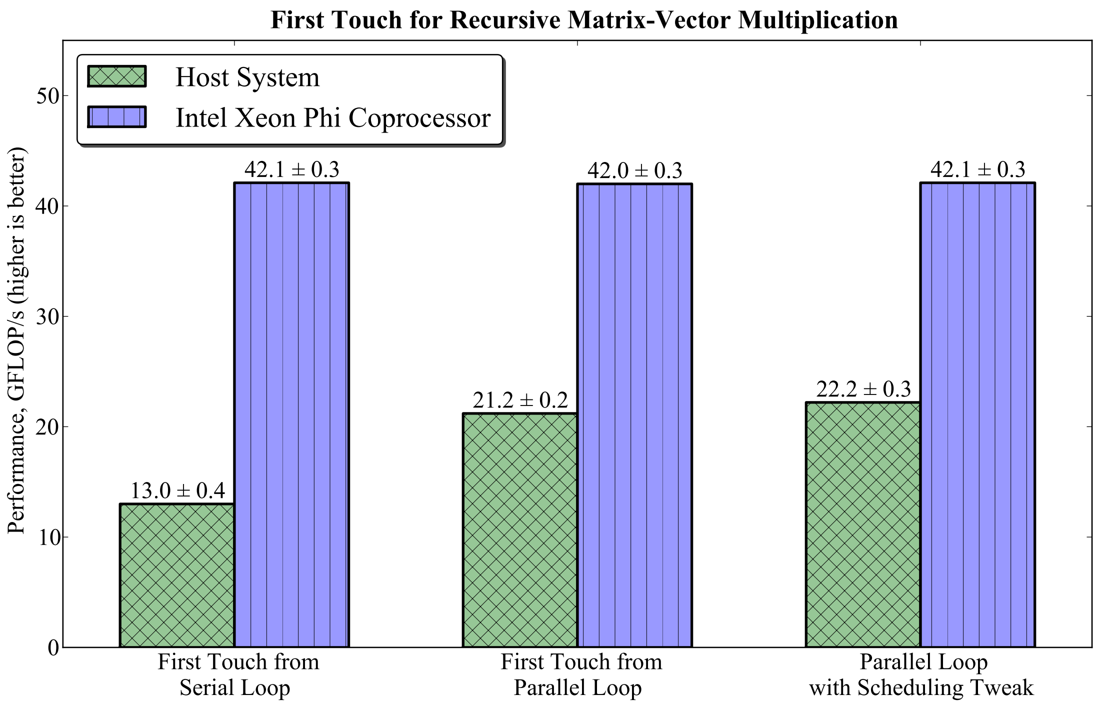


Figure 4.28: Recursive matrix-vector multiplication: different first touch patterns for `A`.

4.5.5. Cross-Procedural Loop Fusion

When different stages of data processing are executed in loops of similar structure, it may be beneficial to fuse such loops. Loop fusion is another loop optimization in which two disjoint loops that have the same iteration count are merged into a single loop. This optimization is safe only if the data dependence between the two loops is such that the fused loop retains the same semantics as the original two loops. Loop fusion is often beneficial for performance, because it increases data locality. Listing 4.72 illustrates loop fusion.

```
// Two distinct loops operating on the same data
for (int i = 0; i < n; i++)
  ProcessingStage1(inData[i], outData[i]);
for (int i = 0; i < n; i++)
  ProcessingStage2(outData[i]);

// The above code expressed as a fused loop
for (int i = 0; i < n; i++) {
  ProcessingStage1(inData[i], outData[i]);
  ProcessingStage2(outData[i]);
}
```

Listing 4.72: Loop fusion may reduce memory traffic by increasing temporal data locality.

Loop fusion is beneficial for cache performance, because in the case of two disjoint loops, by the time that the first loop is finished, the beginning of the data set may have been evicted from caches. However, in a fused loop, all stages of data processing occur while the data is still in the caches. In addition, loop fusion may help to reduce the memory footprint of temporary storage if such storage was needed in order to carry some data from one loop to another.

If two loops that are candidates for fusion are located within the same lexical scope, Intel compilers may fuse them automatically. Intel compilers also capable of some inter-procedural optimization. However, automatic loop fusion may fail if the compiler does not see both loops at compile time (e.g., the loops are located in separate source files), or if additional measures must be taken for value-safe fusion. In cases when automatic loop fusion fails, the programmer may need to implement it explicitly.

Example: Fusing Data Generation and Processing

Opportunities for loop fusion often occur in pipelined data processing. Listing 4.73 shows a synthetic example, in which some data is generated and then processed (full code for this exercise is availabe in Lab 4.09 – see Section 6.2). A total of $m = 10000$ data sets are generated, each containing $n = 40000$ numbers. Data generation in our archetypal example is done by producing random numbers with Intel MKL. The processing of data involves, for each data set, a single pass in which the mean and the standard deviation are computed for that data set. Results are returned as the set of m values of the mean and standard deviation for each set. This specific processing pipeline is not in itself a meaningful workload; rather, it represents a class of applications with pipelined processing of multiple data sets.

```
void GenerateData(const int m, const int n, float* const allData) {
#pragma omp parallel for
  for (int i = 0; i < m; i++) {
    float* data = &allData[i*n]; // Fill data set in i-th position
    // ...
  }
}

void AnalyzeData(const int m, const int n, const float* allData) {
#pragma omp parallel for
  for (int i = 0; i < m; i++) {
    float* data = &allData[i*n]; // Use data set in i-th position
    // ...
  }
}

void RunProcess(const int m, const int n) {
  // ... Setup omitted ...
  float* data = (float)_mm_malloc(sizeof(float)*n*m, 64);
  GenerateData(m, n, allData);
  AnalyzeData(m, n, allData);
}
```

Listing 4.73: Generation and processing of data in functions with disjoint parallel loops.

Benchmarking this code yields a baseline performance of 2.77 ± 0.02 billion values processed per second on the host, and 2.68 ± 0.04 billion values per second on the coprocessor (see Figure 4.29).

The absolute value of these performance measurements are difficult to relate to the theoretical peak performance of the hardware. This is because the actual workload involves generation of normally distributed random numbers with Intel MKL. However, opportunities for optimization are apparent from the structure of the code. Indeed, the entire data set is `n×m×sizeof(float)`=1.5 GiB in size. During the data generation step, the entire data set is accessed once, and during the analysis step, it is accessed a second time. Because 1.5 GiB is far greater than the size of the cache in the system, the second access will go to the main memory.

To optimize the application, we can fuse the parallel loops in functions `GenerateData()` and `AnalyzeData()` and move them out into `RunProcess()`, as shown in Listing 4.74.

```
void GenerateData(const int n, const int i, float* const data) {
  // ... Generate a single data set number i...
}

void AnalyzeData(const int n, const int i, const float* data) {
  // ... Analyze a single data set number i...
}

void RunProcess(const int m, const int n) {
  // ... Setup omitted ...
  float* data = (float)_mm_malloc(sizeof(float)*n*m, 64);
#pragma omp parallel for
  for (int i = 0; i < m; i++) {
    float* data = &allData[i*n]; //Process dataset in i-th position
    GenerateData(n, i, data);
    AnalyzeData(n, i, data);
  }
}
```

Listing 4.74: Fused parallel loops in one function.

This optimization alone improves the performance on the host by 20% and on the coprocessor by 6%. More importantly, it opens up an opportunity for another memory traffic optimization: shrinking down the memory footprint of the application.

Indeed, the large array `allData` in Listing 4.73 was only necessary to carry the data from function `GenerateData()` to `AnalyzeData()`. This is no longer needed in Listing 4.74 where the loops over the data sets are fused. Therefore, function `RunProcess()` may be re-written as in Listing 4.75. There, one array of size `n` is allocated in every thread instead of `m=10000` such arrays.

```
void RunProcess(const int m, const int n) {
  // ... Setup omitted ...
#pragma omp parallel for
  for (int i = 0; i < m; i++) {
    float data[n] __attribute__((aligned(64))); // Smaller array
    GenerateData(n, i, data);
    AnalyzeData(n, i, data);
  }
}
```

Listing 4.75: Loop fusion allows to shrink the application memory footprint by eliminating unnecessary scratch data containers.

After shrinking the memory footprint, the performance went up another 27% on the host and 57% on the coprocessor, achieving 4.22 and 4.45 billion values per second on the respective platforms. This is because the application data now fits completely in the level 2 cache both on the CPU and on the coprocessor. Therefore, the data generation step does not require memory accesses.

The effect of loop fusion and with reduced scratch memory footprint is shown in Figure 4.29. The complete working code for this example can be found among the Exercises (see Section 6.2).

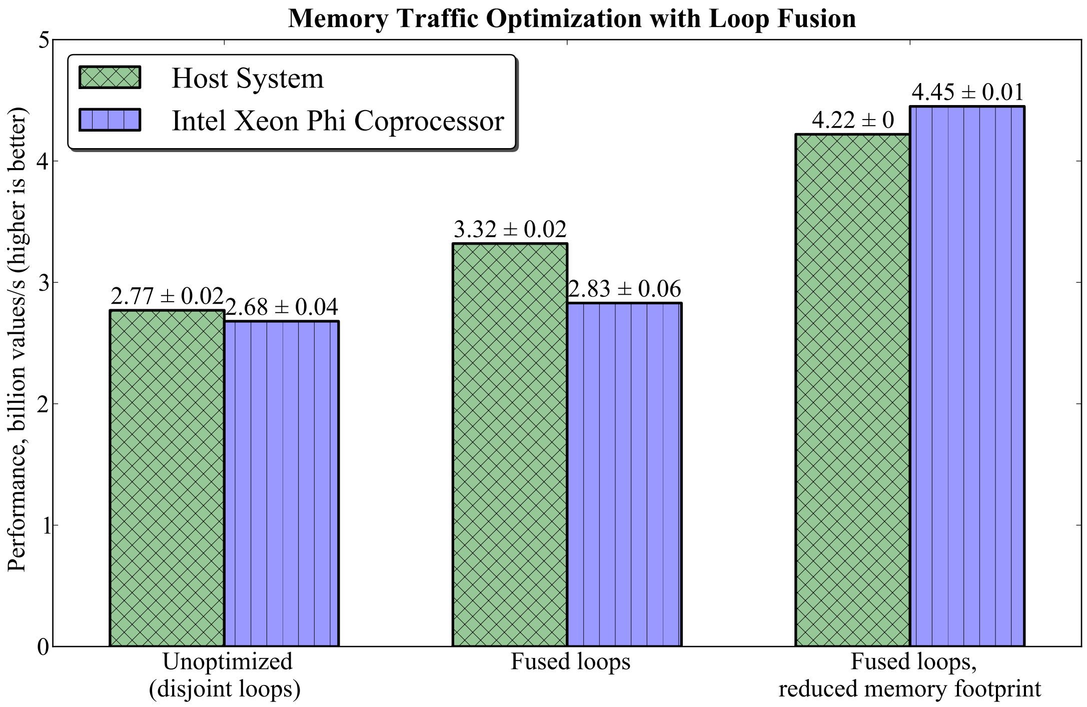


Figure 4.29: Performance of the code generating and analyzing pseudo-random data. The non-optimized case is shown in Listing 4.73, and the optimized case in Listing 4.74 and Listing 4.75.

4.5.6. Advanced Topic: Prefetching

Intel Xeon processors and Intel Xeon Phi coprocessors improve cache traffic on the hardware level with the help of hardware prefetchers. These devices monitor memory access pattern of running applications, train to it, and predictively issue requests to fetch data from memory into caches several cycles before this data is used by the core. Intel Xeon processors have L1 and L2 hardware prefetchers, and Intel Xeon Phi coprocessors have only an L2 hardware prefetcher.

In addition to hardware prefetching, Intel Xeon processors and Intel Xeon Phi coprocessors support software prefetch instructions. These instructions request that a certain address (cache line) is fetched from memory into a cache. Software prefetch instructions do not stall execution, and therefore they can be issued many cycles before the fetched line is used by the core. The time between the prefetch instruction and the instruction consuming the data on the core is called the prefetch distance. In loops, prefetch distances are typically measured in the number of iterations between the prefetch instruction and data consumption.

The Intel compilers automatically insert prefetch instructions into the compiled code for the MIC architecture at optimization level `-O2` and above. The prefetch distance and prefetched variables are computed using heuristics. These heuristics work for array accesses (e.g., `A[i][j]` or `B[i*n+j]`) and pointer accesses where the address can be predicted in advance (e.g, `*(A+(i*n+j)*8)`). However, by default, the compiler does not issue prefetch instructions for accesses in the form `A[B[i]]`. Such indirect prefetching can be forced by using a pragma of the form `#pragma prefetch A:0:2` before the loop. Here 0 means issue prefetch with `vprefetch0` (hint-0) and 2 means use a prefetch distance of 2 (possibly vectorized) iterations. It is possible to see the report on compiler prefetching using the compiler argument `-opt-report=3`.

To fine-tune the application performance, the programmer may wish to control software prefetching. If this approach is taken, it is advisable to first turn off automatic compiler prefetching using the compiler argument `-no-opt-prefetch` (to disable prefetching in the whole source file) or placing `#pragma noprefetch` before a function or a loop (for a more fine-grained control). After that, prefetching can be modified with the argument `-opt-prefetch-distance` (to affect the whole file) or effected with the intrinsic `_mm_prefetch()` or `#pragma prefetch` (for precise control of each variable).

The following general considerations may be helpful in the planning of prefetch optimization:

a) It is possible to diagnose whether the performance of a particular application can be improved with software prefetching. The most simple test is to turn off prefetching in the whole application or a particular loop or function. If the performance drops significantly, then prefetching plays an important role, and fine-tuning the prefetch distances can lead to performance increase.

b) Prefetching is more important for Intel Xeon Phi coprocessors than for Intel Xeon processors. This is in part explained by the fact that Intel Xeon cores are out-of-order processors, while Intel Xeon Phi cores are in-order. Out-of-order execution allows Intel Xeon processors to overlap computation with memory latency. Additionally, the lack of a hardware L1 prefetcher on Intel Xeon Phi coprocessors makes software prefetching necessary on the lowest level of cache traffic.

c) Loop tiling and recursive cache-oblivious algorithms (see Section 4.5.2 and Section 4.5.3) improve application performance by reducing cache traffic, and, therefore, prefetching becomes less important for algorithms optimized with these techniques.

d) If software prefetching maintains good cache traffic, hardware prefetching does not come into effect.

Additional information on prefetching on Intel Xeon Phi coprocessors can be found in this presentation by Rakesh Krishnaiyer [39]. The Intel C++ Compiler Reference Guide has detailed information about pragmas `prefetch` and `noprefetch` and compiler argument `-opt-prefetch`.

4.6. Offload Traffic Control

Applications that use the coprocessor in the offload mode can benefit from the optimization of the communication between the host and the coprocessor. In this section, we demonstrate the fundamental strategies for optimizing communication in offload applications: memory buffer retention and data persistence on the coprocessor between offloads.

4.6.1. Bandwidth Optimization with Persistent Buffers

In Section 2.2.4 we mentioned that by default, the offload library allocates memory buffers for offloaded arrays before offload and deallocates them after offload, which may result in performance loss. In this section we will verify this statement with benchmarks and also verify the that effecting memory buffer persistence is a viable solution.

Default Offload Mode

The code demonstrated in Listing 4.76 will be used for benchmarking the default offload mode.

```
// Default offload procedure:
// 1) allocate memory on coprocessor,
// 2) transfer data,
// 3) deallocate memory on coprocessor
for (int trial = 0; trial < nTrials; trial++) {
  const double t0 = omp_get_wtime();
#pragma offload_transfer target(mic:0) in(data: length(size))
  const double t1 = omp_get_wtime();
  printf("The offload latency=%.6f s, bandwidth=%.3f GB/s",
                                   (t1-t0), 1e-9*size/(t1-t0));
}
```

Listing 4.76: Transfer of data to the coprocessor in the default offload mode.

This function transfers array `data` to the coprocessor in the default mode. At the beginning of each offload, the offload library will allocate memory for the respective array on the coprocessor, then the data will be transferred over the PCIe bus, calculations will be performed, and memory will be deallocated.

Memory Buffer Retention Between Offloads

Consider the case when a function performing offload is called multiple times, and all or some of the pointer-based arrays sent to the coprocessor have the same size. Then offload can be optimized using the clauses `free_if` and `offload_if` in order to preserve the memory allocated for the array on the coprocessor (see also Section 2.2.4) as shown in Listing 4.77.

```
// Allocate and retain a buffer on the coprocessor
#pragma offload_transfer target(mic:0) \
                      in(data: length(size) alloc_if(1) free_if(0))

for (int trial = 0; trial < nTrials; trial++) {
  const double t0 = omp_get_wtime();
  // Re-use the buffer retained on the coprocessor:
#pragma offload_transfer target(mic:0) \
                      in(data: length(size) alloc_if(0) free_if(0))
  const double t1 = omp_get_wtime();
  printf("The offload latency=%.6f s, bandwidth=%.3f GB/s",
                                  (t1-t0), 1e-9*size/(t1-t0));
}

// Delete the buffer on the coprocessor
#pragma offload_transfer target(mic:0) \
                      in(data: length(size) alloc_if(0) free_if(1))
```

Listing 4.77: Optimized offload with memory buffer retention.

Benchmarks of the default and optimized offload traffic are shown in Figure 4.30 and Figure 4.31. For each mode of offload, we benchmarked the “in” and “out” data transfer direction.

The effect of memory buffer retention on the offload performance is very significant. For large arrays, memory buffer retention increases the bandwidth of data movement almost by a factor of 2.8, achieving 6.8 GB/s. For smaller arrays, the effect is even more dramatic because the latency of the memory allocation operation comes into play. The latency of small array offload into a retained memory buffer is around 10 μs, while without buffer retention it is as high as 2000 μs.

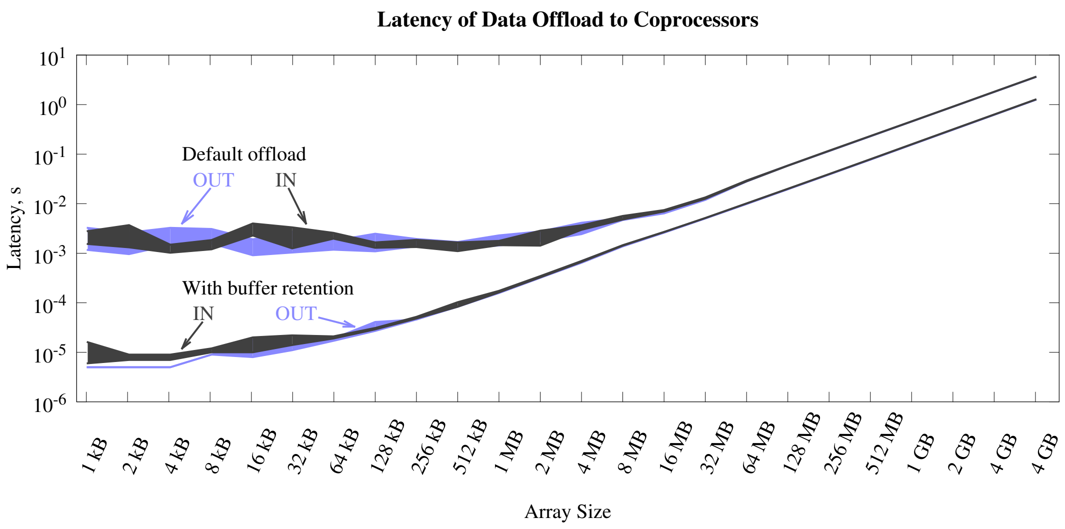


Figure 4.30: Latencies of data offload to the coprocessor. See Section 4.6.1 for details.

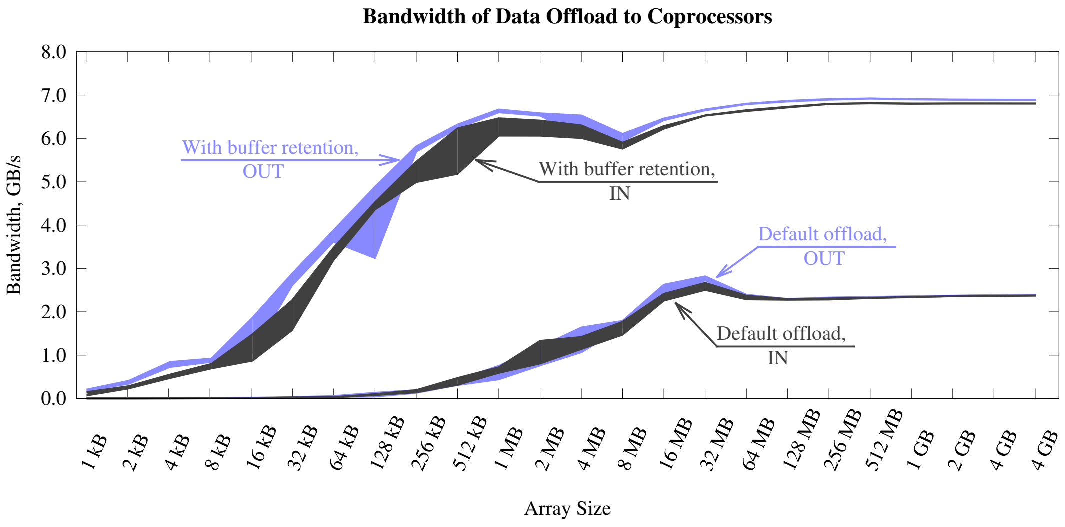


Figure 4.31: Offload bandwidth calculated by dividing the transferred data size by latency.

Example: Matrix-Matrix Multiplication with Offload

To illustrate memory buffer retention in practice, we will benchmark a synthetic application performing matrix-matrix multiplication using the Intel MKL implementation of DGEMM. We assume that multiple matrices are initialized on the host, then copied to the coprocessor for multiplication, and the result is returned back to the coprocessor. See Lab 4.10 for complete code (refer to Section 6.2).

The arithmetic complexity of DGEMM scales as $O(n^3)$ while its data size scales as $O(n^2)$. As we established in Section 1.3.4, in this case, the larger the size of the matrix, the less significant the offload time becomes. Therefore, to simulate a more challenging situation, we will multiply relatively small matrices with `n=1024`, and do it with `nMatrices=8` sets of matrices.

An initial implementation of this workload is shown in Listing 4.78.

```
  for(int i = 0; i < nMatrices; i++) {
    double* A = &A_arr[i*n*n];
    double* B = &B_arr[i*n*n];
    double* C = &C_arr[i*n*n];
#pragma offload target(mic:0) \
  in(A: length(n*n)) in(B: length(n*n)) out(C: length(n*n))
    {
      cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                  n, n, n, 1.0, A, n, B, n, 0.0, C, n);
    }
  }
```

Listing 4.78: DGEMM in the offload mode with default offload options.

Timing this code inside and outside the scope of the offload pragma indicates that the effective performance of matrix-matrix multiplication (i.e., taking into account data movement time) is 58 ± 4 GFLOP/s. If we neglect the data movement time, the performance is ≈ 115 GFLOP/s. The effective bandwidth of data movement (i.e., taking into account the memory buffer allocation time) is ≈ 1.5 GB/s. See also Figure 4.33).

This situation is clearly not optimal: data movement slows down the computation by almost a factor of 2, and the bandwidth of data movement is a factor of 4 lower than the practical value of ≈ 6 GB/s (see Figure 4.31).

Let's optimize the code by retaining the memory buffer into which matrices are copied between offloads. According to Figure 4.31, we should achieve around 6 GB/s for data movement.

```
// Allocate a buffer for all sets of matrices:
#pragma offload_transfer target(mic:0)  \
  in(A_buff: length(n*n) alloc_if(1) free_if(0)) \
  in(B_buff: length(n*n) alloc_if(1) free_if(0)) \
  in(C_buff: length(n*n) alloc_if(1) free_if(0))

  for(int i = 0; i < nMatrices; i++) {
    double* A = &A_arr[i*n*n];
    double* B = &B_arr[i*n*n];
    double* C = &C_arr[i*n*n];

    // Transfer the i-th set of matrices into the buffer
#pragma offload_transfer target(mic:0)  \
  in(A[0:n*n]: into (A_buff[0:n*n]))    \
  in(B[0:n*n]: into (B_buff[0:n*n]))

#pragma offload target(mic:0)                      \
  in(A_buff: length(0) alloc_if(0) free_if(0))     \
  in(B_buff: length(0) alloc_if(0) free_if(0))     \
  in(C_buff: length(0) alloc_if(0) free_if(0))
    { // Perform DGEMM on the coprocessor without moving data
      cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
           n, n, n, 1.0, A_buff, n, B_buff, n, 0.0, C_buff, n);
    }

   // Get the results out of the buffer
#pragma offload_transfer target(mic:0)  \
  out(C_buff[0:n*n]: into (C[0:n*n]))
  }

// Deallocate the buffer from the coprocessor
#pragma offload_transfer target(mic:0)  \
  in(A_buff: length(n*n) alloc_if(0) free_if(1)) \
  in(B_buff: length(n*n) alloc_if(0) free_if(1)) \
  in(C_buff: length(n*n) alloc_if(0) free_if(1))
```

Listing 4.79: DGEMM in the offload mode with memory buffer retention.

Listing 4.79 demonstrates a solution with memory buffer retention. We used just one buffer for all the sets of matrices that we needed to multiply. To copy the respective data set into the buffer, we used the specifier `into` in `#pragma offload_transfer`.

The result of this optimization is an increase in the effective performance (i.e., taking into account data movement time) by a factor of 4.6 to 269 ± 1 GFLOP/s. Additional timing inside the offload region shows that the performance on the coprocessor (i.e., neglecting data movement) is ≈ 600 GFLOP/s. Timing the `offload_transfer` pragmas shows that data movement takes around 55% of the total computation time and proceeds at a bandwidth of ≈ 5.8 GB/s.

We achieved two improvements with memory buffer retention:

1) Effective bandwidth of data movement to the coprocessor was accelerated from 1.6 to 5.8 GB/s, because with buffer retention, data movement does not involve memory allocation.

2) The performance of the computational part was boosted from 115 to 600 GFLOP/s. The root cause of the quenched compute performance with default offload settings (the 115 GFLOP/s measurements) is overhead present in the COI library. This overhead is related to the creation of virtuall memory pages for the newly allocated memory buffer. This overhead may be eliminated in later versions of MPSS.

Despite the increased performance, we see that the latency of data movement slows the computation down from the 600 GFLOP/s that it achieves directly on the coprocessor to an effective performance of 270 GFLOP/s. At the same time, because our goal is to multiply several sets of matrices, we could potentially gain another factor of 2 in performance by overlapping communication with computation. This optimization is shown in Section 4.6.2.

4.6.2. Masking Offload Latency with Double Buffering

In Section 4.6.1, the time spent on data movement turned out to be comparable to the computation time. This scenario can be optimized by overlapping some of the communication with computation using a technique known as double buffering. To execute double buffering, we can use the asynchronous offload functionality in the explicit offload model. See Lab 4.10 for complete implementation of this example (refer to Section 6.2).

To implement double buffering, we will have to create two sets of buffers: one for holding the matrices currently being multiplied and another one for holding matrices to be multiplied in the next iteration. Then we can start asynchronous offload executing DGEMM and, concurrently with it, start staging in data for the next set of matrices. The timeline in this algorithm is shown in Figure 4.32, where it is also juxtaposed with the single-buffer approach taken in Section 4.6.1.

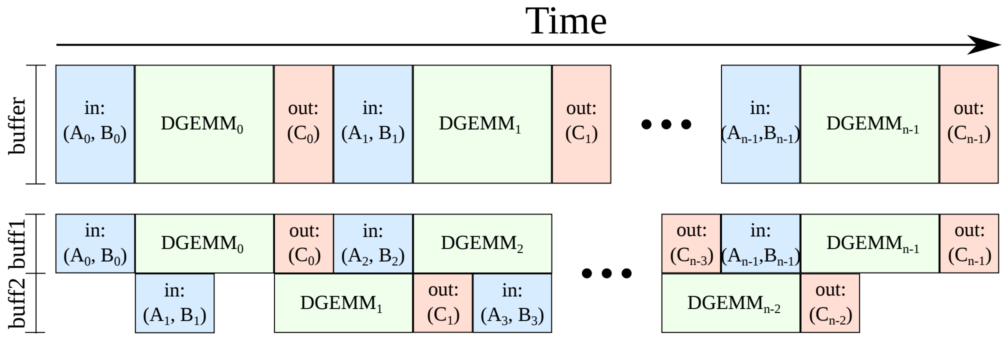


Figure 4.32: Timeline of single-buffered (non-optimized) and double-buffered (optimized) offload application running DGEMM on multiple sets of matrices A_i, B_i and C_i ($i = 0, 1, \ldots, n-1$).

With double buffering, the first iteration and the last iteration are special. In the first iteration, we have to send the first set of data into the coprocessor, however, there is no calculation to run in the background at that time. In the last iteration, we have to fetch the last set of results from the coprocessor, but there are no more concurrent calculations running at that time. This is why in our implementation (Listing 4.80), the loop in `i` runs from `1` to `nMatrices-2` The first and the last iteration are not shown in that listing, but can be found in the exercises accompanying the book.

```
for(int i = 1; i < nMatrices-1; i++) {
  double* A_trans = &A_arr[(i+1)*n*n]; // We send the next data set
  double* B_trans = &B_arr[(i+1)*n*n]; // We send the next data set
  double* C_trans = &C_arr[(i-1)*n*n]; // We receive previous reslt

#pragma offload target(mic:0)  signal(A_buff_calc)     \
    in(A_buff_calc: length(0) alloc_if(0) free_if(0)) \
    in(B_buff_calc: length(0) alloc_if(0) free_if(0)) \
    in(C_buff_calc: length(0) alloc_if(0) free_if(0))
    {
      cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
n, n, n, 1.0, A_buff_calc, n, B_buff_calc, n, 0.0, C_buff_calc, n);
    }

#pragma offload_transfer target(mic:0) \
    in(A_trans[0:n*n]: into (A_buff_trans[0:n*n])) \
    in(B_trans[0:n*n]: into (B_buff_trans[0:n*n])

#pragma offload_transfer target(mic:0) \
    out(C_buff_trans[0:n*n]: into (C_trans[0:n*n]))

#pragma offload_wait target(mic:0) wait(A_buff_calc)

// Swapping Buffers
 if(i%2==1) {
  A_buff_trans=A_buff2; B_buff_trans=B_buff2; C_buff_trans=C_buff2;
  A_buff_calc =A_buff1; B_buff_calc =B_buff1; C_buff_calc =C_buff1;
 } else {
  A_buff_trans=A_buff1; B_buff_trans=B_buff1; C_buff_trans=C_buff1;
  A_buff_calc =A_buff2; B_buff_calc =B_buff2; C_buff_calc =C_buff2;
 }
}
```

Listing 4.80: DGEMM in the offload mode with double buffering.

Double buffering masks data movement latency in all iterations except the first and the last one. The performance that we achieved is 391 ± 17 GFLOP/s. This performance is a result of overlapping most of the communication proceeding at an effective 5.0 GB/s with computation running on the coprocessor at about 600 GFLOP/s.

Figure 4.33 summarizes the results of Section 4.6.1 and 4.6.2.

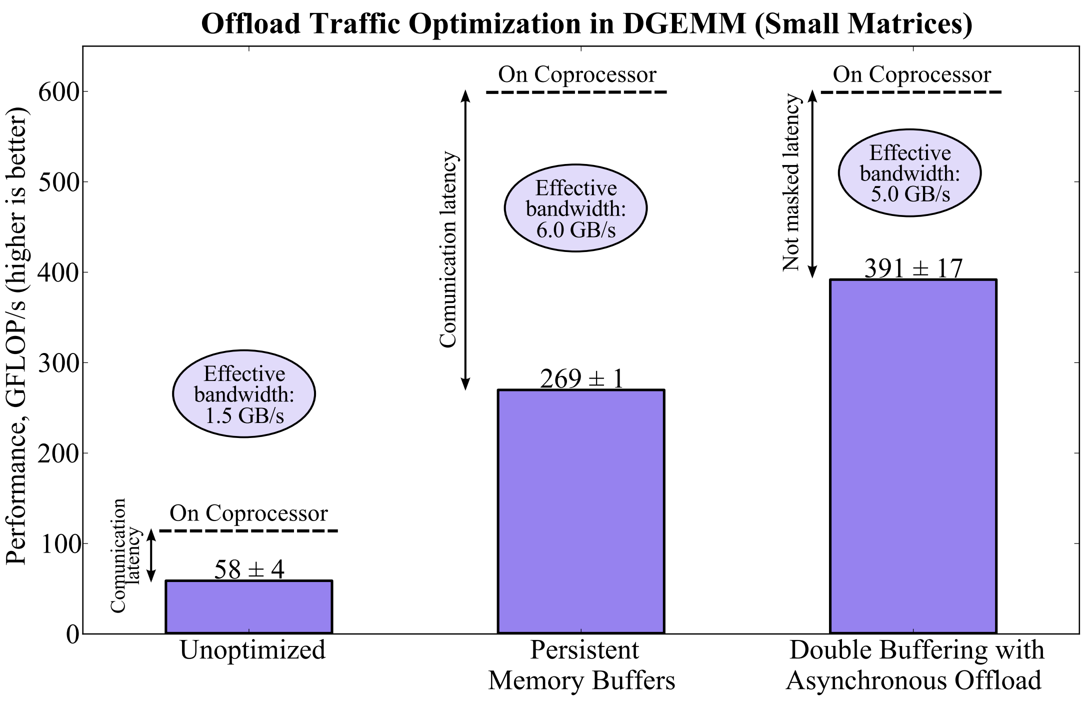


Figure 4.33: Optimization of DGEMM in the offload mode with memory retention and asynchronous offload for communication masking.

4.7. Optimization Strategies for MPI Applications

MPI applications in computing systems with Intel Xeon Phi coprocessors face three unique challenges:

1. When the workload is shared between the host processors and the coprocessors on an equal basis, the computing system becomes heterogeneous. In traditional homogeneous computing clusters, one may orchestrate work sharing based on the assumption that equal parts of the work take equal amounts time on any two compute devices. However, in computing systems with Intel Xeon Phi coprocessors, the same amount of work may take different time depending on whether it is processed by a host processor or by a coprocessors, because they perform computation at different rates. Therefore, work scheduling now must either take into account the relative performance of different compute units, or utilize dynamic scheduling to balance the workload.

2. Without multi-threading (shared memory parallelism), the total number of MPI processes on a single compute node (coprocessor) can be as high as 240. This an order of magnitude greater than in CPU-based systems. These numerous processes may produce excessive amounts of MPI communication. If communication quenches the performance of the algorithm, the programmer must consider communication-efficient algorithms or efficient communication fabrics. Hybrid OpenMP/MPI programming can be employed in order to reduce the number of MPI processes by utilizing multi-threading within each process.

3. Coprocessors have smaller amount of memory per core than typical CPU-based systems. Because of that, pure MPI applications requiring up to 240 processes per coprocessor can run into memory limitations. Once again, hybrid OpenMP/MPI approach may alleviate the load on the memory subsystem if some of the data can be shared between processes. In some cases, the number of threads per MPI process may be a tuning parameter of the application.

In this section, discuss these challenges and provide examples of MPI application optimization in these areas.

4.7.1. Static Load Balancing

Here we consider an application that uses Intel Xeon Phi coprocessors as additional compute nodes in a heterogeneous cluster (see Figure 3.6) and discuss load balancing possibilities for this setup.

Example Problem: Asian Option Pricing

We illustrate load balancing in MPI for an application that uses a Monte Carlo method for pricing Asian options (full code is available in Lab 4.11 – see Section 6.2). This problem does not require intensive data transfer, and every Monte Carlo trial is independent from other trials. Therefore, this method can be categorized as an *embarrassingly parallel* algorithm. Figure 4.34 schematically illustrates the core of the calculation.

1) Simulate Random-Walk of Asset Price

$$dS(t) = \mu S(t)dt + \sigma S(t)dB(t)$$

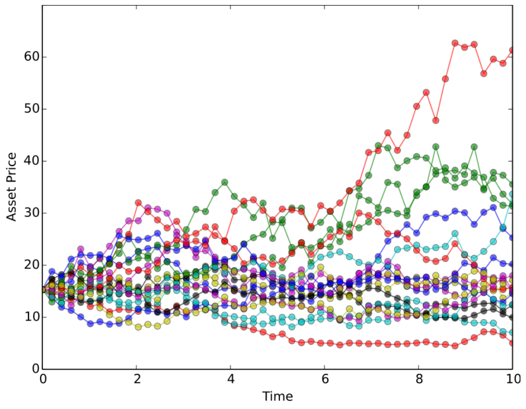
Using the Solution

$$S(t) = S(0)e^{(\mu-\frac{1}{2}\sigma^2)t+\sigma\sqrt{t}N(0,1)}$$

2) Perform Asian Option Price Averaging

$$\langle S\rangle_{\text{arithm}} = \frac{1}{N}\sum_{i=0}^{N-1} S(t_i),$$

$$\langle S\rangle_{\text{geom}} = \exp\left(\frac{1}{N}\sum_{i=0}^{N-1}\log S(t_i)\right)$$



3) Compute Discounted Pay-off

$$P_{\text{put}} = e^{-rT}\mathbb{E}\left(\max\left\{0; K - \langle S\rangle\right\}\right),$$

$$P_{\text{call}} = e^{-rT}\mathbb{E}\left(\max\left\{0; \langle S\rangle - K\right\}\right)$$

Figure 4.34: Asian option pricing.

Let's take a brief discourse to understand the underlying problem. Options are contracts which allow one party to buy or sell, on some future date, an asset (e.g., a stock) from/to the other party at a "strike price" agreed upon the signing of contract. A contract to buy is called a "call option", and a contract to sell is a “put option”. A style of options called Asian options has the feature that the option payoff is calculated based on the mean price (arithmetic or geometric) of the asset, sampled at prearranged instances. This reduces the risks associated with market volatility and short-term market manipulation. To make profit, the seller of the option must set a price that offsets the anticipated risks associated with the asset price fluctuations.

For risk analysis of Asian options, a Monte Carlo simulation method can be used (see Figure 4.34). In this method, multiple stochastic histories of the asset price are simulated based on the available information on the asset volatility. Details of the method are discussed in [40]. Here, we will focus on the implementation of the parallel algorithm for this Monte Carlo method.

Suppose our task is to price N options, where for each option we have different sets of parameters such as starting price, volatility, time averaging interval, etc. For each option, we will simulate P random paths (i.e., stock price timelines) and perform statistical analysis using these simulations. To parallelize this algorithm in a cluster, we will adopt the following approach:

1) N options (i.e., sets of parameters for option pricing) are distributed across all MPI processes.

2) Each process will analyze one option at a time.

3) Within each option, P random paths will be distributed across OpenMP threads, and within each thread, across multiple vector lanes.

We have already covered the optimization of vector and multi-threaded calculations, and therefore here, we will only focus on the distribution of work across MPI processes.

Asian Option Pricing without Load Balancing

In Listing 4.81 we demonstrate an initial approach to the problem in MPI.

```
1  void ComputeOnAllNodes(
2    const int nOptions, // Number of option parameters to price
3    const OptionType* const option, // Array of option parameters
4    PayoffType* payoff, // Array of option parameters
5    const int mpiWorldSize, // Size of MPI world for load distribution
6    const int myRank,         // My ID
7    ) {
8
9    //Calculating workload share based on the rank
10   const double optionsPerProcess =
11                               double(nOptions)/double(mpiWorldSize);
12   const int myFirstOption = int(optionsPerProcess*(myRank));
13   const int myLastOption  = int(optionsPerProcess*(myRank+1));
14
15   // Static, even load distribution: assign options to ranks
16   for (int i = myFirstOption; i < myLastOption; i++) {
17     ComputeOptionPayoffs(option[i], payoff[i]);
18   }
19
20   // Collect results from all processes to reportingRank
21   if (myRank == reportingRank) {
22     MPI_Reduce(MPI_IN_PLACE, (float*)payoff, 4*nOptions,
23                MPI_FLOAT, MPI_SUM, reportingRank, MPI_COMM_WORLD);
24   } else {
25     MPI_Reduce((float*)payoff, (float*)payoff, 4*nOptions,
26                MPI_FLOAT, MPI_SUM, reportingRank, MPI_COMM_WORLD);
27   }
28 }
```

Listing 4.81: Even load distribution in Asian option pricing.

Here, function `ComputeOptionPayoffs()` receives one set of option parameters and performs a computation for it. Multi-threading and vectorization are inside of this function.

From the calculation in lines 10-13, it is obvious that each MPI process receives approximately the same number of options to price. If each work-item (i.e., option) takes the same amount of time to process, then load balancing will not be an issue on a homogeneous platform. However, once the platform becomes heterogeneous (i.e., some MPI processes run on CPUs and other on coprocessors), load imbalance may occur.

We benchmarked this calculation, first using just the host CPU with two MPI processes (one process per socket) as shown in Listing 4.82. This configuration yielded a performance of $2.9 \cdot 10^9$ random values processed per second (see also the first set of bars in Figure 4.41).

```
vega@lyra% # Linking host executable called "app"
vega@lyra% mpiicpc -qopenmp -mkl -xhost -o app *.o #
vega@lyra% cat machines-cpu.txt # Machine file for CPU-only run
lyra:2
vega@lyra% mpirun -machine machines-cpu.txt $PWD/app # Run on host
// ... Executable "app" runs on CPUs
```

Listing 4.82: Executing an MPI application on the CPU architecture.

After that, we used two Intel Xeon Phi coprocessor for the calculation, placing one MPI process on each (see Listing 4.83). This resulted in $6.2 \cdot 10^9$ values per second performance.

```
vega@lyra% # Linking coprpcessor executable called "app-MIC"
vega@lyra% mpiicpc -qopenmp -mkl -mmic -o app-MIC *.oMIC # MIC exec
vega@lyra% cat machines-mic.txt # Machine file for MIC-only run
lyra-mic0:1
lyra-mic1:1
vega@lyra% export I_MPI_MIC=1 # Enable MIC support in Intel MPI
vega@lyra% LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MIC_LD_LIBRARY_PATH
vega@lyra% export LD_LIBRARY_PATH # Locate MIC libraries
vega@lyra% mpirun -machine machines-mic.txt $PWD/app # Run on MIC
// ... Executable "app-MIC" runs on coprocessors
```

Listing 4.83: Executing an MPI application on the MIC architecture.

Finally, we joined forces of the CPU and the coprocessors and ran a heterogeneous calculation (Listing 4.84). This calculation performed at $5.8 \cdot 10^9$ values per second, which is worse than coprocessor-only setup.

```
vega@lyra% cat machines-het.txt # Machine file for heterogeneous
lyra:2
lyra-mic0:1
lyra-mic1:1
vega@lyra% export I_MPI_MIC=1 # Enable MIC support in Intel MPI
vega@lyra% LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MIC_LD_LIBRARY_PATH
vega@lyra% export LD_LIBRARY_PATH # Locate MIC libraries
vega@lyra% export I_MPI_MIC_POSTFIX="-MIC" # Get name of MIC exec.
vega@lyra% mpirun -machine machines-het.txt $PWD/app # Run on all
// ... Executable "app" runs on CPUs
// ... Executable "app-MIC" runs on coprocessors
```

Listing 4.84: Executing an MPI application on a heterogeneous system comprised of processors and coprocessors.

Note that in the last setup, we had to use different executables for the CPU architecture and for the coprocessor architecture. The former was named `app` and the latter `app-MIC`. To use a machine file in this setup (which restricts us to specifying only one executable in the command line), we employed the environment variable `I_MPI_MIC_POSTFIX` (see Section 2.4.3 for more information).

Obviously, the latter performance result of $5.8 \cdot 10^9$ values per second is disappointing. Considering the embarrassingly parallel nature of the problem, we would expect the performance of the host to add up with the performance for the coprocessors for a total of $2.9 \cdot 10^9 + 6.2 \cdot 10^9 = 9.1 \cdot 10^9$ values per second.

As mentioned above, the problem may be traced to imperfect load balance. Indeed, in our setup, half the options were processed on the host and the other half on two coprocessors. Coprocessors were done with their share of the work while the host was only half way done with its share.

One of the ways to remedy this situation is static load balancing.

Static Load Redistribution

To redistribute the load, we can introduce a work sharing parameter α quantitatively defined as

$$\alpha = \frac{B_{\mathrm{CPU}}}{B_{\mathrm{MIC}}}, \tag{4.13}$$

where B_{CPU} and B_{MIC} are the number of options processed on the CPU architecture and on the MIC architecture, respectively. If the number of CPU-based ranks is equal to the number of MIC-based ranks, then the value $\alpha = 1.0$ reproduces the case of unbalanced application with even load distribution (Listing 4.81). Values $\alpha > 1.0$ assign more work to processes running on CPUs. Correspondingly, for $\alpha < 1.0$, all CPU processes receive less work than all MIC architecture processes. The optimal value of α depends on the specific problem and computing system components (the number of coprocessors, the clock frequency of host processors, etc).

An implementation of a code with static load balancing will have to compute the following:

1) For each MPI process, it will have to determine whether it runs on a CPU or on a MIC architecture coprocessor.
2) Each MPI process will have to know its rank within the group performing computation on CPUs or on coprocessors.

To perform the former task, an array `rankTypes` of size equal to the MPI world size may be computed and propagated across all processes as shown in Listing 4.85.

```
int rankTypes[mpiWorldSize];
rankTypes[:] = 0;
MPI_Barrier(MPI_COMM_WORLD);
#ifdef __MIC__
rankTypes[myRank] = 1;
#endif
MPI_Allreduce(MPI_IN_PLACE, &rankTypes, mpiWorldSize,
              MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

Listing 4.85: Determining which ranks belong to the CPU and to the MIC architecture.

To determine each rank's place within the CPU or the MIC team, the MPI functionality of groups may be used as shown in Listing 4.86.

```
// Create two new groups: all CPUs and all MICs:
// 1. Create a list of ranks that are CPUs and MICs
std::vector<int> cpuRanks, micRanks;
for (int i = 0; i < mpiWorldSize; i++) {
  if (rankTypes[i] == 0) cpuRanks.push_back(i);
  if (rankTypes[i] == 1) micRanks.push_back(i);
}

// 2. Create MPI groups, one of CPUs and another of MICs
MPI_Group newGroup, origGroup;
MPI_Comm_group(MPI_COMM_WORLD, &origGroup);
if (rankTypes[myRank] == 0) {
  MPI_Group_incl(
          origGroup, cpuRanks.size(), &cpuRanks[0], &newGroup); }
else {
  MPI_Group_incl(
          origGroup, micRanks.size(), &micRanks[0], &newGroup); }

// 3. Query my place in the new group
int myGroupRank;
MPI_Group_rank(newGroup, &myGroupRank);
```

Listing 4.86: Determining the rank of a process among the group of processes running on the same architecture.

The result will be an integer `myGroupRank`, which is a zero-based identificator of the process within the group comprised of processes on the same architecture as the current rank.

Note that this functionality is closely related to MPI communicators. A communicator may be created with the new groups, allowing collective communication that proceeds only within one of the groups and not within the MPI world. This note is purely informational; we do not need to use the communicator functionality for the problem at hand.

With the groups created, we can query the parameter α (in this example we set it via the environment variable `OPTIONS_ALPHA`) and compute the fraction of the workload to be processed on all CPU ranks as shown in Listing 4.87.

```
double alpha = 1.0; // Default value: equal work to CPUs and MICs
if (getenv("OPTIONS_ALPHA") != NULL)
  alpha = atof(getenv("OPTIONS_ALPHA")); // Load balancing param.

const int lastOptForCPUs =
    int(alpha*nOptions*double(cpuRanks.size())/mpiWorldSize);
```

Listing 4.87: Determining which ranks belong to the CPU and to the MIC architecture.

Finally, static load balancing may be expressed as in Listing 4.88.

```
int myFirstOpt, myLastOpt;
if (rankTypes[myRank] == 0) { // I am a MIC-based rank
  const double optionsPerProcess =
               double(lastOptForCPUs)/double(cpuRanks.size());
  myFirstOpt = int(optionsPerProcess*(myGroupRank));
  myLastOpt  = int(optionsPerProcess*(myGroupRank+1));
} else { // I am a CPU-based rank
  const double optionsPerProcess =
        double(nOpts-lastOptForCPUs)/double(micRanks.size());
  myFirstOpt=lastOptForCPUs+int(optionsPerProcess*(myGroupRank));
  myLastOpt=lastOptForCPUs+int(optionsPerProcess*(myGroupRank+1));
}

// This rank will process options from myFirstOpt to myLastOpt
for (int i = myFirstOpt; i < myLastOpt; i++) {
  ComputeOptionPayoffs(option[i], payoff[i]);
}

// ... MPI_Reduce to follow as before
```

Listing 4.88: Statically partitioning the workload beween MPI processes.

To take advantage of the load balancing property of the implementation in Listing 4.88, we will need to tune the value of α. This may be done either by analytical estimates, or by performing a calibration run. To illustrate the impact of α on performance, we did the latter, and results of calibration are shown in Figure 4.35.

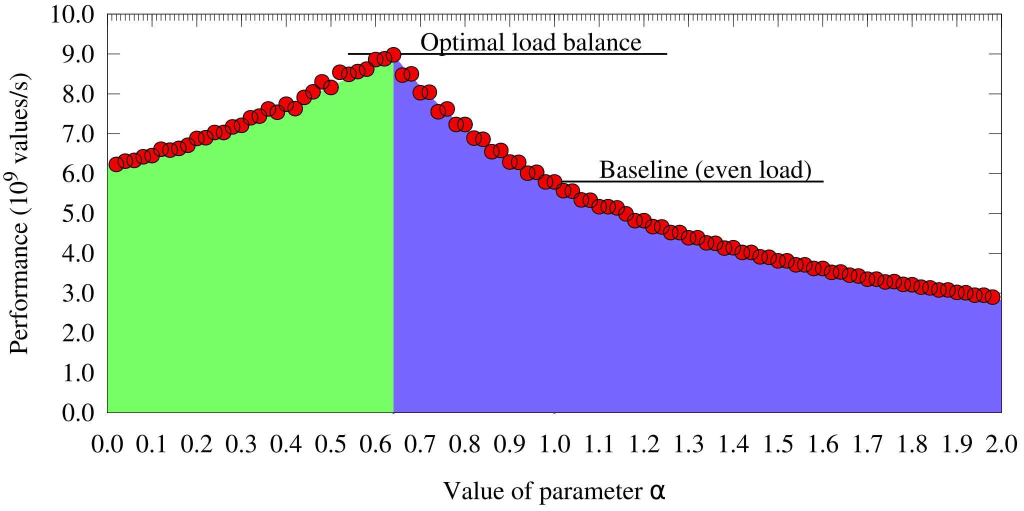


Figure 4.35: Load balancing: performance of the Asian options pricing Monte Carlo code as a function of the parameter α.

The optimal performance of the heterogeneous code that uses two processes on a two-way CPU and one process on each of two coprocessors is $9.0 \cdot 10^9$ values per second (see Figure 4.41). This is close to the theoretical maximum performance that we estimated assuming perfect scalability across the processors and coprocessors of our system. The difference between the unbalanced application and the application with static load balancing is depicted in Figure 4.36 and Figure 4.37.

Static load balancing allowed us to scale a Monte Carlo calculation across the host and the coprocessor with near-optimal performance. However, this approach is not always optimal. Static load balancing requires fine tuning of parameters for every computing system configuration, which may be inconvenient. In addition, this approach may not produce good results if the calculation time varies from one work item to another. In this situation, some processes may be "unlucky" to receive a longer workload, and all other processes will have to wait for the slowest process at the synchronization point. The solution to both problems is dynamic load balancing, which we discuss in Section 4.7.2 and Section 4.7.3.

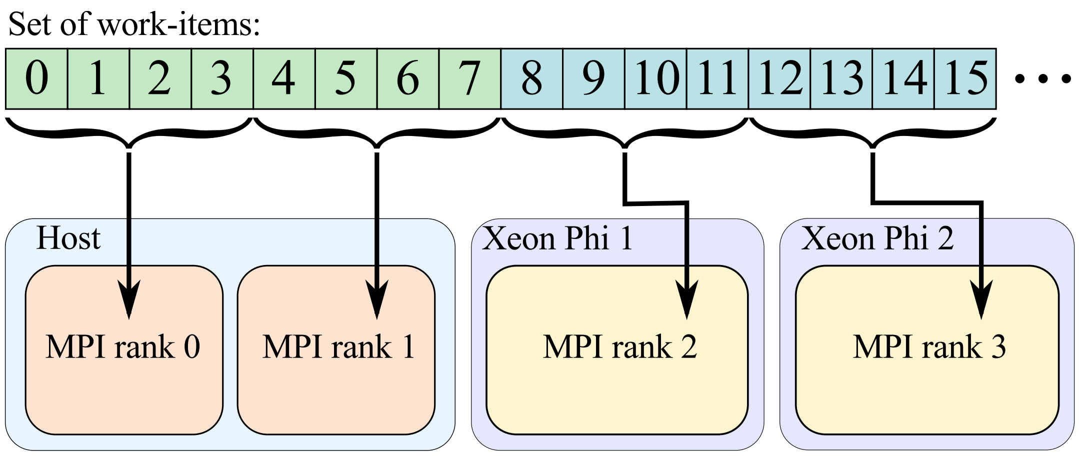


Figure 4.36: Heterogeneous MPI application without load balancing. Each MPI process receives the same share of work for processing, regardless of the architecture that it is executing on.

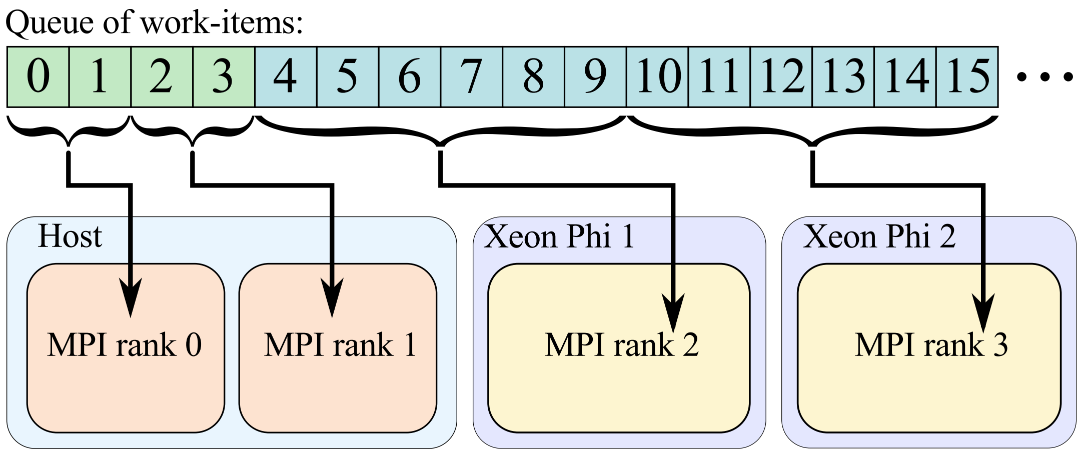


Figure 4.37: Heterogeneous MPI application with static load balancing. The share of work for each process is determined by the architecture on which the process is executing. Mapping of work-items to processes is assigned statically at the beginning of the run.

4.7.2. Dynamic Work Scheduling

With dynamic load balancing, MPI processes that finish with their share of work receive additional work. If the scheduling scheme is optimized, then it is not necessary to calibrate the application for every system configuration, and fluctuations of the execution time from one part of the problem to another are naturally absorbed.

In this section we will show how to implement dynamic load balancing in the *boss-worker* model using MPI communication between processes. *Boss* — one of the MPI processes — is dedicated to assigning parts of the problem (called "work-items" in this context) from a global queue to *workers* — the rest of MPI processes in the application. When a worker finishes its assigned work-item, it reports back to the boss to receive either another item, or a command to terminate calculations.

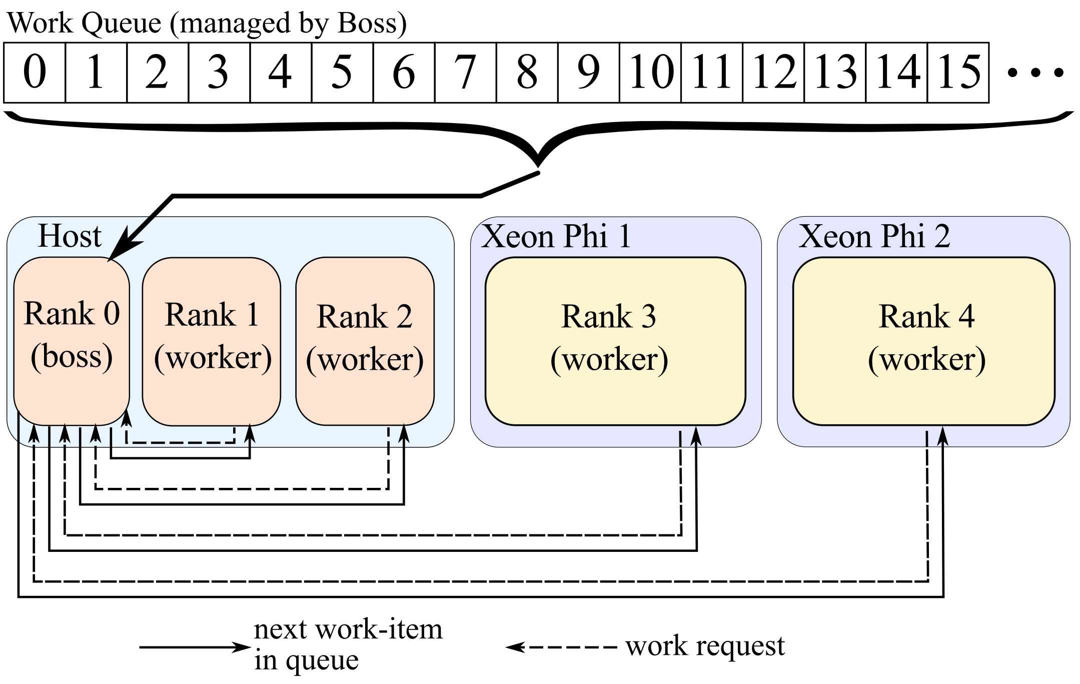


Figure 4.38: Heterogeneous MPI application with dynamic load balancing in the boss-worker model. One of the MPI processes, "boss" (in this case, rank 0 process), is dedicated to distributing work-items to other processes (workers).

Boss-Worker Scheme Implementation

To illustrate dynamic load balancing in MPI, we will use the same example problem as in Section 4.7.1, Asian option pricing with a Monte Carlo method (full code is available in Lab 4.11 – see Section 6.2).. The core of the calculation remains the same, however, additional communication for dynamic work scheduling is included in the code of each process. The code that implements this scheduling algorithm is shown in Listing 4.89, Listing 4.90 and Listing 4.91.

Listing 4.89 shows code executed in every MPI process. This code branches on the value of the rank: the boss (rank 0) runs the function `DistributeWork()` and workers (all other ranks) run `ReceiveWork()`. At the end of the calculation, an MPI barrier is used before collective communication which delivers results for reporting. In some applications, this part of the work may not be necessary, as results can be delievered inside of `DistributeWork()/ReceiveWork()`.

```
void ComputeOnAllNodes(
  const int nOptions, // Number of option parameters to price
  const OptionType* const option, // Array of option parameters
  PayoffType* payoff, // Array of option parameters
  const int mpiWorldSize, // Size of MPI world for load distribution
  const int myRank        // My ID
  ) {
  MPI_Status mpiStatus;

  if (myRank == 0) // Boss's branch
    DistributeWork(nOptions, option, mpiWorldSize);
  else // Workers' branch
    ReceiveWork(option, payoff, myRank);

  MPI_Barrier(MPI_COMM_WORLD);

  // ... MPI_Reduce to follow as before
}
```

Listing 4.89: Dynamic partitioning of the workload beween MPI processes.

In Listing 4.90, the worker part of the code is shown. Before a worker begins calculation, it waits for a message from the boss (rank 0) with the value of `optionIdx` indicating the number of the work-item to process. In our implementation, all workers already have the data for all work items, so only the indices of work-items are exchanged. Naturally, in other applications, this message exchange can also carry the data of the work-item and, potentially, return the result to the boss process. When the work-item is processed, the worker waits for more messages, until the received message contains termination signal, which indicates the end of the calculation. This scheme is similar to the dynamic scheduling mode for OpenMP loops (see Section Section 4.4.3).

```
void ReceiveWork(
  const OptionType* const option, // Array of option parameters
  PayoffType* payoff, // Array of option parameters
  const int myRank,    // My ID
  ) {
  int optionIdx = 0;
  MPI_Status mpiStatus;
  bool terminate = false;

  MPI_Send(&myRank,1,MPI_INT,0,1,MPI_COMM_WORLD); // Request work
  while(!terminate) {
    // Get the next option to process
    MPI_Recv(&optionIdx,1,MPI_INT,0,1,MPI_COMM_WORLD,&mpiStatus);
    if(optionIdx == terminate_val) {
      terminate = true;
    } else {
      // Process the assigned option
      ComputeOptionPayoffs(option[optionIdx], payoff[optionIdx]);

      // Request more work
      MPI_Send(&myRank,1,MPI_INT,0,1,MPI_COMM_WORLD);
    }
  }
}
```

Listing 4.90: Worker algorithm for dynamic load distribution.

Finally, Listing 4.91 contains the boss part of the process. The main loop in this code proceeds to distribute indices of work-items to process between the workers, untill all work-items have been processed. After that, the boss sends out termination signals to all workers. Note the usage of the wildcard source mask `MPI_ANY_SOURCE` in `MPI_Recv()`. This is the key in dynamic scheduling: whichever worker is the first to report for work will receive the next work-item.

```
void DistributeWork(
  const int nOptions, // Number of option parameters to price
  const OptionType* const option, // Array of option parameters
  const int mpiWorldSize // Size of MPI world for load distribution
  ) {
  int option_index = 0;
  MPI_Status mpiStatus;

  // Distribute option parameters to work on, one by one
  int terminates_sent = 0;
  while(terminates_sent < mpiWorldSize-1) {
    int workerId;
    // Wait for a request for work from any worker
    MPI_Recv(&workerId,1,
             MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&mpiStatus);
    if(option_index < nOptions) {
      // Assign the next option to the requesting worker
      MPI_Send(&option_index,1,MPI_INT,workerId,1,MPI_COMM_WORLD);
      option_index++;
    } else {
      // All work completed; send termination signal
      MPI_Send(&terminate_val,1,MPI_INT,workerId,1,MPI_COMM_WORLD);
      terminates_sent++;
    }
  }
}
```

Listing 4.91: Boss algorithm for dynamic load distribution.

MPI Process Pinning and Accommodating the Boss Process

The boss-worker implementation listed above assumes that the boss is rank 0, and no work is performed in this rank. Because we have multi-threading inside of MPI processes (as shown in Figure 3.6), the cores assigned to the boss will be idle. It may not be a problem if the number of workers is much greater than 1, however, in our setup with 2 processes on the host (see Figure 4.38), the compute power wasted in the idling threads of the boss process can become costly.

To assign fewer threads to the boss (and, therefore, to have fewer idling threads), it is possible to execute, for instance, 3 processes on the host instead of 2. However, there still will be a problem rooted in the way that Intel MPI inter-operates with OpenMP. By default, Intel MPI uses process pinning, i.e., affinity of processes to hardware components is enforced. This generally has a positive effect, ensuring NUMA locality of computation akin to the example with system partitioning in Section 4.4.5. However, with 3 processes, process pinning will still dedicate several cores to the boss and fewer than half the cores of the system to each worker.

One possible solution to this problem is to disable pinning. This can be done by setting the environment variable `I_MPI_PIN=0` as shown in Listing 4.92.

```
vega@lyra% cat machines-cpu.txt # Machine file for CPU-only run
lyra:1 # Boss
lyra:2 # Two workers
vega@lyra% export I_MPI_PIN=0 # Disable process pinning in MPI
vega@lyra% mpirun -machine machines-cpu.txt $PWD/app # Run on host
// ... Executable "app" runs on CPUs
```

Listing 4.92: Disabling process pinning in an MPI application with a boss process.

Indeed, with pinning disabled, we can safely increase the number of threads per worker so that workers utilize all cores. The boss will have minimal interference with the workers, as it will migrate across all threads in the system. This method has worked well in other applications (e.g., [40] uses more paths per option and works well with disabled pinning because the load per core is greater). However, in our case, disabling pinning leads to performance degradation because affinity of threads to cores is lost.

On the host, we achieve a performance of $1.6 \cdot 10^9$ values per second, on two coprocessors $3.9 \cdot 10^9$ values per second, and on the host together with two coprocessors $6.3 \cdot 10^9$ values per second (see the third set of bars in Figure 4.41). Interestingly, the scaling from CPU-only and MIC-only to heterogeneous calculation appears to be "super-linear", i.e., $6.3 > (1.6+3.9)$. This is because when the boss process is moved from coprocessors to the host, the performance of the coprocessors is improved. The net result in the heterogeneous system is better than that without load balancing (we measured $5.8 \cdot 10^9$ values per second in this case), but worse than with static scheduling ($9.0 \cdot 10^9$ values per second).

The messages of this exercise are:

1. Dynamic load scheduling allows to improve heterogeneous application performance over schemes with no scheduling.
2. The presence of a boss process that manages the queue may detract resources from the system, which may be a significant penalty if the system is not very large.
3. Process pinning in Intel MPI has positive effect on performance.
4. Disabling process pinning in order to accommodate the boss process may or may not work depending on the nature of the workload.

Additionally, it is worth mentioning that the boss-worker scheduling scheme may require some tuning. For instance, if the latency of communication becomes comparable to the work-item processing time, it may be beneficial to increase the granularity of scheduling. E.g., handing out multiple work-items to each worker may improve results.

Besides the centralized boss-worker scheme, dynamic load balancing may be performed using collective scheduling schemes such as work stealing [41].

A general drawback of dynamic load balancing schemes is that from one run to another, the distribution of work across MPI processes may vary depending on runtime conditions and MPI message arrival times. As a consequence, the result of a calculation is not bitwise-reproducible from run to run if the calculation involves random numbers or not precisely associative operations. This is true of almost all applications except those operating exclusively on integers.

In Section 4.7.3 we will illustrate another Intel MPI functionality: multi-threaded communication. This will allow us to improve our boss-worker scheme, because we will be able to retain process pinning.

4.7.3. Multi-threading within MPI Processes

Intel MPI inter-operates with Intel OpenMP by allowing multi-threading inside of MPI processes. Intel MPI detects NUMA nodes, ordering of OS procs and cache sharing between cores, and uses this information for partitioning the system between multiple MPI processes. The tool `cpuinfo` shows the architecture information available to Intel MPI (see Listing 4.93).

```
vega@lyra% cpuinfo
...
=====  Processor composition  =====
Processor name    : Intel(R) Xeon(R)  E5-2697 v2
Packages(sockets) : 2
Cores             : 24
Processors(CPUs)  : 48
Cores per package : 12
Threads per core  : 2

=====  Processor identification  =====
Processor    Thread Id.        Core Id.        Package Id.
0            0                 0               0
1            0                 1               0
2            0                 2               0
...
12           0                 0               1
13           0                 1               1
...
=====  Placement on packages  =====
Package Id.  Core Id.                    Processors
0            0,1,2,3,4,5,8,9,10,11,12,13  (0,24)(1,25)(2,26)...
1            0,1,2,3,4,5,8,9,10,11,12,13  (12,36)(13,37)(14,38)...

=====  Cache sharing  =====
Cache  Size    Processors
L1     32  KB   (0,24)(1,25)(2,26)(3,27)(4,28)(5,29)(6,30)(7,31)...
L2     256 KB   (0,24)(1,25)(2,26)(3,27)(4,28)(5,29)(6,30)(7,31)...
L3     30  MB   (0,1,2,3,4,5,6,7,8,9,10,11,24,25,26,27,28,29,30,...
```

Listing 4.93: Using the Intel MPI tool `cpuinfo`.

This section discusses two models of multi-threading in MPI:

1. computing in multiple threads, communicating from one thread, and
2. computing and communicating from multiple threads.

Tuning Thread and Process Parallelism

By default, the MPI runtime library partitions the system evenly between all processes running on any given compute node, and uses process pinning to restrict the OpenMP scaling in each process to its respective partition of the resources. As a part of the pinning process, Intel MPI sets the number of OpenMP threads for each process. To control the number of MPI processes versus the number of OpenMP cores, the best method is modifying number of processes per host specified in the machine file.

The optimal number of MPI processes per CPU or per coprocessor may be a tuning parameter of the application. For example, in the case of the Monte Carlo code (Section 4.7.1), using two workers per host rather than one allowed to pin them to their respective sockets. This improves NUMA locality of the data used by the processes. As another example, the dynamic load balancing scheme (Section 4.7.2) may suffer if too many processes are used per node because the amount of MPI communication will increase.

Tuning the number of MPI processes per host can be even more important in bandwidth-bound applications. For example, in [22], tuning the number of threads was necessary to control the length of the inner loops, effectively implementing domain tiling across different MPI processes (see Figure 4.39).

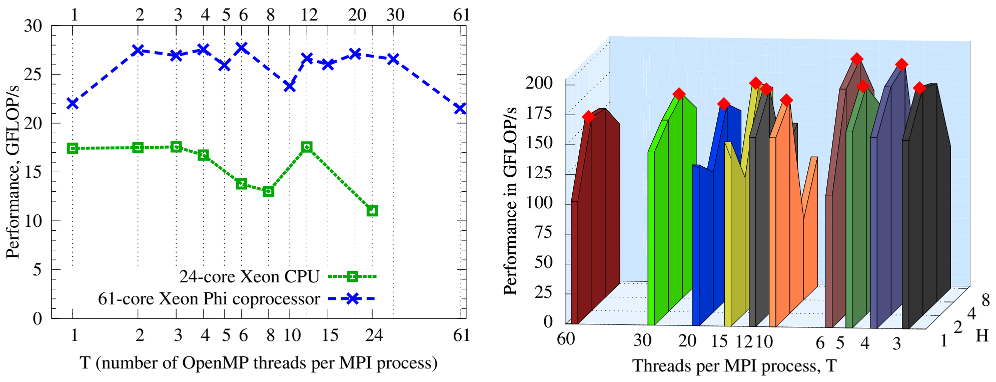


Figure 4.39: The number of OpenMP threads per MPI process may be a tuning parameter of the application (figures from [22]).

MPI Calls from OpenMP Threads

By default (i.e., if the procedures shown in Listing 2.11 and Listing 2.12 are followed), inter-operation between MPI and OpenMP may not involve MPI calls from a parallel OpenMP region. In applications that must perform MPI communication from OpenMP threads within MPI processes, special measures must be taken.

1. The thread-safe version of Intel MPI Library must be linked by using the compiler flag `-mt_mpi`.

2. MPI must be initialized with the call `MPI_Init_thread()` as shown in Listing 4.94.

```c
int required=MPI_THREAD_SERIALIZED;
int provided;

MPI_Init_thread(&argc, &argv, required, &provided);

if (provided < required){
   if (rank == 0)
      printf("Requested threading support is not available.\n";
   exit(1);
}
```

Listing 4.94: Hybrid OpenMP and MPI initialization.

Here, parameter `required` parameter can be one of the following:

`MPI_THREAD_SINGLE` The process is single-threaded.

`MPI_THREAD_FUNNELED` The process may be multi-threaded, but the application must ensure that only the main thread makes MPI calls.

`MPI_THREAD_SERIALIZED` The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time.

`MPI_THREAD_MULTIPLE` Multiple threads may call MPI, with no restrictions.

The call to `MPI_Init_thread()` will set the value of parameter `provided` to the value granted by the implementation.

Example: Boss-Worker Model with Multi-Threading

We can use a multi-threaded MPI implementation to resolve the difficulty with pinning and allocating resources to the boss process which we encountered in Section 4.7.2. To do this, we will dedicate one thread in process with rank 0 to scheduling. All other threads will be dedicated to work processing. That is, rank 0 will contain both the boss and the worker. Communication in this rank will occur from two OpenMP threads because the boss will have to schedule some of the work to the worker in rank 0.

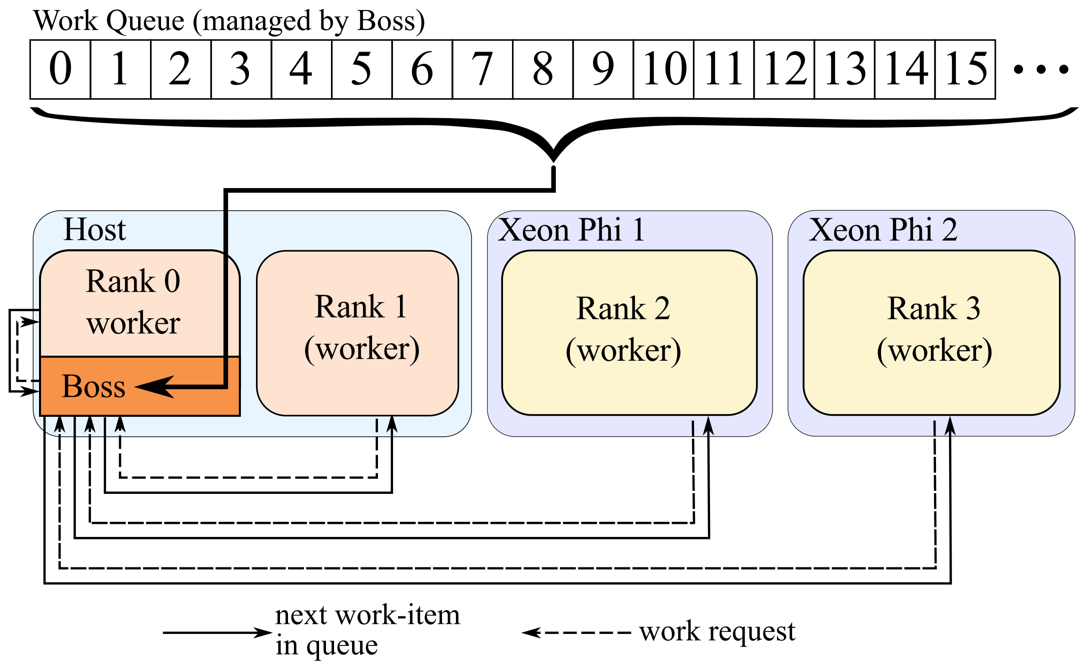


Figure 4.40: Heterogeneous MPI application with dynamic load balancing in the boss-worker model. One of the MPI processes, "boss" (in this case, rank 0 process), contains a thread dedicated to distributing work-items to other processes (workers).

The implementation of the algorithm will have to change very little. Specifically, the code that branches the execution in rank 0 (originally, Listing 4.89) will have to be modified by incorporating an OpenMP parallel region with a group of `omp section` pragmas (see Listing 4.95).

```
void ComputeOnAllNodes(
  const int nOptions, // Number of option parameters to price
  const OptionType* const option, // Array of option parameters
  PayoffType* payoff, // Array of option parameters
  const int mpiWorldSize, // Size of MPI world for load distribution
  const int myRank        // My ID
  ) {
  MPI_Status mpiStatus;

  if(myRank == 0) { // Rank 0 has both a boss and a worker inside:
    const int nThreads = omp_get_max_threads();
    omp_set_nested(1);
#pragma omp parallel sections num_threads(2)
    {
#pragma omp section
      { DistributeWork(nOptions, option, mpiWorldSize); } // Boss
#pragma omp section
      { omp_set_num_threads(nThreads-1); // Worker in rank 0:
        ReceiveWork(option, payoff, myRank, optioncount);
    }
  } else { // Only workers in all other ranks:
    ReceiveWork(option, payoff, myRank, optioncount);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  // ... MPI_Reduce to follow as before
}
```

Listing 4.95: Dynamic partitioning of the workload beween MPI processes.

Note that we had to enable nested parallelism in OpenMP because the function `ReceiveWork()` calls `ComputeOptionPayoffs()`, which contains an OpenMP parallel region inside. Additionally, in order to use 2 threads in the outer parallel region (which provides a separate thread for the boss), but many more threads in the inner parallel region (for the worker), we saved the number of threads prior to spawning and used it in the worker section.

Another change in the code that will be required is the number of workers in the boss function `DistributeWork()` (Listing 4.91). Instead of waiting for `(mpiWorldSize-1)` termination requests, it must now wait for `mpiWorldSize` requests.

The new application may be executed with pinning enabled, and without creating a separate rank for the boss (i.e., as in Listing 4.82, Listing 4.83 and Listing 4.84).

The performance results that we achieved are: $2.8 \cdot 10^9$, $5.2 \cdot 10^9$ and $8.6 \cdot 10^9$ random values per second on the host, on coprocessors and on both, respectively (see the fourth set of bars in Figure 4.41). A slight decrease in performance observed in MIC-only calculation from accommodating a boss thread is resolved in the heterogeneous run, which achieves a performance equal to 96% of that of the statically balanced run. However, unlike the latter, the implementation with dynamic load scheduling does not require tuning (i.e., scanning for the optimal value of parameter α).

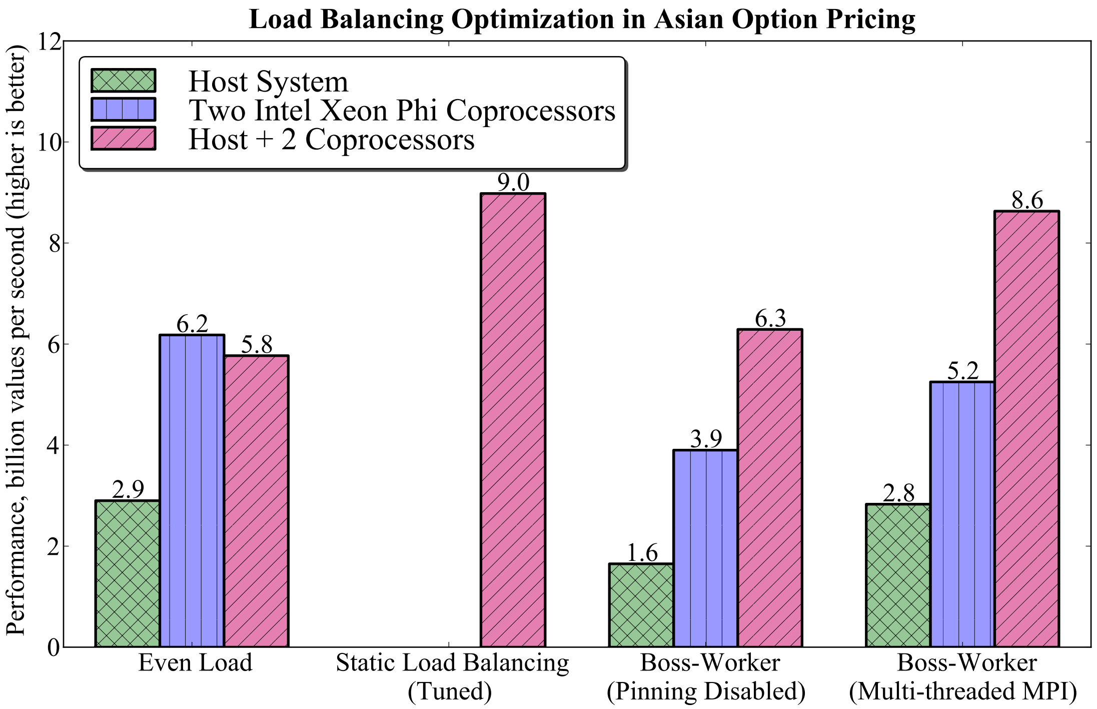


Figure 4.41: Performance of the Monte Carlo code for Asian Options pricing with different scheduling modes.

4.7.4. Fabric Control

When MPI applications using Intel Xeon Phi coprocessors rely on offload programming (Figure 3.7), MPI communication takes place only between hosts. In this case, tuning and control of communication fabrics is no different from that in CPU-only clusters.

However, when an application uses Intel Xeon Phi coprocessors in the native model (Figure 3.6), and coprocessors operate as independent IP-addressable manycore nodes in a computing cluster, communication between coprocessors may take place across InfiniBand (Figure 1.12). In this case, there is additional freedom for tuning and fabric control.

In this section we demonstrate the usage of Intel Xeon Phi coprocessors in clusters connected with Gigabit Ethernet as well as InfiniBand interconnects, and report the latencies and bandwidths of MPI messages with and without InfiniBand support.

Interconnect Technologies

Three types on commonly used network interconnects may be used in clusters with Intel Xeon Phi coprocessors:

Gigabit Ethernet is immediately available in most computing systems. It can be used for MPI communication between hosts as well as for peer-to-peer messaging between coprocessors. Intel Xeon Phi coprocessors do not have Ethernet ports, however, they can attach to the host's network adapter via the Linux functionality of bridging (see Section 1.2.5).

InfiniBand is an RDMA-capable interconnect technology. It may be used for host-to-host, host-to-coprocessor and coprocessor-to-coprocessor communication. Coprocessors do not have on-board InfiniBand adapters, but they can interact with the network of the hosts through CCL or PSM (Section 1.2.5). Two brands of InfiniBand interconnects are supported by coprocessors: Mellanox and Intel True Scale. The libraries and protocols used by these two brands are different and not interchangeable.

10 Gigabit and 40 Gigabit Ethernet may be used in place of Gigabit Ethernet, however, it will work only for communication between hosts.

As of MPSS 3.4.1, Intel Xeon Phi coprocessors are unable to take advantage of RDMA over 10 Gb and 40 Gb Ethernet.

Intel MPI Fabrics

Intel MPI is able to detect the interconnect technologies available in the system and automatically use them. However, the programmer may control and, if necessary, switch to different fabrics using the environment variable `I_MPI_FABRICS`. The variable accepts one of the two syntaxes:

```
I_MPI_FABRICS=<fabric>
```

to specify one fabric for all communication paths and

```
I_MPI_FABRICS=<intra-node fabric>:<inter-node fabric>
```

to specify different fabrics for intra-node communication (i.e., messages between processes running on the same CPU or on the same coprocessor) and inter-node communication (i.e., messages between different CPUs, different coprocessors or a CPU and a coprocessor).

The following fabrics make sense with Intel Xeon Phi coprocessors:

tcp makes MPI operate over the TCP/IP stack, which is the protocol for Gigabit Ethernet (in inter-node communication) or TCP sockets (intra-node communication).

dapl uses the Direct Access Programming Library (DAPL) for RDMA communication over CCL using Mellanox InfiniBand interconnects and also in the special case of virtual interface `ib-scif` (see below).

tmp uses the Tag Matching Interface (TMI) for RDMA communication over PSM using Intel True Scale interconnects.

shm uses the shared-memory copy protocol (only available for intra-node communication).

We mentioned above that there is one special case where the fabric `dapl` is used. In a stand-alone machine (i.e., a system with no InfiniBand interconnects), a virtualized InfiniBand network may be configured for communication between the host CPU and the coprocessors (see Section 1.2.5). This creates a virtual interface `ib-scif`, which uses DAPL to communicate between the host and coprocessor(s).

Fabric Performance Benchmarks

To provide high-level guidance to on fabric selection in MPI applications for systems with Intel Xeon Phi coprocessors, we ran performance tests using the Intel MPI benchmark in the PingPong mode.

The PingPong mode of the Intel MPI benchmark sends an MPI message from rank 0 to rank 1 using `MPI_Send()`. Once rank 1 receives the message using `MPI_Recv()`, it replies with a message of the same size to rank 0. The time elapsed from the start of `MPI_Send()` to the completion of `MPI_Recv()` on rank 0 is the latency of PingPong. The benchmark application reports the latency and the bandwidth of communication. The latter is derived by dividing the message size by the latency and multiplying by 2 (to account for two messages: one sent and one received).

To execute the benchmark, the sequence of commands as shown in Listing 4.96 was used.

```
vega@lyra% # Enable MPI on coprocessors:
vega@lyra% export I_MPI_MIC=1
vega@lyra% # Intel MPI benchmark executables for CPU and MIC:
vega@lyra% export IMB_CPU=${I_MPI_ROOT}/bin64/IMB-MPI1
vega@lyra% export IMB_MIC=${I_MPI_ROOT}/mic/bin/IMB-MPI1
vega@lyra%
vega@lyra% # Set fabric to test:
vega@lyra% export I_MPI_FABRICS=tmi
vega@lyra%
vega@lyra% # Run communication benchmark between CPU and local MIC
vega@lyra% mpirun \
>         -host lyra      -np 1 ${IMB_CPU} PingPong -msglog 0:26 :\
>         -host lyra-mic0 -np 1 ${IMB_MIC}
```

Listing 4.96: Using the Intel MPI benchmark.

To vary the pairs of MPI communication end-points, the hosts given to `mpirun` may be changed (the name of the executable must change, too). To vary the communication fabrics, the environment variable `I_MPI_FABRICS` was set to values `dapl`, `shm`, `tcp` and `tmi`. The argument `-msglog` determines the range of tested message sizes.

First, we performed intra-device benchmarks. In these benchmarks, both MPI communication end-points were placed on the same host or on the same coprocessor. Note that with the fabric `dapl`, we had to specify another environment variable: `I_MPI_DAPL_PROVIDER=ofa-v2-scif0`. This instructed the Intel MPI runtime to communicate across the virtual InfiniBand interface `ib-scif` provided by CCL rather than across a physical interconnect. That is because `dapl` is the fabric for Mellanox branded interconnects, while we had Intel True Scale interconnects installed in our system. Results are shown in Listing 4.42.

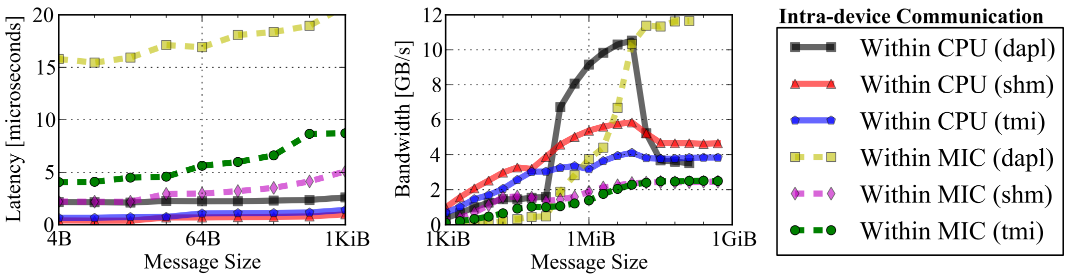


Figure 4.42: Intra-device MPI communication with `I_MPI_FABRICS=dapl`, `shm` and `tmi`.

Benchmarks in Figure 4.42 show that, depending on the message size, and depending on whether one wants to optimize bandwidth or latency, different fabrics must be chosen. While fabrics `shm` and `tmi` provide the best latency of small messages, `dapl` may yield far greater bandwidth for large messages (greater than 256 KiB).

In practice, when high bandwidth of intra-node communication is desired, and the system is based on Intel True Scale interconnects, the programmer may override the default choice of Intel MPI (`tmi`) and set the fabric `dapl`. Similarly, if Mellanox interconnects are installed, the default choice is `dapl`; if low latency is desired, it may be beneficial to override this setting and set the fabric to `shm`.

Note that this conclusion is only for intra-device communication.

The next set of benchmarks, shown in Listing 4.43, demonstrates inter-device and inter-node communication with fabric `tcp`. This fabric uses the Gigabit Ethernet network to communicate between systems. To communicate within a system between the host and coprocessor(s), it uses virtual network interfaces `mic0`, `mic1`, etc. provided by MPSS. Data sent to these interfaces is physically carried across the PCIe bus of the host system.

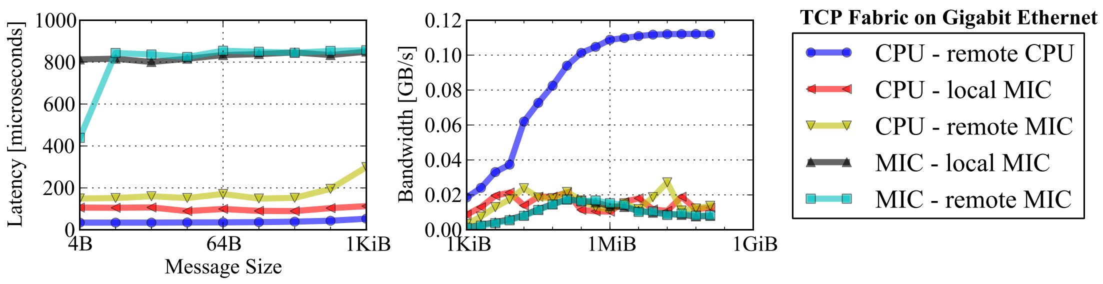


Figure 4.43: With `I_MPI_FABRICS=tcp`, data is transferred over a Gigabit Ethernet network.

Benchmarks of fabric `tcp` show that it is a very slow option for communication. The path "CPU – remote CPU" (messaging with a remote system across the physical Gigabit Ethernet network) indeed achieves Gigabit/s bandwidth and $\approx 30\ \mu$s latency for messages over 1 MiB. However, all paths using the virtual fabric have latencies in the hundreds of microseconds and bandwidth around 20 MB/s[1].

This result illustrates that the physical throughput of interconnects is unimportant if virtualized TCP/IP network is used on coprocessors. Therefore, TCP/IP networking with 10 Gigabit or 40 Gigabit interconnects is not useful if native applications for Intel Xeon Phi coprocessors are used.

This bandwidth limitation is a property of the TCP/IP software stack for the MIC architecture. It may be improved in future versions of MPSS. We also expect that in second generation MIC architecture coprocessors (see Section 1.4), with improved serial performance, the TCP/IP stack performance will be naturally improved. Of course, the socket version of the 2nd generation Intel Xeon Phi will, in all likelihood, have direct control of the on-board Ethernet adapters and should not be bottlenecked by this effect.

[1]Measurementes apply to external bridge configuration. With internal bridge or packet forwarding (see Section 1.2.5), bandwidth of large messages approaches 400 MB/s

In Figure 4.44 we show benchmarks of fabric `tmi`, which uses the Intel True Scale adapters to carry network traffic. We benchmark the same pairs of devices as with `tcp`.

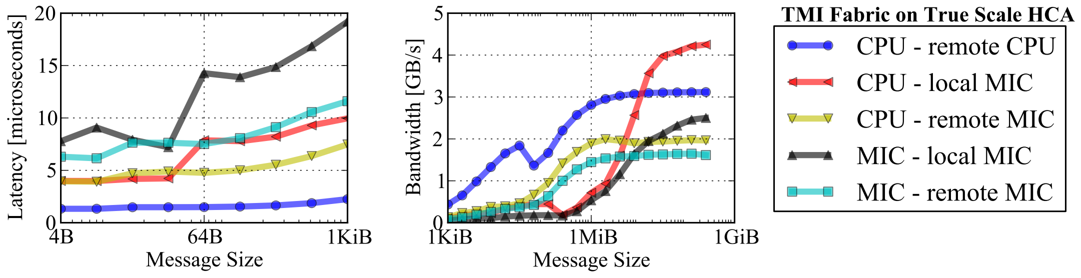


Figure 4.44: Benchmarks with `I_MPI_FABRICS=tmi`. Data is transferred by Intel True Scale interconnects and switches.

In terms of latency, the fastest path is communication between CPUs. It achieves sub-microsecond latencies for small messages. Latency of paths involving coprocessors is greater, but still under 10 μs for the smallest messages, which is orders of magnitude better than with `tcp`.

Bandwidth of large messages plateaus around 1 MiB message size (except CPU – local MIC and MIC – local MIC paths, which plateau at tens of MiB) and achieves around 3 GB/s for communication between CPUs, 2 GB/s for communication involving coprocessors, and over 4 GB/s for communication between the CPU and a local coprocessor.

Both the latency and the bandwidth of MPI messages are better between CPUs than between coprocessors. While this cannot be remedied by fabric control, it indicates that in applications with high peer-to-peer MPI communication traffic, it may be beneficial to use the MPI + offload approach (Figure 3.7) rather than the native application mode (Figure 3.6). See, e.g., [42] for a case study confirming this.

With four Intel Xeon Phi coprocessors in the system, two of them are sitting on PCIe slots controlled by CPU 1 and the other two on slots controlled by CPU 2. The Intel True Scale cards are also placed so that one is on CPU 1 and the other on CPU 2. Therefore, for communication between systems, the fabric library may choose the interconnect nearest (in terms of locality to the PCIe root complex) to them.

As of the version of Intel MPI available at the time of the writing of this book, we were not able to achieve a similar behavior with two Mellanox-branded interconnects. However, with a single Mellanox interconnect, we observed slightly higher bandwidth of large messages (see [10]). Also, Mellanox HCAs can use the fabric `dapl`, which makes for seamless interoperation with the virtual interface `ib-scif`.

To illustrate what happens when communication must take place across the Quick Path Interconnect (QPI) (a switch connecting the two CPU sockets in a two-way system), we performed an additional benchmark with fabric `dapl`. In this benchmark, MPI communication end-points were all within the same machine, and the virtual interface `ib-scif` was used. Results are shown in Figure 4.45.

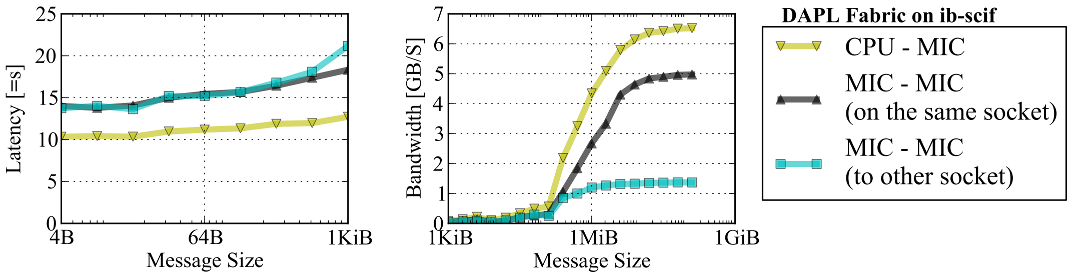


Figure 4.45: Benchmarks with `I_MPI_FABRICS=dapl` and `I_MPI_DAPL_PROVIDER=ofa-v2-scif0`. Data is transferred across the virtual InfiniBand interconnect `scif0`, physically carried by the local PCIe bus.

The path "MIC – MIC (on the same socket)" refers to communication between coprocessor 0 and coprocessor 1, which are both controlled by CPU 1. In the path "MIC – MIC (to other socket)", communication is between coprocessor 0 and coprocessor 2, of which the latter is controlled by CPU 2. As Figure 4.45 shows, the former path achieves 5 GB/s for large messages and the latter – only a quarter of that, 1.3 GB/s.

The benchmark in Figure 4.45 also shows the "CPU – MIC" path within a local system. It achieves more than 6 GB/s, which is close to the theoretical PCIe bandwidth and to the offload bandwidth benchmark (see Section 4.6.1). The programmer may take advantage of this fabric when the highest CPU to MIC bandwidth is desirable.

CHAPTER 5
Software Development Tools

5.1. Intel Math Kernel Library

Intel® Math Kernel Library (Intel MKL), first introduced to the public in 2003, is a collection of general-purpose mathematical functions. Core functionality of MKL includes Basic Linear Algebra Subprograms (BLAS), Linear Algebra Package (LAPACK), Scalable Linear Algebra Package (ScaLAPACK), sparse solvers, fast Fourier transform, and vector math. Implementations of Intel MKL functions are optimized for Intel Xeon processors, and a number of functions are also optimized for Intel Xeon Phi coprocessors. The scope of functions optimized for the MIC architecture is expected to grow with every new release of the library. Figure 5.1 illustrates the structure and applicability of Intel MKL.

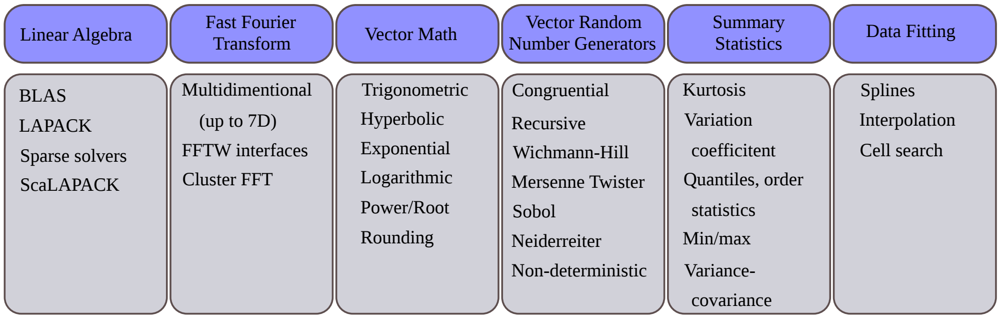


Figure 5.1: Intel MKL structure.

Earlier in our discussion we saw examples of workloads that use the Intel MKL (see Sections 4.2.5, 4.4.5 and 4.7.1). In this section, we outline the MKL usage models, provide general usage and optimization advice, and report benchmarks of some of the MKL functions. Complete documentation on the Intel MKL can be found in the Intel MKL Reference Manual [43]. We discuss the Intel MKL version 11.2 for Linux* OS.

5.1.1. Functions Offered by MKL

Intel MKL includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
  - Level 1 routines: vector operations (dot-product, scalar-vector product, rotation of points, etc.).
  - Level 2 routines: matrix-vector operations (matrix-vector product with general, band, Hermitian, symmetric matrices, etc.; solution of a linear system of equations with a triangular matrix).
  - Level 3 routines: matrix-matrix operations (matrix-matrix product of general, symmetric and Hermitian matrices, rank-k, rank-2k updates, etc.)
- Sparse BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices).
- LAPACK routines for solving systems of linear equations.
- LAPACK routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations.
- Auxiliary and utility LAPACK routines.
- ScaLAPACK computational, driver and auxiliary routines (only in Intel MKL for Linux* and Windows* operating systems).
- PBLAS routines for distributed vector, matrix-vector, and matrix-matrix operation.
- Sparse solver routines, direct and iterative. Includes
  - Parallel Direct Sparse Solver (PARDISO) interface,
  - PARDISO for clusters,
  - Direct Sparse Solver (DSS),
  - Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS),

  - Preconditioners based on incomplete LU factorization technique,
  - Sparse matrix checker routines.

- Extended eigensolver routines based on the FEAST Eigenvalue Solver 2.0 (SMP implementation only).

- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces). These functions offer functionality similar to that of SVML, except that VML is not tied to Intel compilers and designed to work better with large arrays. To use VML, the computation of vector functions must be done on arrays of arguments, rather than in loops with scalar semantics.

- Vector Statistical Library (VSL) functions for generating vectors of pseudorandom numbers with different types of statistical distributions and for performing convolution and correlation computations.

- General Fast Fourier Transform (FFT) Functions, providing fast computation of Discrete Fourier Transform via the FFT algorithms and having Fortran and C interfaces.

- Cluster FFT functions (only in Intel MKL for Linux* and Windows* operating systems).

- Tools for solving partial differential equations: trigonometric transform routines and Poisson solver.

- Optimization solver routines for solving nonlinear least squares problems through trust region algorithms and computing the Jacobi matrix by central differences.

- Basic Linear Algebra Communication Subprograms (BLACS) that are used to support a linear algebra oriented message passing interface.

- Data fitting functions for spline-based approximation of functions, derivatives and integrals of functions, and search.

5.1.2. Linking Applications with MKL. Link Line Advisor

MKL uses a layered model for linking, which gives it flexibility in access to parallelism in different environments. The layers of MKL are:

Interface layer matches compiled code with threading/computational layer. Used to control LP64/ILP64 interfaces (32-bit integer indices versus 64-bit indices), compatibility with compilers that return function values differently, and mapping between single precision and double precision names for applications using Cray*-style naming (SP2DP interface).

Threading layer provides a way to link threaded or sequential mode of the library with supported compilers (Intel, GNU* and PGI*).

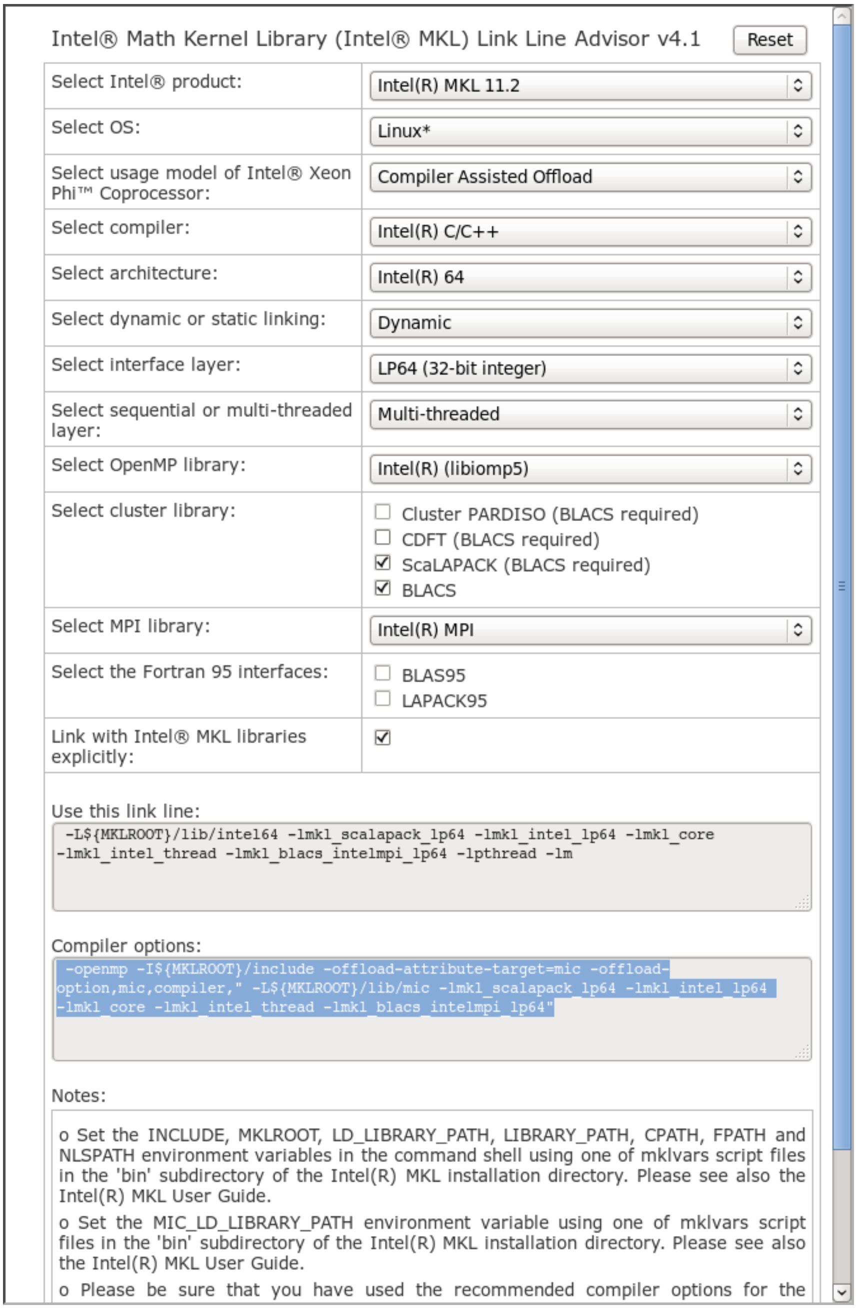
Computational layer accommodates multiple architectures by dispatching the function calls to the appropriate binary code.

Due to the layered model, compilation in some situations may be difficult. Generally, in order to compile applications using the Intel MKL with the Intel C++ Compiler, the command line argument `-mkl` must be specified, and MKL header files must be included in the source code in order to declare the functions and data types used in the application. However, when the application using Intel MKL is run in cluster environments, cross-compiled, or compiled with a non-Intel compiler, it may be difficult to determine the correct set of compiler arguments.

To assist users with this problem, the Intel MKL Link Line Advisor can be used [44]. The Advisor is an interactive Web page, which requests information about your system and on how you intend to use Intel MKL (link dynamically or statically; use threaded or serial mode; use of OpenMP, MPI, and other libraries). Using this information, the tool automatically generates the appropriate set of compiler and linker arguments.

Figure 5.2 illustrates the interface of the Intel MKL Link Line Advisor. The user provides the intended MKL usage mode via drop-down menus and checkbox lists, and the Advisor provides link line and compiler options which can be copied and pasted into the user's build system.

Note: The version of MKL requested by the compiler is not obvious when MKL is installed as a part of the Intel Parallel Studio XE suite. MKL version can be found by viewing the file `${MKLROOT}/include/mkl.h`.



Intel® Math Kernel Library (Intel® MKL) Link Line Advisor v4.1

Select Intel® product:	Intel(R) MKL 11.2
Select OS:	Linux*
Select usage model of Intel® Xeon Phi™ Coprocessor:	Compiler Assisted Offload
Select compiler:	Intel(R) C/C++
Select architecture:	Intel(R) 64
Select dynamic or static linking:	Dynamic
Select interface layer:	LP64 (32-bit integer)
Select sequential or multi-threaded layer:	Multi-threaded
Select OpenMP library:	Intel(R) (libiomp5)
Select cluster library:	☐ Cluster PARDISO (BLACS required) ☐ CDFT (BLACS required) ☑ ScaLAPACK (BLACS required) ☑ BLACS
Select MPI library:	Intel(R) MPI
Select the Fortran 95 interfaces:	☐ BLAS95 ☐ LAPACK95
Link with Intel® MKL libraries explicitly:	☑

Use this link line:

```
 -L${MKLROOT}/lib/intel64 -lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_core
-lmkl_intel_thread -lmkl_blacs_intelmpi_lp64 -lpthread -lm
```

Compiler options:

```
 -openmp -I${MKLROOT}/include -offload-attribute-target=mic -offload-
option,mic,compiler," -L${MKLROOT}/lib/mic -lmkl_scalapack_lp64 -lmkl_intel_lp64
-lmkl_core -lmkl_intel_thread -lmkl_blacs_intelmpi_lp64"
```

Notes:

o Set the INCLUDE, MKLROOT, LD_LIBRARY_PATH, LIBRARY_PATH, CPATH, FPATH and NLSPATH environment variables in the command shell using one of mklvars script files in the 'bin' subdirectory of the Intel(R) MKL installation directory. Please see also the Intel(R) MKL User Guide.

o Set the MIC_LD_LIBRARY_PATH environment variable using one of mklvars script files in the 'bin' subdirectory of the Intel(R) MKL installation directory. Please see also the Intel(R) MKL User Guide.

o Please be sure that you have used the recommended compiler options for the

Figure 5.2: Web interface of the Intel MKL Link Line Advisor.

5.1.3. MKL on Intel Xeon Phi Coprocessors

MKL supports computation on Intel Xeon Phi coprocessors in three modes of operation:

1. Automatic Offload (AO)

```
cblas_dgemm(..., ...);
```

```
icpc -mkl Code.cc
MKL_MIC_ENABLE=1
```

No code change is required to offload calculations to an Intel Xeon Phi coprocessor. The library takes care of data transfer and execution management.

2. Compiler Assisted Offload (CAO)

```
#pragma offload target(mic)
cblas_dgemm(..., ...);
```

```
icpc -mkl Code.cc
```

Programmer maintains explicit control of data transfer and remote execution, using compiler offload pragmas and directives. Can be used together with Automatic Offload.

3. Native Execution

```
cblas_dgemm(..., ...);
```

```
icpc -mkl -mmic Code.cc
```

Uses an Intel Xeon Phi coprocessor as an independent compute node. Data is initialized and processed on the coprocessor or communicated via MPI.

The operation modes discussed above enable *heterogeneous computing*, which takes advantage of both the multi-core host system and manycore Intel Xeon Phi coprocessors. For applications developed for CPUs and employing the Intel MKL, AO and native mode allow to use Intel Xeon Phi coprocessors without code modification. Using CAO in this case would require code modification but give the programmer fine control over the compute devices.

5.1.4. Automatic offload

For an application that already uses Intel MKL for calculations on the host system, the easiest way to launch calculations on an Intel Xeon Phi coprocessor is using the Automatic Offload (AO) mode. In order for AO to work, the application must be linked against a recent MKL version with support for the Intel Xeon Phi architecture. Nothing else needs to be done to use the coprocessor. The library will automatically detect available coprocessors, decide when it is beneficial to offload calculations to the coprocessor, transfer the data over the PCIe bus, and initiate offloaded computation on the coprocessor.

Applicability

The AO mode is supported only for a select subset of MKL functions. As of Intel MKL 11.2, the list of functions supporting automatic offload is:

1. ?GEMM (general matrix-matrix multiplication),
2. ?SYMM (matrix-matrix product, one input matrix is symmetric),
3. ?TRMM (matrix-matrix product, one input matrix is triangular),
4. ?TRSM (solution of a triangular matrix equation),
5. ?GETRF (LU factorization of a general $m \times n$ matrix),
6. ?POTRF (Cholesky factor.-n of a Hermitian positive-definite matrix),
7. ?GEQRF (QR factorization of a general $m \times n$ matrix),
8. ?SYRDB (reduction of a symmetric matrix to tridiagonal form), and
9. 1-dimensional FFT in batch mode.

For each of these, there is a size threshold at which offload may begin. Only when the problem size is above the threshold does automatic offload occur.

Automatic offload may be used in user applications written in C, C++ and Fortran, and also when MKL is used as a back-end for higher-level applications. For example, it is possible to use Intel Xeon Phi coprocessors in some workloads from R and MATLAB using the automatic offload feature of Intel MKL.

Enabling and Disabling AO

AO may be enabled either by setting an environment variable, or by calling the respective support function, as shown in Listing 5.1.

C/C++ function call	Set an environment variable

`mkl_mic_enable();`	`vega@lyra% export \`
	`> MKL_MIC_ENABLE=1`

Listing 5.1: Two ways to enable the Intel MKL Automatic Offload (AO).

To disable AO after it was previously enabled, use the corresponding support function call or environment variable, as shown in Listing 5.2.

```
mkl_mic_disable();
```

```
vega@lyra% export \
> MKL_MIC_ENABLE=0
```

Listing 5.2: Two ways to disable the Intel MKL Automatic Offload (AO).

In all of the above controls, the function call method overrides the environment variable setting.

Performance Tuning with AO

For offload tasks, MKL uses the same COI driver as user applications do. Therefore, environment variables used in the explicit offload programming model (see Section 2.2.8 and Section 2.2.9) also have effect in automatic offload. For instance, in systems with more than one coprocessor, it is possible to restrict MKL to using certain coprocessors using the environment variable `OFFLOAD_DEVICES`. Offload report can be obtained using `OFFLOAD_REPORT`.

MKL uses OpenMP for threading and therefore OpenMP-specific tuning controls may be used. For instance, variables `OMP_NUM_THREADS` (see Section 3.2.2) and `KMP_AFFINITY` (Section 4.4.5) have effect on the host and on the coprocessor. Reserved variables `MIC_OMP_NUM_THREADS` and `MIC_KMP_AFFINITY` may be used for tuning the offloaded part.

Other controls for automatic offload are discussed in the User's Guide pages on this subject.

Example

To demonstrate the capabilities of automatic offload in MKL, we will benchmark the application developed in Section 4.4.5 (code for this example is available as Lab 5.01 – see Section 6.2). This application calls the MKL implementation of DGEMM on square matrices, this time of size 16000×16000. There is no explicit offload in the code, i.e., it is unaware of Intel Xeon Phi coprocessors. However, it is compiled with an Intel compiler and linked against MKL.

First, we run and tune the application on the host CPU using the insight for thread affinity tuning from Section 4.4.5 as shown in Listing 5.3.

```
vega@lyra% export OMP_NUM_THREADS=24
vega@lyra% export KMP_AFFINITY=compact,1
vega@lyra% ./app-CPU 16000
Benchmarking DGEMM.
Problem size: 16000x16000 (6.144 GiB)
    Platform: CPU
     Threads: 24
    Affinity: compact,1

Trial     Time, s   Perf, GFLOP/s
    1  1.736e+01         471.78 *
    2  1.673e+01         489.79 *
    3  1.669e+01         490.97
    4  1.662e+01         492.85
    5  1.668e+01         491.25
    6  1.665e+01         491.99
    7  1.664e+01         492.33
    8  1.665e+01         492.16
    9  1.670e+01         490.68
   10  1.664e+01         492.45
-----------------------------------------------------
Average performance:         491.84 +- 0.72 GFLOP/s
-----------------------------------------------------
* - warm-up, not included in average
```

Listing 5.3: Running and tuning DGEMM on the host CPU (24-core two-way Intel Xeon processor with enabled hyper-threading).

For our reference, we also compiled this application for native execution on an Intel Xeon Phi coprocessor and ran it with $N = 16000$ and `KMP_AFFINITY=compact`. The result was a native performance of 992.1 ± 0.1 GFLOP/s.

In the next step (Listing 5.4), we attempt automatic offload to just one coprocessor. This attempt fails (we achieve lower performance than without automatic offload) because we have not tuned affinity on the coprocessor.

```
vega@lyra% export OFFLOAD_DEVICES=0 # Allow to use one coprocessor
vega@lyra% export MKL_MIC_ENABLE=1 # Enable automatic offload
vega@lyra% ./app-CPU 16000
...
Average performance:           438.94 +- 3.98 GFLOP/s
```

Listing 5.4: Attempting to run DGEMM with automatic offload.

Once we tune affinity on the coprocessor, automatic offload begins to show performance increase (Listing 5.5).

```
vega@lyra% export MIC_KMP_AFFINITY=compact # Tune affinity on copr.
vega@lyra% ./app-CPU 16000 # Using host and coprocessor number 0
...
Average performance:          1194.05 +- 5.78 GFLOP/s
```

Listing 5.5: Tuning DGEMM on the coprocessor in automatic offload.

The problem with $N = 16000$ is too small to fully exploit the parallelism available in our system. To appreciate the performance accessible with automatic offload we increase the problem size to the largest size that fits in the memory of the host: $N = 46000$. Result is shown in Listing 5.6.

```
vega@lyra% ./app-CPU 46000 # Large enough for host+coprocessor
...
Average performance:          1321.95 +- 1.68 GFLOP/s
```

Listing 5.6: Using a larger problem size for automatic offload.

At this point we can allow all coprocessors to take part in the calculation (Listing 5.7). Our benchmark system contains four coprocessor (see Section 4.1.4) and in order to use them all, we can either explicitly set `OFFLOAD_DEVICES`, or unset this variable (and export the unset value).

```
vega@lyra% export OFFLOAD_DEVICES=0,1,2,3 # Using host+4 coprocess.
vega@lyra% ./app-CPU 46000
...
Average performance:          3584.04 +- 9.26 GFLOP/s
```

Listing 5.7: Scaling DGEMM across four coprocessors.

Summary of these performance results is shown in Figure 5.3.

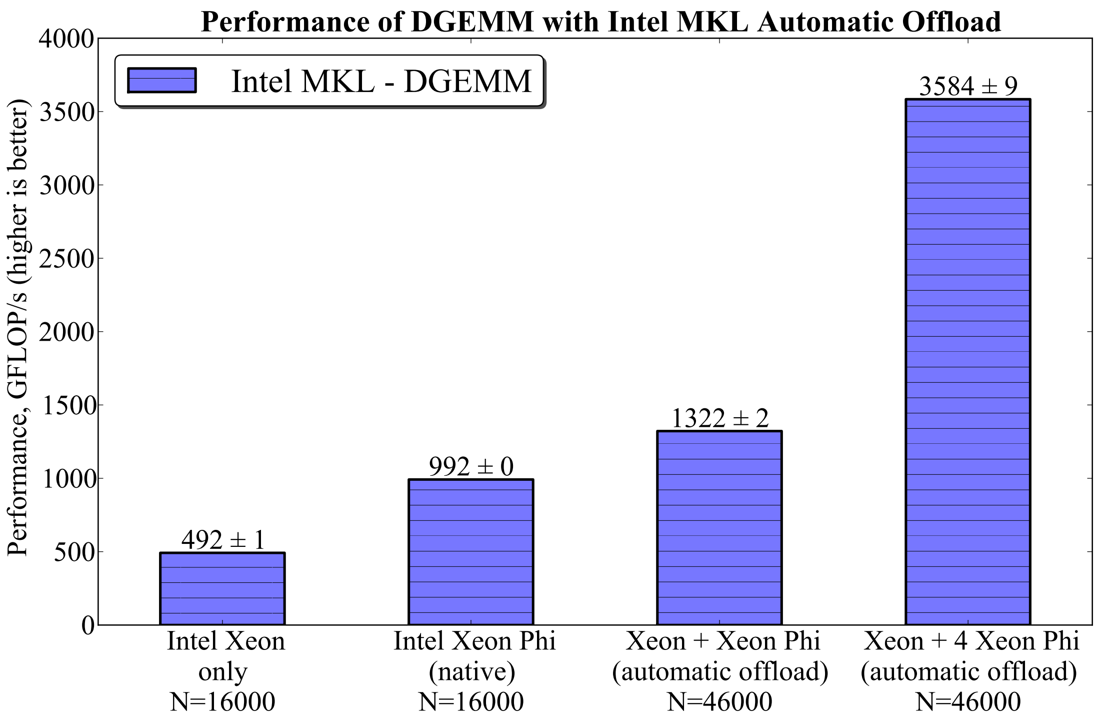


Figure 5.3: Performance of DGEMM in Intel MKL on the host, on the coprocessor, and in heterogeneous runs in the automatic offload mode.

The the four coprocessors teamed with the CPU are yielding a performance of 3586 GFLOP/s in double precision, which corresponds to efficiency of 80% compared to the sum of single-node performance of the CPU (492 GFLOP/s) and four coprocessors (992 GFLOP/s each). This efficiency is achieved using one line of code calling DGEMM and by setting several environment variables. The scaling is handled by the MKL library.

Finally, to see what is happening behind the scenes, we can use the offload report as shown in Listing 5.8. Note that when offload report is generated with MKL in the automatic offload mode, it includes information about work sharing between the host and the coprocessors. Tuning work sharing using `MKL_HOST_WORKDIVISION` may improve results, however, we leave it to the reader to test.

```
vega@lyra% export OFFLOAD_REPORT=2
vega@lyra% ./app-CPU 46000
...
Trial    Time, s   Perf, GFLOP/s
[MKL] [MIC --] [AO Function]         DGEMM
[MKL] [MIC --] [AO DGEMM Workdivision]  0.10 0.23 0.23 0.23 0.23
[MKL] [MIC 00] [AO DGEMM CPU Time]      59.277983 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time]      48.787105 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data] 20769920000 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data] 69154560000 bytes
[MKL] [MIC 01] [AO DGEMM CPU Time]      59.277983 seconds
[MKL] [MIC 01] [AO DGEMM MIC Time]      50.568903 seconds
[MKL] [MIC 01] [AO DGEMM CPU->MIC Data] 20769920000 bytes
[MKL] [MIC 01] [AO DGEMM MIC->CPU Data] 69154560000 bytes
[MKL] [MIC 02] [AO DGEMM CPU Time]      59.277983 seconds
[MKL] [MIC 02] [AO DGEMM MIC Time]      48.797464 seconds
[MKL] [MIC 02] [AO DGEMM CPU->MIC Data] 20769920000 bytes
[MKL] [MIC 02] [AO DGEMM MIC->CPU Data] 69154560000 bytes
[MKL] [MIC 03] [AO DGEMM CPU Time]      59.277983 seconds
[MKL] [MIC 03] [AO DGEMM MIC Time]      48.829253 seconds
[MKL] [MIC 03] [AO DGEMM CPU->MIC Data] 20769920000 bytes
[MKL] [MIC 03] [AO DGEMM MIC->CPU Data] 69154560000 bytes
    1  6.011e+01         3238.44 *
...
```

Listing 5.8: Obtaining report of automatic offload.

5.1.5. Compiler-Assisted Offload

It is possible to offload Intel MKL functions to the coprocessor using `#pragma offload`. This approach, known as Compiler Assisted Offload (CAO), requires that the user takes care of data transfer to the coprocessor.

The benefit of CAO is a more fine-grained control over data traffic and compute device usage than with AO. For instance, memory buffer retention, data persistence and overlapping computation with communication are possible in CAO (like, for example, in Section 4.6.2). With this techniques, CAO may produce better results than AO.

The disadvantage of CAO compared to AO is that automatic distribution of the offloaded functions across multiple coprocessors is not possible.

Listing 5.9 demonstrates calling the `DGEMM` routine using CAO (see Section 4.6.2 and Listing 4.80 for more information on this implementation).

```
#pragma offload target(mic:0)  signal(A_buff_calc)     \
    in(A_buff_calc: length(0) alloc_if(0) free_if(0)) \
    in(B_buff_calc: length(0) alloc_if(0) free_if(0)) \
    in(C_buff_calc: length(0) alloc_if(0) free_if(0))
    {
      cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
n, n, n, 1.0, A_buff_calc, n, B_buff_calc, n, 0.0, C_buff_calc, n);
    }
```

Listing 5.9: Example of the Compiler Assisted Offload usage mode of Intel MKL.

5.1.6. Native Execution

As discussed in Section 2.1, applications for native execution on Intel Xeon Phi coprocessors can be built with the compiler argument `-mmic`. To use Intel MKL in a native application, an additional argument `-mkl` is required. Native applications with Intel MKL functions operate just like native applications with user-defined functions. In MPI applications where MPI processes run on the host as well as on coprocessors, the code for the coprocessor part is compiled as a native application.

5.1.7. Benchmarks of Select MKL Functions

In this section we perform benchmarks of some of the Intel MKL functions optimized both for the multi-core and for the manycore architecture. In these benchmarks we measure the sustained performance of only performance-critical functions (i.e., ignore initialization and warm-up time). For more information about this methodology, see item 2 in Section 4.1.3.

Results are shown in Figure 5.4, and specifics of the benchmarks are discussed below.

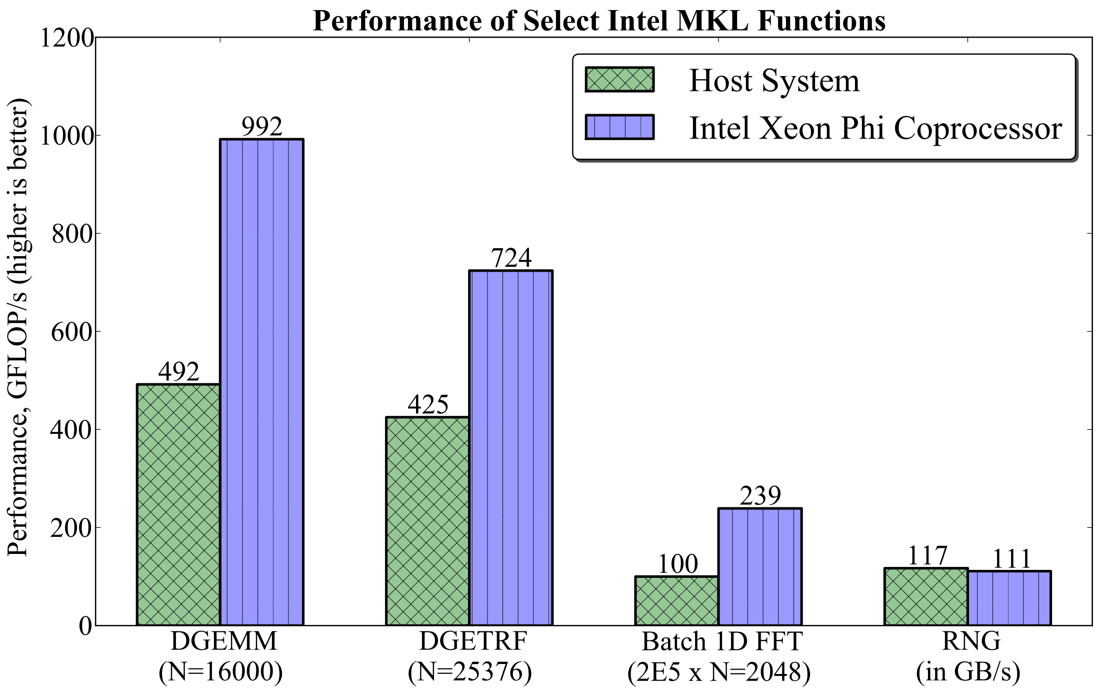


Figure 5.4: Benchmarks of select Intel MKL functions. See Section 5.1.7 for details.

All benchmarks of Intel Xeon Phi coprocessors are executed on a single coprocessor in the native mode. For all benchmarks, on the host, the affinity setting was `KMP_AFFINITY=compact,1,granularity=fine` and `OMP_NUM_THREADS=24`. On the coprocessor, we used the affinity setting `KMP_AFFINITY=compact`.

DGEMM

We benchmarked the multi-threaded implementation of DGEMM (a BLAS level 3 function for general matrix-matrix multiplication) in MKL using the application from Section 4.4.5. The benchmarked matrix size is $N = 16,000$ on the host and on the coprocessor in the native mode. The core of the DGEMM benchmark code is shown in Listing 5.5.

```
const double tStart = omp_get_wtime();
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, 1.0, A, n, B, n, 0.0, C, n);
const double tEnd = omp_get_wtime();
```

Figure 5.5: Core of the DGEMM benchmark.

Calculation time was translated to performance in GFLOP/s by assuming $2N^3$ FLOPs in every multiplication.

DGETRF

The structure of our benchmark of DGETRF (a LAPACK function for LU decomposition of general rectangular matrices) is similar to that of DGEMM, except that we timed the function `LAPACKE_dgetrf()`. We benchmarked matrix size $N = 25376$ on the host and on the coprocessor in the native mode. This size is the greatest multiple of 244×8 (product of the number of threads and the vector length) for which DGETRF may be run on a coprocessor with 16 GiB of memory. The core of the DGETRF benchmark code is shown in Listing 5.6.

```
const double tStart = omp_get_wtime();
LAPACKE_dgetrf(LAPACK_ROW_MAJOR, n, n, A, n, &ipiv[0]);
const double tEnd = omp_get_wtime();
```

Figure 5.6: Core of the DGETRF benchmark.

Calculation time was translated to performance in GFLOP/s by assuming $(2/3)N^3$ FLOPs in every decomposition.

1D FFT

We benchmarked 1-dimensional double precision complex-to-complex fast Fourier transforms (FFTs) in the batch mode. The batch size was set to $n = 2 \times 10^5$, and the size of each FFT in the batch to $m = 2048$. The core of the FFT benchmark code is shown in Listing 5.7.

```
DftiCreateDescriptor(&dh, DFTI_DOUBLE, DFTI_COMPLEX, 1, m);//m=2048
DftiSetValue(dh, DFTI_NUMBER_OF_TRANSFORMS, n); // n=200000
DftiSetValue(dh, DFTI_INPUT_DISTANCE, m);
DftiCommitDescriptor(dh);
const double tStart = omp_get_wtime();
DftiComputeForward(dh, A);
const double tEnd = omp_get_wtime();
```

Figure 5.7: Core of the FFT benchmark.

Calculation time was translated to performance in GFLOP/s by assuming $2m \log_2 m$ FLOPs in every transform.

Random Number Generator

MKL has multiple random bit stream generators, including multiplicative congruential, generalized feedback shift, combined multiple recursive, Mersenne Twister, Gray code-based, abstract and non-deterministic generators. We benchmark only the Mersenne Twister generator due to its popularity and high performance.

On top of the freedom of choice of the bit stream generator, there are various distribution generators, which can produce floating-point numbers with uniform, Gaussian, exponential, Laplace, and other distributions, or discrete numbers with uniform, Bernoulli, geometric, Poisson and many other distributions. We eliminate that latter degree of freedom (and the associated computation overhead) by generating simply a uniform bit stream.

The rate at which MKL can produce the bit stream is greater than the memory bandwidth of our system. Therefore, in order to exclude the bandwidth factor from our measurements, we tuned the random array size per core so that (i) it is large enough to reap the benefits of vectorization, but (ii) small enough to avoid spilling into the main memory. In practice, we had to set the benchmark to produce 2.4×10^5 words per thread on the host system and 10^4 words per thread on the coprocessor. Word in this context is 4 bytes.

The core of the Random Number Generator (RNG) benchmark code is shown in Listing 5.8.

```
  const long nT = omp_get_max_threads();
  unsigned int* A[nT];
  VSLStreamStatePtr rng[nT];

#pragma omp parallel
  { // Initialize RNGs in every thread
    const long thread = omp_get_thread_num();
    vslNewStream(&rng[thread], VSL_BRNG_MT2203 + thread, 0);
    A[thread]=(unsigned int*)_mm_malloc(n*sizeof(unsigned int),64);
    A[thread][0:n] = 0; // First touch, important on host
  }

  const double tStart = omp_get_wtime();
#pragma omp parallel
  { // Generate a random bit stream
    const long thread = omp_get_thread_num();
    viRngUniformBits(VSL_RNG_METHOD_UNIFORMBITS_STD, rng[thread],
                     n, A[thread]);
  }
  const double tEnd = omp_get_wtime();
```

Figure 5.8: Core of the RNG benchmark.

Calculation time was translated to performance in units of bandwidth (GB/s) by dividing the size of the generated bit stream by the elapsed calculation time.

5.2. Intel VTune Amplifier XE

Intel VTune Amplifier XE (or VTune for short) is a software tool for performance analysis of serial and multi-threaded applications for Intel processors and coprocessors. Performance is analyzed by means of collecting hardware event counts from the processor's PMU in Intel-manufactured CPUs and coprocessors.

The functionality offered by VTune includes:

1. Diagnosing overall performance metrics in an application, such as the cycles per instruction (CPI) ratio, thread concurrency, rate of cache misses, achieved memory bandwidth, occurrence of data page walks, amount of NUMA remote memory accesses, and other.
2. Detecting the hotspots, i.e., parts of the application that take the most time. The granularity of hotspot detection may be done at the level of individual functions or, if the amount of statistical data allows, at the level of individual lines of code or assembly instructions.

Because VTune relies on hardware-based event statistics, it can detect bottlenecks without significantly slowing down the execution of the application, which also results in realistic behavior of the analyzed application in terms of concurrency and memory traffic.

A convenient byproduct of the low-granularity hotspot detection is that VTune can display the assembly corresponding to a particular line or block of code. This is important for vector loops when the compiler implements multiple code paths. By just studying the assembly listing of the code, it is impossible to know which code path is taken most frequently at runtime. However, in VTune, it can be done easily because the actual execution time of all code components is measured.

The suite Intel Parallel Studio XE in Professional and Cluster editions includes Intel VTune Amplifier XE. At the same time, Intel VTune Amplifier XE may be obtained as a separate product.

In this section, we will walk through the workflow for application performance analysis in the GUI of Intel VTune Amplifier XE. Complete documentation for Intel VTune Amplifier XEis available in the Intel VTune Amplifier XE User's Guide.

5.2.1. System Administration

During the installation of VTune, the installer may perform all the system configuration steps necessary to use VTune. This includes compiling and running the driver and modifying MPSS and Linux account configuration.

If one of the subsequent system configuration steps (e.g., installation or upgrade of MPSS or Linux kernel) breaks the installation of VTune, the administrator may uninstall and re-install the product to restore functionality. Although VTune comes with scripts which allow less invasive recovery procedures (see, e.g., the section Installing Drivers in the Intel VTune Amplifier User's Guide [45]), re-installation is the easiest practical method, which typically only takes a few minutes.

To verify VTune functionality, we can ensure that the kernel module `sep` is loaded. To analyze coprocessor applications, a similar module must be running in the OS of the coprocessor, and the product must be installed at `/amplxe` in the coprocessor filesystem. Listing 5.10 illustrates checking for proper VTune installation.

```
vega@lyra% # User must belong to group vtune
vega@lyra% groups vega
vega : vega vtune
vega@lyra%
vega@lyra% # Checking kernel module on host
vega@lyra% lsmod | grep sep
sep3_15               526926  0
vega@lyra%
vega@lyra% # Checking kernel module on coprocessor
vega@lyra% ssh mic0 lsmod | grep sep
sep3_15                45337  2
vega@lyra%
vega@lyra% # Checking product installation on coprocessor
vega@lyra% ssh mic0 ls /amplxe
vtune_amplifier_xe_2015.1.1.380310
```

Listing 5.10: Verifying proper installation of VTune on the host and on the coprocessor.

The system administrator should also know that in order to use VTune, the user of a Linux system must belong to a Linux group assigned to the sampling driver (by default, called `vtune`). This group is created during the installation.

5.2.2. Running VTune

Prior to running VTune, the environment variables required for it must be set. This can be done either by sourcing the dedicated script as shown in Listing 5.11, or by sourcing the corresponding script for the whole Parallel Studio XE suite. The administrator may implement this automatically upon login using standard methods for the operating system at hand.

```
vega@lyra% source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
Copyright (C) 2009-2014 Intel Corporation. All rights reserved.
Intel(R) VTune(TM) Amplifier XE 2015 (build 380310)
```

Listing 5.11: Setting environment variables for VTune.

Let's begin our tour of VTune by launching the GUI version of the tool from a terminal using the command `amplxe-gui`. The window shown in Figure 5.9 is the home screen of VTune. At any time we can return to this screen by clicking the toolbar button .

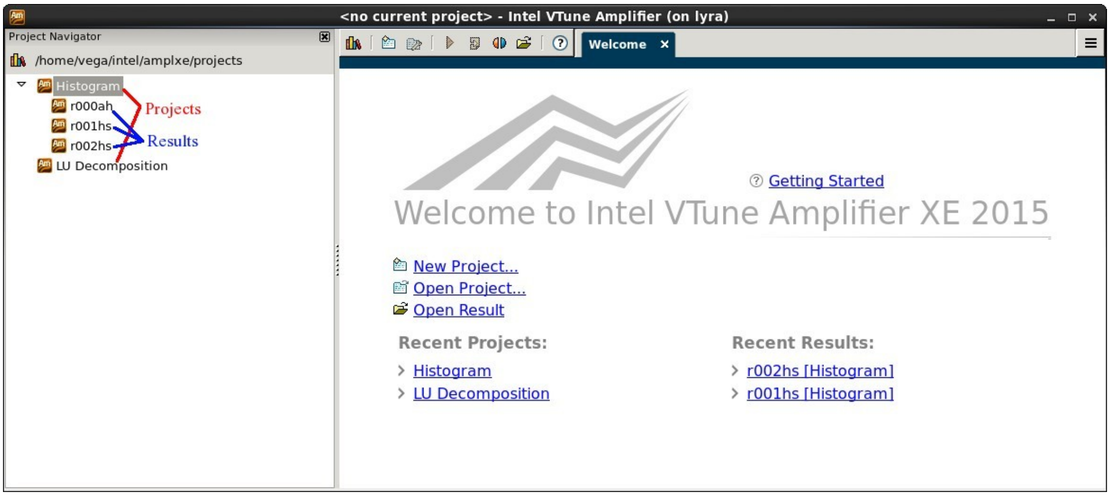


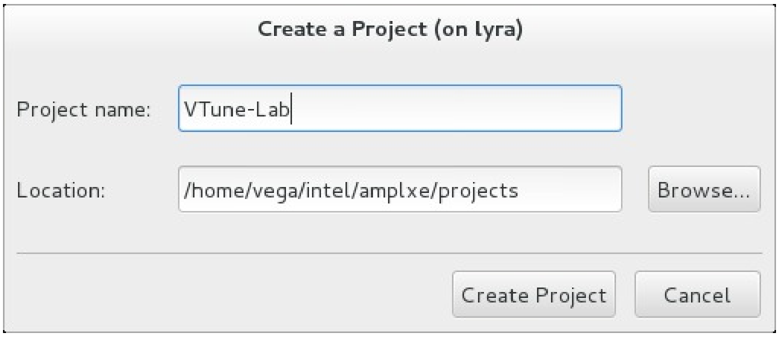
Figure 5.9: Welcome screen of Intel VTune Amplifier XE.

The GUI may be used to analyze applications on a local machine and to view the results. Besides the graphical interface, VTune supports a command line interface invoked by command `amplxe-gui`. The command line tool tool may be used to collect analysis results on a remote machine (or a machine without graphical shell installed). These results may then be copied to the developer's workstation and viewed in the GUI tool.

It is possible to use the GUI of VTune remotely using X11 forwarding (i.e., logging in with `ssh -X`) or via a remote desktop application. This works as long as the target system has the graphical desktop installed, and the bandwidth of connection is sufficient for this to work.

5.2.3. Project Management

We will use code available in Lab 5.02 (see Section 6.2) for our tor of VTune. Let's create a new project by clicking the button in the toolbar and call it "VTune-Lab". The window in Figure 5.10 shows the dialogue for project creation and placement.



Project name: VTune-Lab

Location: /home/vega/intel/amplxe/projects

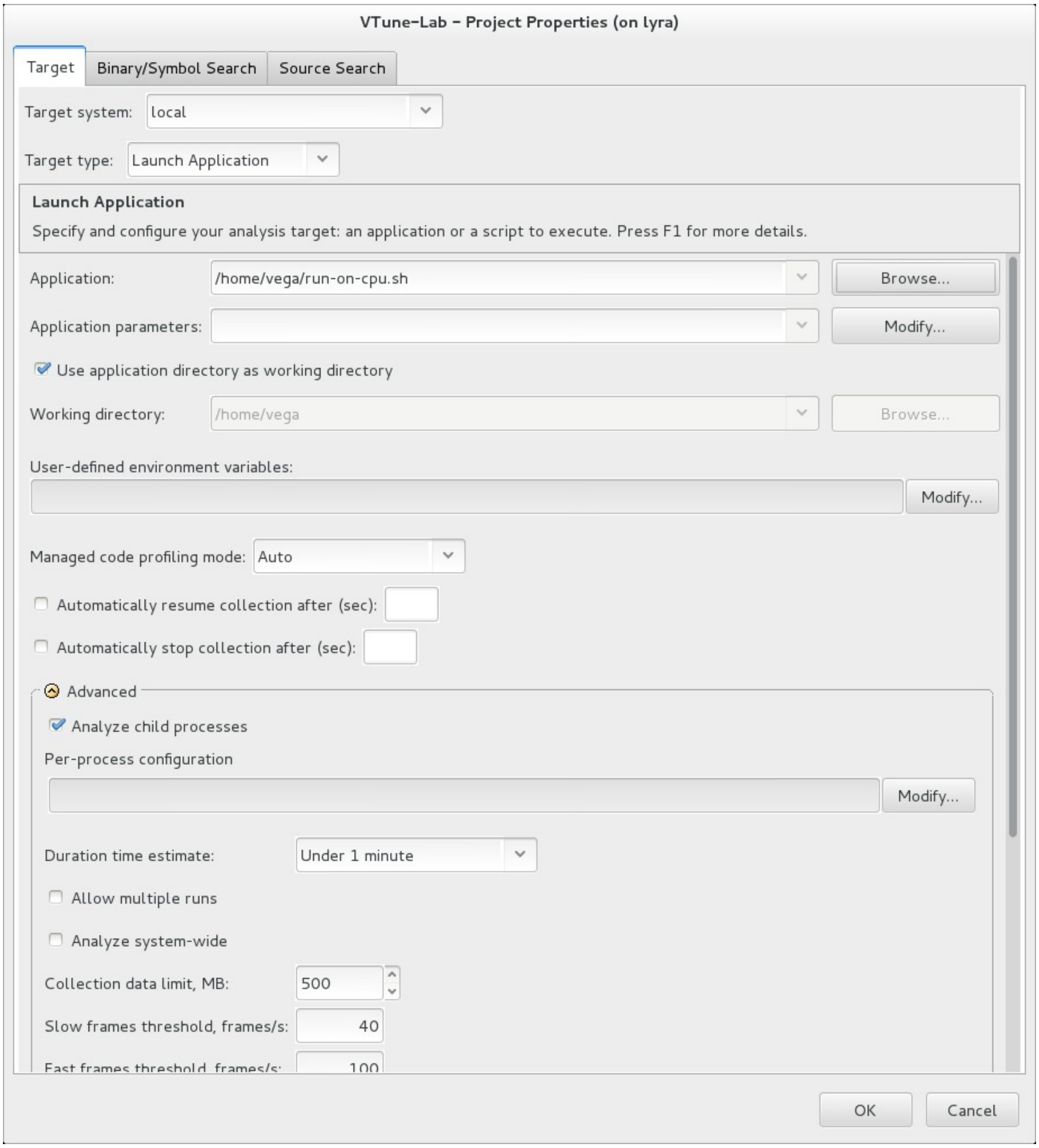
Figure 5.10: Creating a project.

VTune stores configurations and products of analysis as entities called *projects*. Projects are stored in home directories of VTune users at the path `~/intel/amplxe`. List of projects is shown in the sidebar in the left-hand side of the VTune window. Inside of every project, there may be multiple *results*, which are different instances of application analysis. Results show in the sidebar as entries under projects.

It is possible to create or remove projects using the context menu in the sidebar (right mouse click in that area brings up the context menu). New results inside of projects are produced by clicking the button in the toolbar. Results may be managed (deleted, renamed) by right-clicking their names in the sidebar menu.

5.2.4. Analysis on the Host CPU

The window following project creation is shown in Figure 5.11.


VTune-Lab – Project Properties (on lyra)

Target | Binary/Symbol Search | Source Search

Target system: local

Target type: Launch Application

Launch Application

Specify and configure your analysis target: an application or a script to execute. Press F1 for more details.

Application: /home/vega/run-on-cpu.sh [Browse...]

Application parameters: [Modify...]

[✓] Use application directory as working directory

Working directory: /home/vega [Browse...]

User-defined environment variables: [Modify...]

Managed code profiling mode: Auto

[ ] Automatically resume collection after (sec):

[ ] Automatically stop collection after (sec):

Advanced

[✓] Analyze child processes

Per-process configuration [Modify...]

Duration time estimate: Under 1 minute

[ ] Allow multiple runs

[ ] Analyze system-wide

Collection data limit, MB: 500

Slow frames threshold, frames/s: 40

Fast frames threshold, frames/s: 100

[OK] [Cancel]

Figure 5.11: Project properties for a host application.

Here we specify the application to run and parameters, if any. We can return to this window at a later time by clicking the toolbar button ▶.

The first line "Target system" has a drop-down menu, which is set to "local". This will run performance analysis on the host. We could change it to other values to analyze performance on the coprocessor (see Section 5.2.5).

For the application to study we will take the workload from Section 4.4.3. This application solves multiple systems of linear algebraic equations with the iterative Jacobi method. We will take the first implementation, which has load imbalance across threads.

In Figure 5.11, we specify in the line "Application:" a script that executes our application. We find it more convenient to specify a wrapper script rather than directly specifying the executable. The wrapper script can perform additional tasks, such as setting environment variables, staging input data and managing output data. The script that we used to launch our application on the host is shown in Listing 5.12.

```bash
#!/bin/bash

source /opt/intel/parallel_studio_xe_2015/psxevars.sh
export OMP_SCHEDULE=static

./app-CPU
```

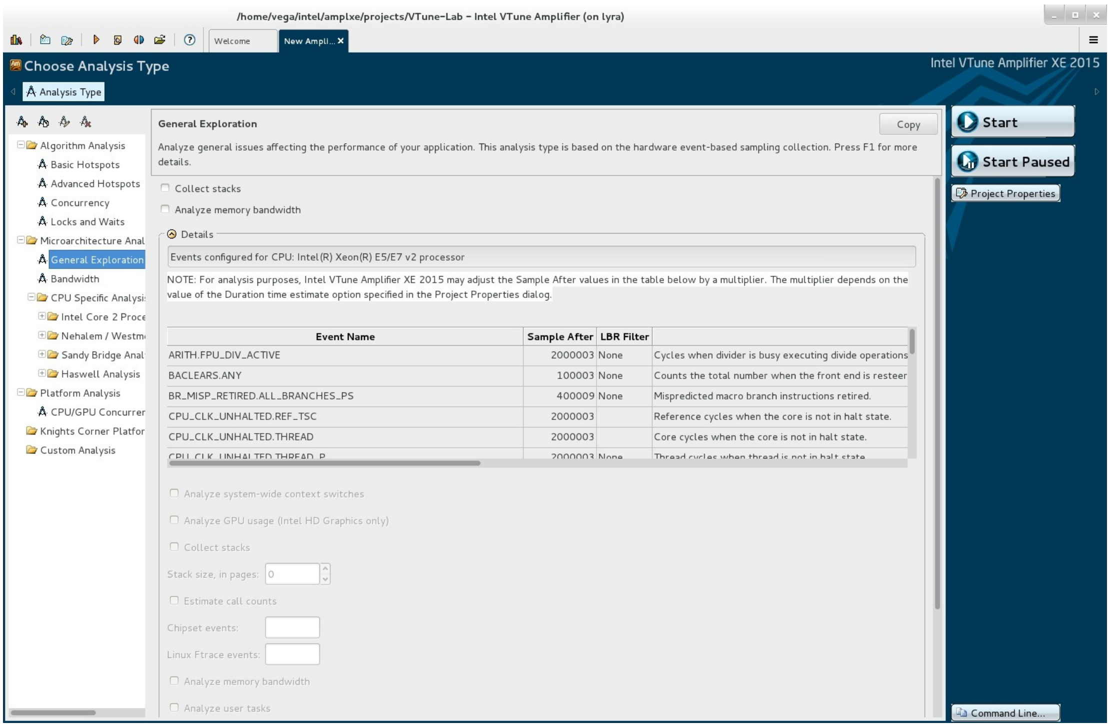
Listing 5.12: Script `run-on-cpu.sh` launching the analyzed application from VTune.

It is important that the executable of the analyzed application (in our case, `app-CPU`) should be compiled with arguments `-g` and `-O3`. The former includes debugging symbols in the executable, which allows one to see function names VTune. The latter optimizes the application in a way that the presence of debugging symbols does not significantly slow down the application.

Other arguments that we could specify during project creation (Figure 5.11) include application parameters, working directory and environment variables (our script already takes care of that), the time slice to be analyzed (to exclude initialization and tear-down) and application duration (determines the amount of saved performance data).

After configuring the project, we are taken to a window titled "New

Analysis" (Figure 5.12). This window allows us to choose the type of analysis and click the button "start" to launch application and begin data collection. We find that a good place to start analysis is "General Exploration".



General Exploration

Analyze general issues affecting the performance of your application. This analysis type is based on the hardware event-based sampling collection. Press F1 for more details.

☐ Collect stacks

☐ Analyze memory bandwidth

Details

Events configured for CPU: Intel(R) Xeon(R) E5/E7 v2 processor

NOTE: For analysis purposes, Intel VTune Amplifier XE 2015 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Project Properties dialog.

Event Name	Sample After	LBR Filter	
ARITH.FPU_DIV_ACTIVE	2000003	None	Cycles when divider is busy executing divide operations
BACLEARS.ANY	100003	None	Counts the total number when the front end is resteer
BR_MISP_RETIRED.ALL_BRANCHES_PS	400009	None	Mispredicted macro branch instructions retired.
CPU_CLK_UNHALTED.REF_TSC	2000003		Reference cycles when the core is not in halt state.
CPU_CLK_UNHALTED.THREAD	2000003		Core cycles when the core is not in halt state.
CPU_CLK_UNHALTED.THREAD_P	2000003	None	Thread cycles when thread is not in halt state.

☐ Analyze system-wide context switches

☐ Analyze GPU usage (Intel HD Graphics only)

☐ Collect stacks

Stack size, in pages: 0

☐ Estimate call counts

Chipset events:

Linux Ftrace events:

☐ Analyze memory bandwidth

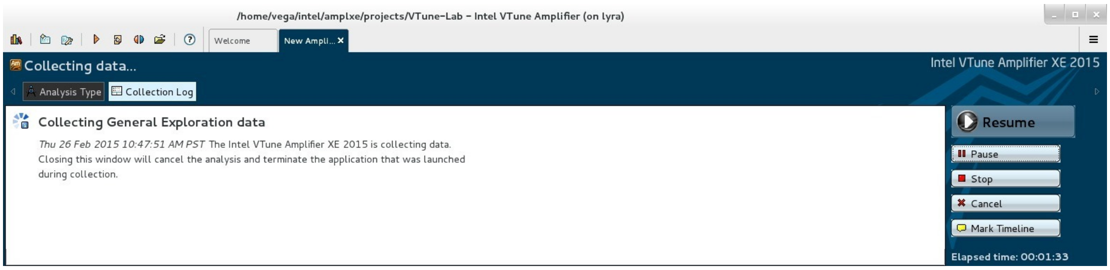
☐ Analyze user tasks

Figure 5.12: Starting a General Exploration analysis.

Note the button Command Line... in the bottom right corner. Clicking this button brings up a window that contains a command line that we could use to run the same kind of analysis using the command line tool `amplxe-cl`. This would be useful if we were running analysis on a remote computer without graphical packages such as a compute node in a cluster.

We click the button Start, and the General Exploration analysis starts. While analysis is running, the window in VTune looks like in Figure 5.13.

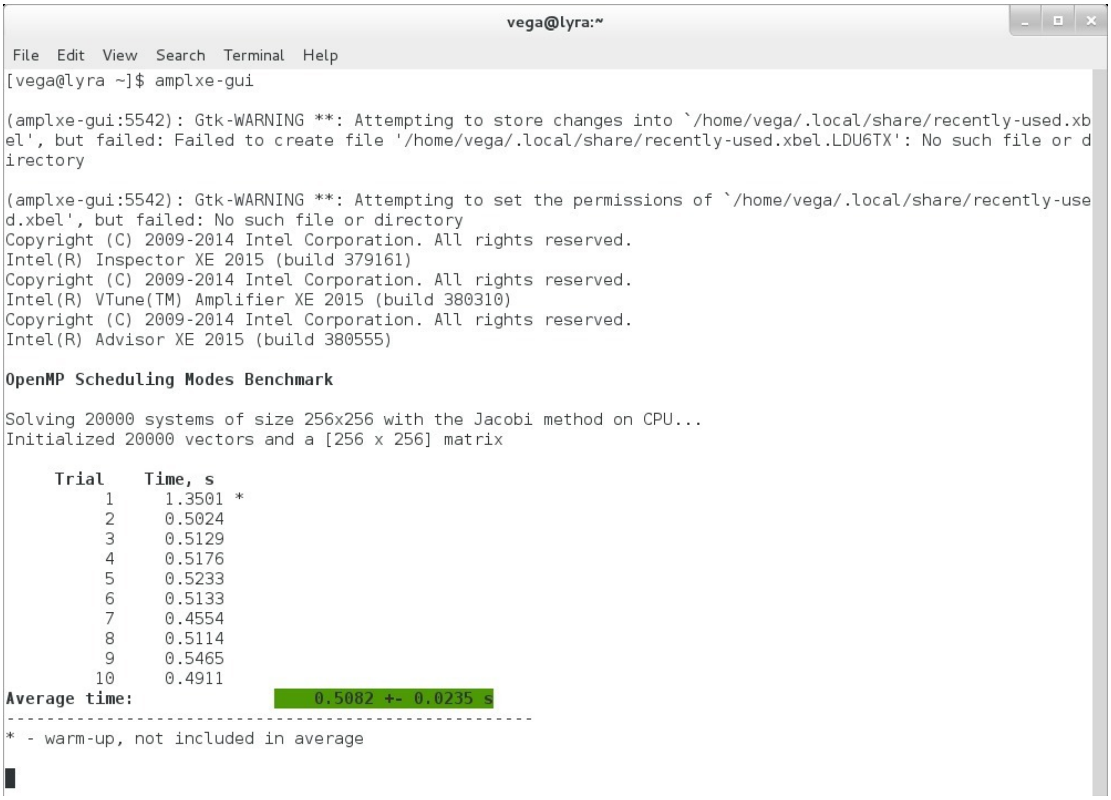
At this point, we can switch back to the terminal from which VTune was launched, and there we will see the running output of our application or error messages, if any (see Figure 5.14).



Collecting General Exploration data

Thu 26 Feb 2015 10:47:51 AM PST The Intel VTune Amplifier XE 2015 is collecting data.
Closing this window will cancel the analysis and terminate the application that was launched during collection.

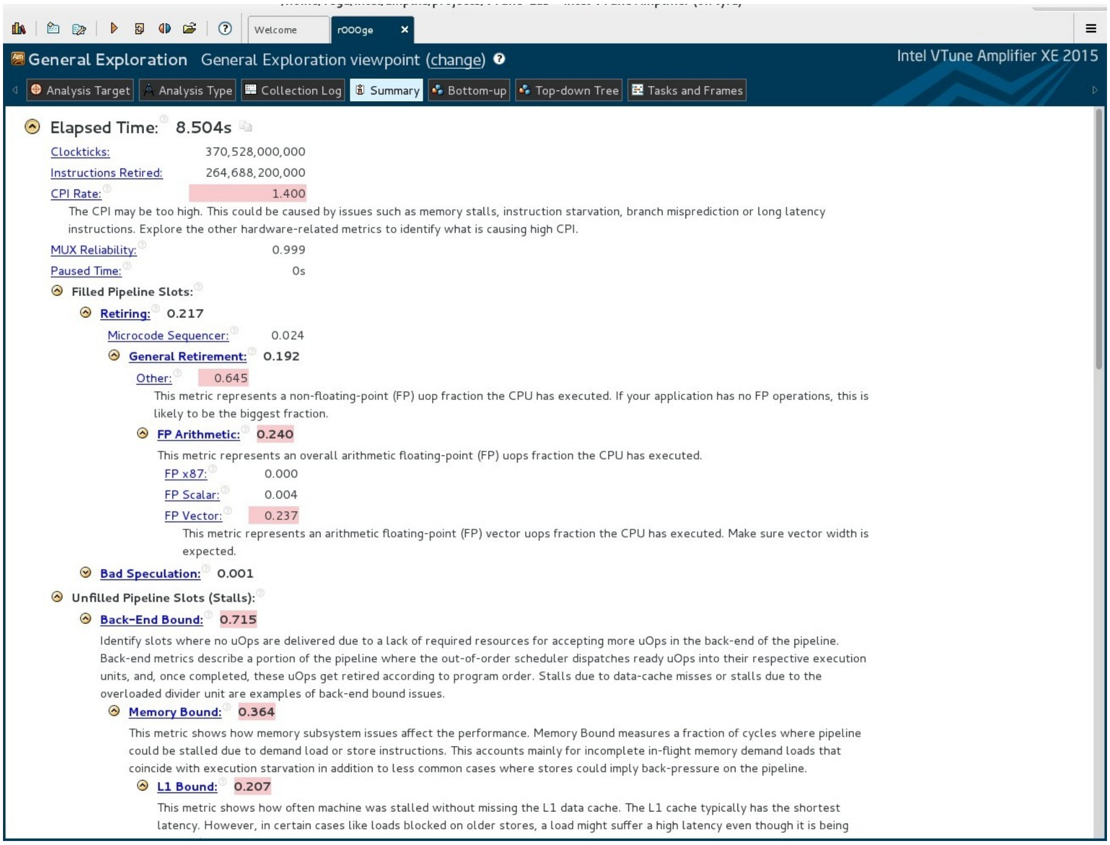
Figure 5.13: Waiting for analysis results.



```
[vega@lyra ~]$ amplxe-gui

(amplxe-gui:5542): Gtk-WARNING **: Attempting to store changes into `/home/vega/.local/share/recently-used.xb
el', but failed: Failed to create file '/home/vega/.local/share/recently-used.xbel.LDU6TX': No such file or d
irectory

(amplxe-gui:5542): Gtk-WARNING **: Attempting to set the permissions of `/home/vega/.local/share/recently-use
d.xbel', but failed: No such file or directory
Copyright (C) 2009-2014 Intel Corporation. All rights reserved.
Intel(R) Inspector XE 2015 (build 379161)
Copyright (C) 2009-2014 Intel Corporation. All rights reserved.
Intel(R) VTune(TM) Amplifier XE 2015 (build 380310)
Copyright (C) 2009-2014 Intel Corporation. All rights reserved.
Intel(R) Advisor XE 2015 (build 380555)

OpenMP Scheduling Modes Benchmark

Solving 20000 systems of size 256x256 with the Jacobi method on CPU...
Initialized 20000 vectors and a [256 x 256] matrix

     Trial    Time, s
         1      1.3501 *
         2      0.5024
         3      0.5129
         4      0.5176
         5      0.5233
         6      0.5133
         7      0.4554
         8      0.5114
         9      0.5465
        10      0.4911
Average time:               0.5082 +- 0.0235 s
-----------------------------------------------------
* - warm-up, not included in average

```

Figure 5.14: Output of application is visible in the terminal from which VTune was launched.

Finally, analysis is finished, and in a minute we see the window shown in Figure 5.15. This is the Summary pane of the General Exploration analysis.



Elapsed Time: 8.504s
Clockticks: 370,528,000,000
Instructions Retired: 264,688,200,000
CPI Rate: 1.400
The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.
MUX Reliability: 0.999
Paused Time: 0s
Filled Pipeline Slots:
Retiring: 0.217
Microcode Sequencer: 0.024
General Retirement: 0.192
Other: 0.645
This metric represents a non-floating-point (FP) uop fraction the CPU has executed. If your application has no FP operations, this is likely to be the biggest fraction.
FP Arithmetic: 0.240
This metric represents an overall arithmetic floating-point (FP) uops fraction the CPU has executed.
FP x87: 0.000
FP Scalar: 0.004
FP Vector: 0.237
This metric represents an arithmetic floating-point (FP) vector uops fraction the CPU has executed. Make sure vector width is expected.
Bad Speculation: 0.001
Unfilled Pipeline Slots (Stalls):
Back-End Bound: 0.715
Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the back-end of the pipeline. Back-end metrics describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to program order. Stalls due to data-cache misses or stalls due to the overloaded divider unit are examples of back-end bound issues.
Memory Bound: 0.364
This metric shows how memory subsystem issues affect the performance. Memory Bound measures a fraction of cycles where pipeline could be stalled due to demand load or store instructions. This accounts mainly for incomplete in-flight memory demand loads that coincide with execution starvation in addition to less common cases where stores could imply back-pressure on the pipeline.
L1 Bound: 0.207
This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being

Figure 5.15: Summary screen of the General Exploration analysis.

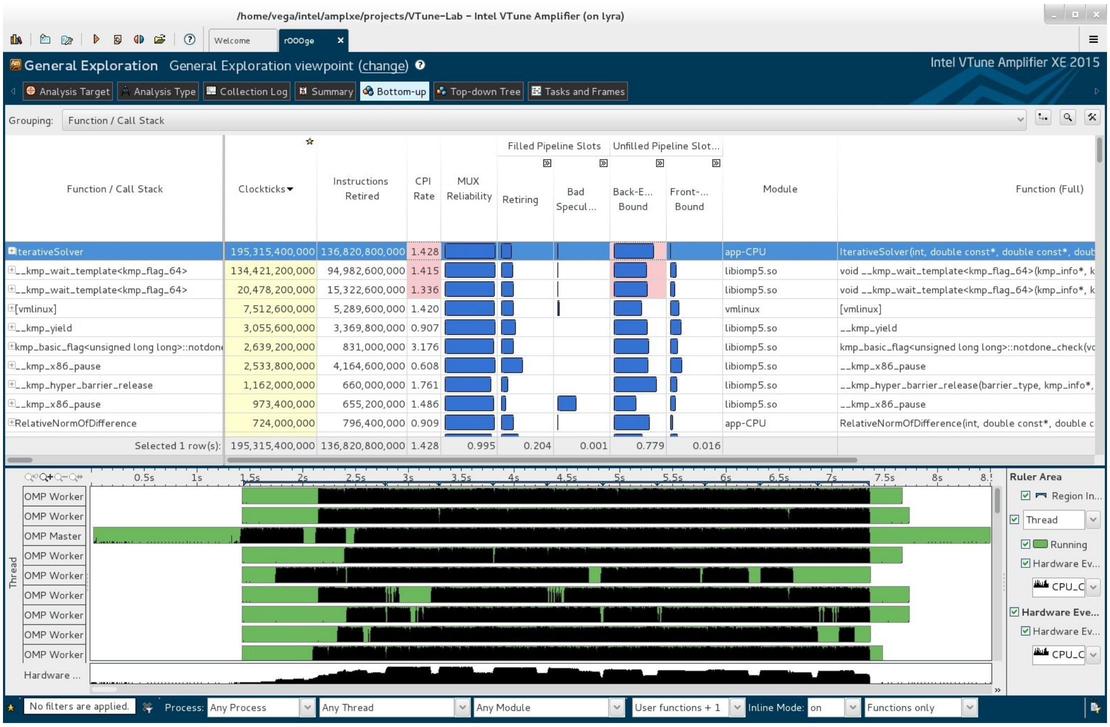
The summary pane shows general application analysis metrics. Metrics that look sub-optimal VTune automatically highlights. We can mouse over any of the metrics and read more information about it in a pop-up window.

One of the most important metrics here is the CPI rate. The lower the CPI, the less is the latency impact in the calculation. On Intel Xeon processors, arithmetically intensive applications may achieve a CPI as low as 0.5.

The impact of latency is further broken down in the group "Unfilled Pipeline Slots". The group preceding it, "Filled Pipeline Slots" lists information about microcode that was executed in "useful" CPU cycles.

We can re-visit the Summary pane at a later time by clicking the button Summary in the menu.

Another pane, brought up by clicking Bottom-up, shows the same metrics as the Summary pane, but this time listed by functions/call stacks. We click this button and see "Bottom-up" pane shown in Figure 5.16.


/home/vega/intel/amplxe/projects/VTune-Lab - Intel VTune Amplifier (on lyra)

General Exploration General Exploration viewpoint (change)

Intel VTune Amplifier XE 2015

Grouping: Function / Call Stack

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	MUX Reliability	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Specul...	Unfilled Pipeline Slot...: Back-E... Bound	Unfilled Pipeline Slot...: Front-... Bound	Module	Function (Full)
IterativeSolver	195,315,400,000	136,820,800,000	1.428						app-CPU	IterativeSolver(int, double const*, double const*, doub
__kmp_wait_template<kmp_flag_64>	134,421,200,000	94,982,600,000	1.415						libiomp5.so	void __kmp_wait_template<kmp_flag_64>(kmp_info*, k
__kmp_wait_template<kmp_flag_64>	20,478,200,000	15,322,600,000	1.336						libiomp5.so	void __kmp_wait_template<kmp_flag_64>(kmp_info*, k
[vmlinux]	7,512,600,000	5,289,600,000	1.420						vmlinux	[vmlinux]
__kmp_yield	3,055,600,000	3,369,800,000	0.907						libiomp5.so	__kmp_yield
kmp_basic_flag<unsigned long long>::notdone	2,639,200,000	831,000,000	3.176						libiomp5.so	kmp_basic_flag<unsigned long long>::notdone_check(vc
__kmp_x86_pause	2,533,800,000	4,164,600,000	0.608						libiomp5.so	__kmp_x86_pause
__kmp_hyper_barrier_release	1,162,000,000	660,000,000	1.761						libiomp5.so	__kmp_hyper_barrier_release(barrier_type, kmp_info*,
__kmp_x86_pause	973,400,000	655,200,000	1.486						libiomp5.so	__kmp_x86_pause
RelativeNormOfDifference	724,000,000	796,400,000	0.909						app-CPU	RelativeNormOfDifference(int, double const*, double c
Selected 1 row(s):	195,315,400,000	136,820,800,000	1.428	0.995	0.204	0.001	0.779	0.016		

No filters are applied. Process: Any Process | Any Thread | Any Module | User functions + 1 | Inline Mode: on | Functions only

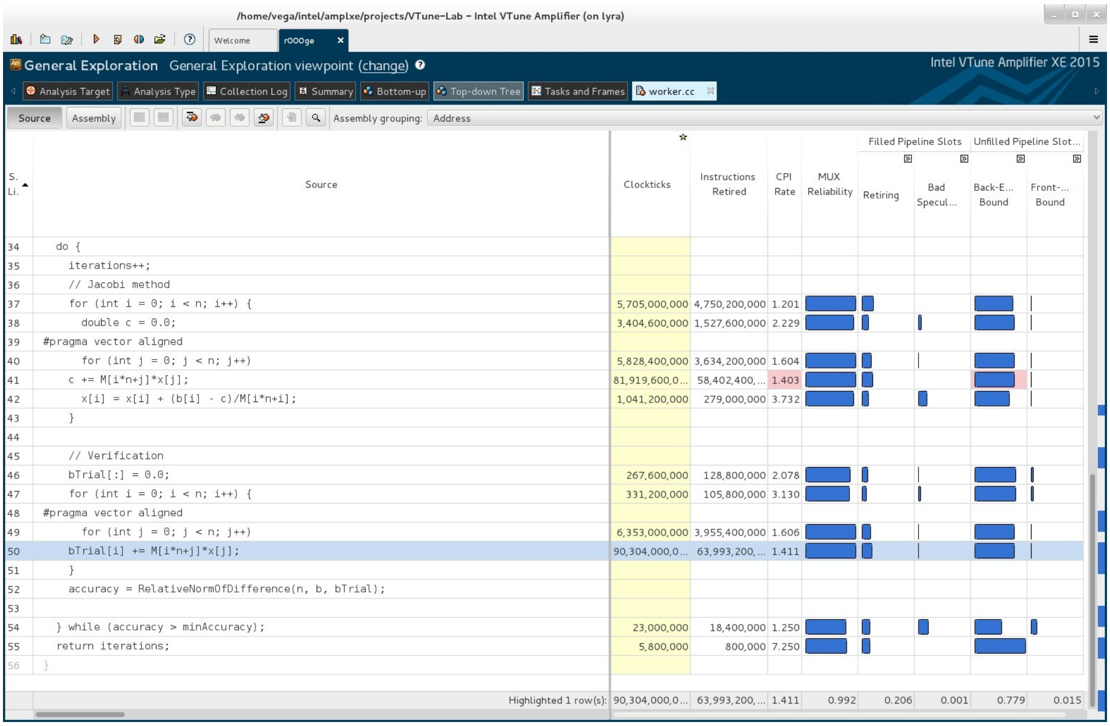
Figure 5.16: Bottom-up pane in General Exploration viewpoint.

The functions are now sorted by Clockticks. The order of sorting may be changed by clicking the column headers. The top consumer of Clockticks is the function `IterativeSolver`, which belongs to our application. It is important that we compiled our application with `-g`; otherwise, the name of this function would not be visible. The bottom-up view is a good place to start code analysis, especially for large codes as it immediately identifies the parts of the code promising good return on investment in optimization.

A similar pane, brought up by the button Top-down Tree (we do not have a screenshot for it) provides similar information, but also detailing the hierarchy of function calls.

At the bottom of the bottom-up window, the green and black stripes show the timelines of different events collected by VTune. We will re-visit this timeline later in Figure 5.22, where it will contain information more useful for this particular application.

Now let's double click with the mouse the name of our top hotspot, function `IterativeSolver`. What appears is the code listing for this function (Figure 5.17). In front of every line, there is a table detailing the number of various performance metrics (clockticks, retired instructions, etc.) collected for this line. This demonstrates low-granularity performance analysis in VTune made possible by dedicated PMUs in hardware.



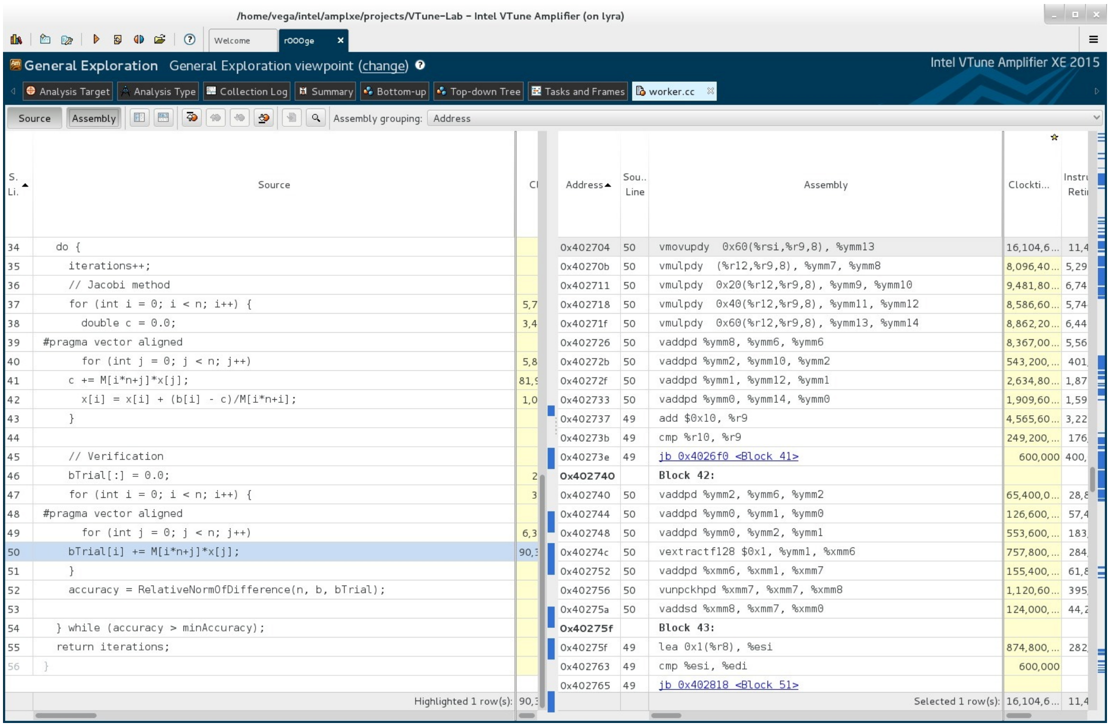
S. Li.	Source	Clockticks	Instructions Retired	CPI Rate	MUX Reliability	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Specul...	Unfilled Pipeline Slot...: Back-E... Bound	Unfilled Pipeline Slot...: Front-... Bound
34	`do {`								
35	`iterations++;`								
36	`// Jacobi method`								
37	`for (int i = 0; i < n; i++) {`	5,705,000,000	4,750,200,000	1.201					
38	`double c = 0.0;`	3,404,600,000	1,527,600,000	2.229					
39	`#pragma vector aligned`								
40	`for (int j = 0; j < n; j++)`	5,828,400,000	3,634,200,000	1.604					
41	`c += M[i*n+j]*x[j];`	81,919,600,0...	58,402,400,...	1.403					
42	`x[i] = x[i] + (b[i] - c)/M[i*n+i];`	1,041,200,000	279,000,000	3.732					
43	`}`								
44									
45	`// Verification`								
46	`bTrial[:] = 0.0;`	267,600,000	128,800,000	2.078					
47	`for (int i = 0; i < n; i++) {`	331,200,000	105,800,000	3.130					
48	`#pragma vector aligned`								
49	`for (int j = 0; j < n; j++)`	6,353,000,000	3,955,400,000	1.606					
50	`bTrial[i] += M[i*n+j]*x[j];`	90,304,000,0...	63,993,200,...	1.411					
51	`}`								
52	`accuracy = RelativeNormOfDifference(n, b, bTrial);`								
53									
54	`} while (accuracy > minAccuracy);`	23,000,000	18,400,000	1.250					
55	`return iterations;`	5,800,000	800,000	7.250					
56	`}`								
	Highlighted 1 row(s):	90,304,000,0...	63,993,200,...	1.411	0.992	0.206	0.001	0.779	0.015

Figure 5.17: Viewing the source code of a function.

Occasionally, the collected events are off by one line. This is because statistical sampling does not provide absolutely accurate data. Events are collected in sampling intervals, and all events in the entire sampling interval (thousands to millions of events) are attributed to a given code context. This may yield inaccurate results.

Nevertheless, the Source Code pane may is a very useful tool in itself, allowing to drill down into hotspots and sub-optimal hardware event culprits at the granularity of a single line of code.

There is an additional functionality available in the Source Code pane, which is especially helpful for the analysis of vectorization. This functionality is the Assembly pane. We click the button Assembly to see it (Figure 5.18). The assembly lines corresponding to the source code line in the Source Code pane are highlighted.


/home/vega/intel/amplxe/projects/VTune-Lab - Intel VTune Amplifier (on lyra)
Welcome
r000ge
General Exploration
General Exploration viewpoint (change)
Intel VTune Amplifier XE 2015
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Top-down Tree
Tasks and Frames
worker.cc
Source
Assembly
Assembly grouping: Address

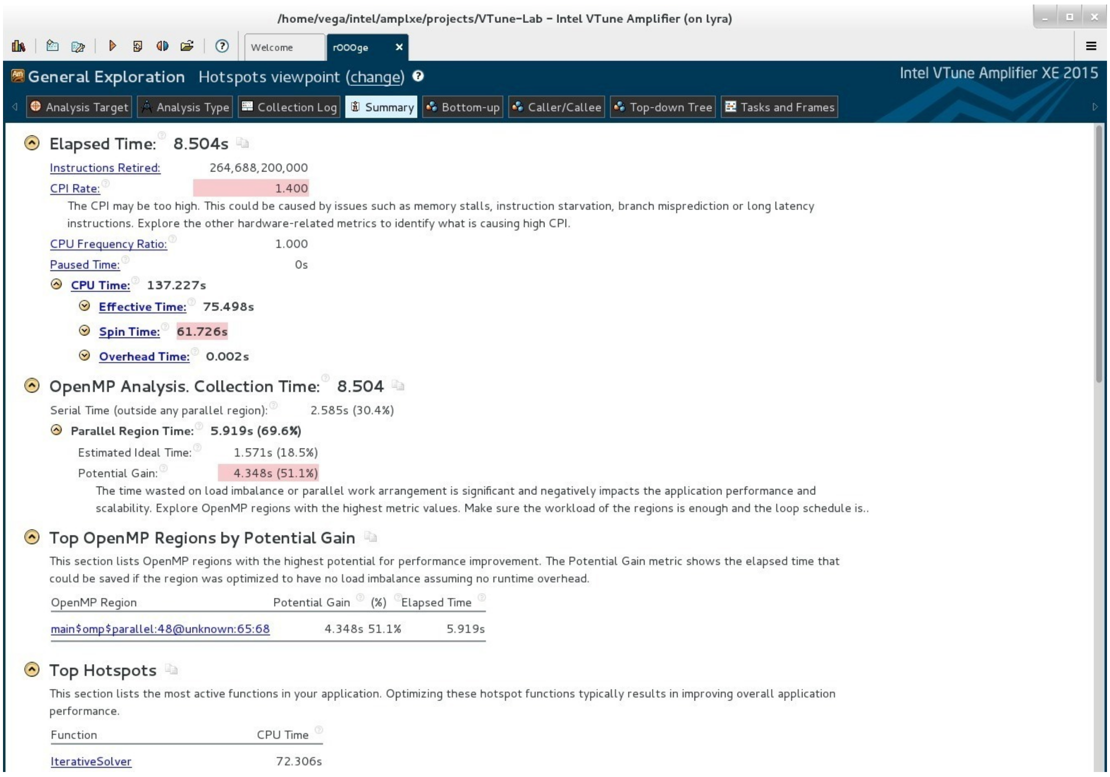
S. Li.	Source	Cl
34	do {	
35	iterations++;	
36	// Jacobi method	
37	for (int i = 0; i < n; i++) {	5,7
38	double c = 0.0;	3,4
39	#pragma vector aligned	
40	for (int j = 0; j < n; j++)	5,8
41	c += M[i*n+j]*x[j];	81,9
42	x[i] = x[i] + (b[i] - c)/M[i*n+i];	1,0
43	}	
44		
45	// Verification	
46	bTrial[:] = 0.0;	2
47	for (int i = 0; i < n; i++) {	3
48	#pragma vector aligned	
49	for (int j = 0; j < n; j++)	6,3
50	bTrial[i] += M[i*n+j]*x[j];	90,3
51	}	
52	accuracy = RelativeNormOfDifference(n, b, bTrial);	
53		
54	} while (accuracy > minAccuracy);	
55	return iterations;	
56	}	
	Highlighted 1 row(s):	90,3

Address	Sou.. Line	Assembly	Clockti...	Instru Reti
0x402704	50	vmovupdy 0x60(%rsi,%r9,8), %ymm13	16,104,6...	11,4
0x40270b	50	vmulpdy (%r12,%r9,8), %ymm7, %ymm8	8,096,40...	5,29
0x402711	50	vmulpdy 0x20(%r12,%r9,8), %ymm9, %ymm10	9,481,80...	6,74
0x402718	50	vmulpdy 0x40(%r12,%r9,8), %ymm11, %ymm12	8,586,60...	5,74
0x40271f	50	vmulpdy 0x60(%r12,%r9,8), %ymm13, %ymm14	8,862,20...	6,44
0x402726	50	vaddpd %ymm8, %ymm6, %ymm6	8,367,00...	5,56
0x40272b	50	vaddpd %ymm2, %ymm10, %ymm2	543,200,...	401,
0x40272f	50	vaddpd %ymm1, %ymm12, %ymm1	2,634,80...	1,87
0x402733	50	vaddpd %ymm0, %ymm14, %ymm0	1,909,60...	1,59
0x402737	49	add $0x10, %r9	4,565,60...	3,22
0x40273b	49	cmp %r10, %r9	249,200,...	176,
0x40273e	49	jb 0x4026f0 <Block 41>	600,000	400,
0x402740		**Block 42:**		
0x402740	50	vaddpd %ymm2, %ymm6, %ymm2	65,400,0...	28,8
0x402744	50	vaddpd %ymm0, %ymm1, %ymm0	126,600,...	57,4
0x402748	50	vaddpd %ymm0, %ymm2, %ymm1	553,600,...	183,
0x40274c	50	vextractf128 $0x1, %ymm1, %xmm6	757,800,...	284,
0x402752	50	vaddpd %xmm6, %xmm1, %xmm7	155,400,...	61,8
0x402756	50	vunpckhpd %xmm7, %xmm7, %xmm8	1,120,60...	395,
0x40275a	50	vaddsd %xmm8, %xmm7, %xmm0	124,000,...	44,2
0x40275f		**Block 43:**		
0x40275f	49	lea 0x1(%r8), %esi	874,800,...	282,
0x402763	49	cmp %esi, %edi	600,000	
0x402765	49	jb 0x402818 <Block 51>		
		Selected 1 row(s):	16,104,6...	11,4

Figure 5.18: Matching source code to assembly.

We can mouse over most of the symbols in the Assembly pane, and a help window with a description of this symbol will pop up. Analysis of assembly in VTune has an advantage over studying assembly listings produced by the compiler. In VTune, it is immediately evident from the measured events which code path was taken most often at runtime.

So far, we have been exploring information in VTune using the General Exploration viewpoint. It shows multiple performance metrics for all components, which may make it difficult to infer the most important information for identifying simply what takes time in an application. To focus the reported metrics on consumed time, we can switch to the Hotspots viewpoint using the control General Exploration viewpoint (change) under the toolbar. Figure 5.19 shows the Summary pane in the Hotspots viewpoint.


/home/vega/intel/amplxe/projects/VTune-Lab - Intel VTune Amplifier (on lyra)
Welcome
r000ge
General Exploration Hotspots viewpoint (change)
Intel VTune Amplifier XE 2015
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks and Frames

Elapsed Time: 8.504s

Instructions Retired: 264,688,200,000
CPI Rate: 1.400
The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.
CPU Frequency Ratio: 1.000
Paused Time: 0s
CPU Time: 137.227s
Effective Time: 75.498s
Spin Time: 61.726s
Overhead Time: 0.002s

OpenMP Analysis. Collection Time: 8.504

Serial Time (outside any parallel region): 2.585s (30.4%)
Parallel Region Time: 5.919s (69.6%)
Estimated Ideal Time: 1.571s (18.5%)
Potential Gain: 4.348s (51.1%)
The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance and scalability. Explore OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule is..

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	Potential Gain	(%)	Elapsed Time
main$omp$parallel:48@unknown:65:68	4.348s	51.1%	5.919s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	CPU Time
IterativeSolver	72.306s

Figure 5.19: Summary screen in the Hotspots viewpoint.

Scrolling down in that pane, we can find the concurrency histogram (Figure 5.20), which shows the amounts of time spent by the application at different parallel efficiencies. The smeared histogram in Figure 5.20 is an example of a poorly parallelized application (in this case, the culprit is load imbalance between threads).

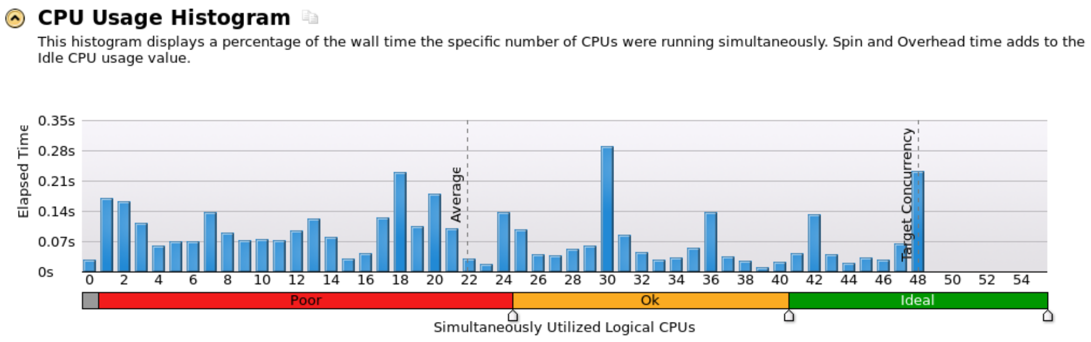


Figure 5.20: Concurrency histogram (non-optimized code).

Information such as in the above histogram is a clue for the programmer to investigate multi-threading issue. On the other hand, concurrency histogram as in Figure 5.21 concentrated in the "Ideal" range is a sign that parallelism in the application is adequate.

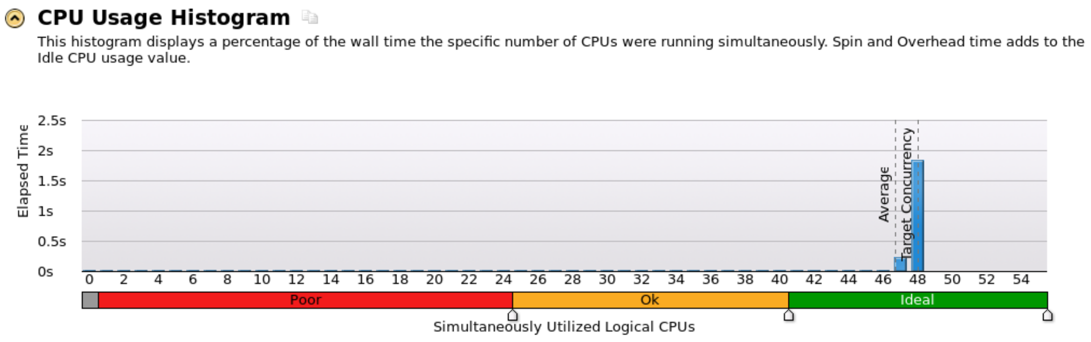
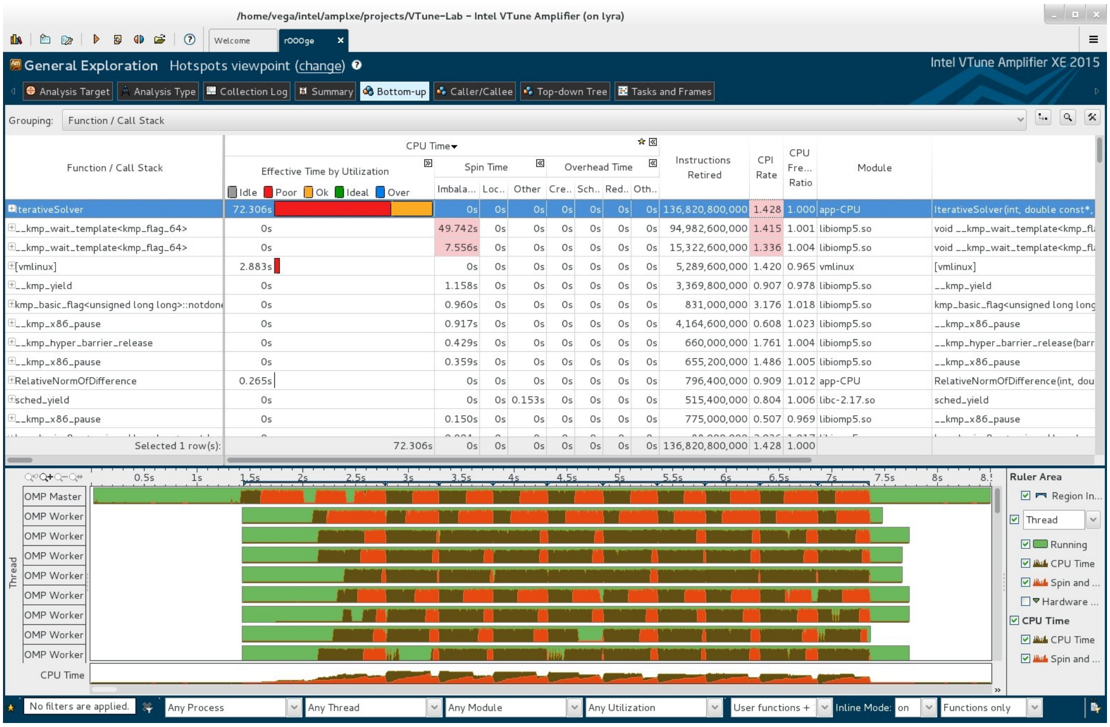


Figure 5.21: Concurrency histogram (optimized code).

In the Hotspots viewpoint, the color scheme used in the concurrency histogram translates to the Bottom-up pane and also to the Source Code pane. For instance, in the Bottom-up pane (Figure 5.22), column “Effective time by utilization” shows that the majority of time the application was in the “Poor” and a smaller fraction of time in the “OK” mode of operation..

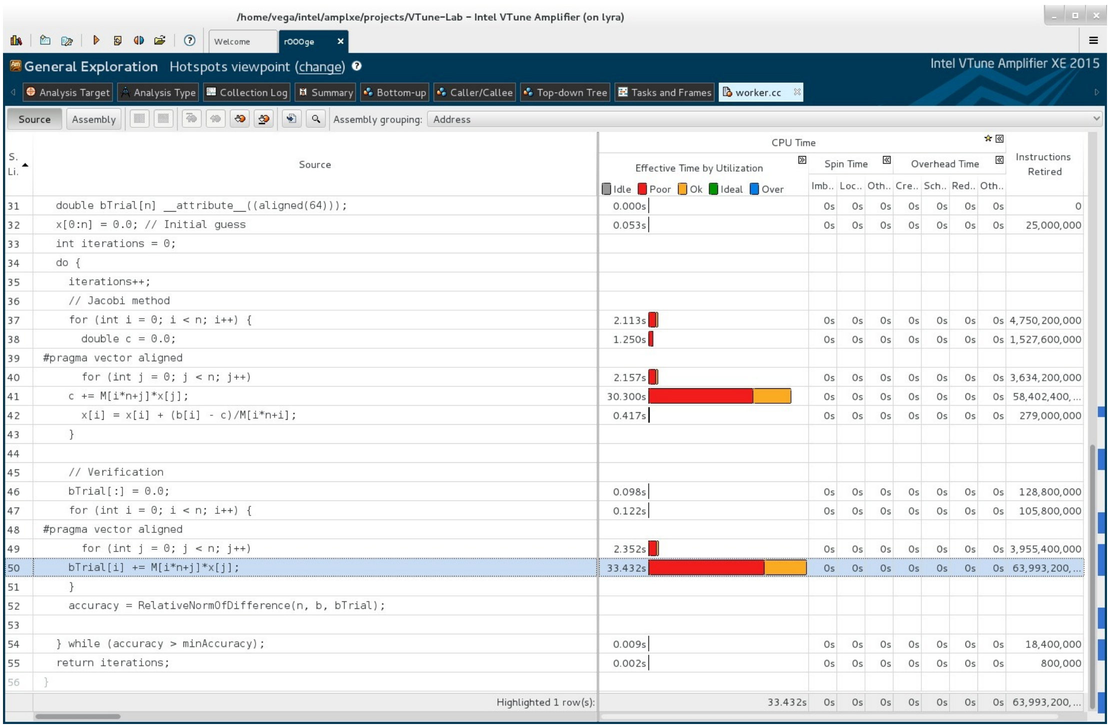


Function / Call Stack	CPU Time								Instructions Retired	CPI Rate	CPU Fre... Ratio	Module	
	Effective Time by Utilization	Spin Time			Overhead Time								
	▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Imbala...	Loc..	Other	Cre..	Sch..	Red..	Oth..					
IterativeSolver	72.306s 🟥🟧	0s	0s	0s	0s	0s	0s	0s	136,820,800,000	1.428	1.000	app-CPU	IterativeSolver(int, double const*,
__kmp_wait_template<kmp_flag_64>	0s	49.742s	0s	0s	0s	0s	0s	0s	94,982,600,000	1.415	1.001	libiomp5.so	void __kmp_wait_template<kmp_fl
__kmp_wait_template<kmp_flag_64>	0s	7.556s	0s	0s	0s	0s	0s	0s	15,322,600,000	1.336	1.004	libiomp5.so	void __kmp_wait_template<kmp_fl
[vmlinux]	2.883s 🟥	0s	0s	0s	0s	0s	0s	0s	5,289,600,000	1.420	0.965	vmlinux	[vmlinux]
__kmp_yield	0s	1.158s	0s	0s	0s	0s	0s	0s	3,369,800,000	0.907	0.978	libiomp5.so	__kmp_yield
kmp_basic_flag<unsigned long long>::notdone	0s	0.960s	0s	0s	0s	0s	0s	0s	831,000,000	3.176	1.018	libiomp5.so	kmp_basic_flag<unsigned long long
__kmp_x86_pause	0s	0.917s	0s	0s	0s	0s	0s	0s	4,164,600,000	0.608	1.023	libiomp5.so	__kmp_x86_pause
__kmp_hyper_barrier_release	0s	0.429s	0s	0s	0s	0s	0s	0s	660,000,000	1.761	1.004	libiomp5.so	__kmp_hyper_barrier_release(barr
__kmp_x86_pause	0s	0.359s	0s	0s	0s	0s	0s	0s	655,200,000	1.486	1.005	libiomp5.so	__kmp_x86_pause
RelativeNormOfDifference	0.265s	0s	0s	0s	0s	0s	0s	0s	796,400,000	0.909	1.012	app-CPU	RelativeNormOfDifference(int, dou
sched_yield	0s	0s	0s	0.153s	0s	0s	0s	0s	515,400,000	0.804	1.006	libc-2.17.so	sched_yield
__kmp_x86_pause	0s	0.150s	0s	0s	0s	0s	0s	0s	775,000,000	0.507	0.969	libiomp5.so	__kmp_x86_pause
Selected 1 row(s):	72.306s	0s	0s	0s	0s	0s	0s	0s	136,820,800,000	1.428	1.000		

Figure 5.22: Bottom-up pane in the Hotspots viewpoint.

The timeline in Figure 5.22 is useful for detecting load imbalance. Orange-colored regions indicate spinning and idling threads, and brown regions are threads using CPU time. The abundance of orange gaps at the end of each of the 10 application cycles indicates that some threads were working while other were idling. This points the programmer to optimizing load balance.

Finally, let’s double-click the top hotspot (function `IterativeSolver`) to open the Source Code pane (Figure 5.23). As you can see, color-coded concurrency markers are present in front of every line, allowing not only to gauge how much CPU time each line took, but also what fraction of that time the application was running in parallel or was serialized.


/home/vega/intel/amplxe/projects/VTune-Lab – Intel VTune Amplifier (on lyra)
Welcome
r000ge
General Exploration Hotspots viewpoint (change)
Intel VTune Amplifier XE 2015
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks and Frames
worker.cc
Source
Assembly
Assembly grouping: Address

S. Li.	Source	CPU Time								Instructions Retired
		Effective Time by Utilization	Spin Time			Overhead Time				
		▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Imb..	Loc..	Oth..	Cre..	Sch..	Red..	Oth..	
31	`double bTrial[n] __attribute__((aligned(64)));`	0.000s	0s	0s	0s	0s	0s	0s	0s	0
32	`x[0:n] = 0.0; // Initial guess`	0.053s	0s	0s	0s	0s	0s	0s	0s	25,000,000
33	`int iterations = 0;`									
34	`do {`									
35	`iterations++;`									
36	`// Jacobi method`									
37	`for (int i = 0; i < n; i++) {`	2.113s	0s	0s	0s	0s	0s	0s	0s	4,750,200,000
38	`double c = 0.0;`	1.250s	0s	0s	0s	0s	0s	0s	0s	1,527,600,000
39	`#pragma vector aligned`									
40	`for (int j = 0; j < n; j++)`	2.157s	0s	0s	0s	0s	0s	0s	0s	3,634,200,000
41	`c += M[i*n+j]*x[j];`	30.300s	0s	0s	0s	0s	0s	0s	0s	58,402,400,...
42	`x[i] = x[i] + (b[i] - c)/M[i*n+i];`	0.417s	0s	0s	0s	0s	0s	0s	0s	279,000,000
43	`}`									
44										
45	`// Verification`									
46	`bTrial[:] = 0.0;`	0.098s	0s	0s	0s	0s	0s	0s	0s	128,800,000
47	`for (int i = 0; i < n; i++) {`	0.122s	0s	0s	0s	0s	0s	0s	0s	105,800,000
48	`#pragma vector aligned`									
49	`for (int j = 0; j < n; j++)`	2.352s	0s	0s	0s	0s	0s	0s	0s	3,955,400,000
50	`bTrial[i] += M[i*n+j]*x[j];`	33.432s	0s	0s	0s	0s	0s	0s	0s	63,993,200,...
51	`}`									
52	`accuracy = RelativeNormOfDifference(n, b, bTrial);`									
53										
54	`} while (accuracy > minAccuracy);`	0.009s	0s	0s	0s	0s	0s	0s	0s	18,400,000
55	`return iterations;`	0.002s	0s	0s	0s	0s	0s	0s	0s	800,000
56	`}`									
	Highlighted 1 row(s):	33.432s	0s	0s	0s	0s	0s	0s	0s	63,993,200,...

Figure 5.23: Drilling down to the source code (Hotspots viewpoint).

By detecting hotspots and general performance issue, the developer may focus on the low-hanging fruits for optimization first, and modify the application accordingly.

To see the effect of any optimization, we can click ▶ to produce another result under the same project. The new result may be compared to older results. This may be done either "by hand" or using the built-in comparison tool available at the button ◐.

5.2.5. Analysis on an Intel Xeon Phi Coprocessor

In Section 5.2.4 we discussed how to analyze in VTune the performance of an application running on the host processor. Now we will create a new project, in which we will study the performance of applications on Intel Xeon Phi coprocessors.

Native Applications

To analyze an application designed to run natively on a coprocessor, we begin by creating a script that launches this application on the coprocessor (i.e., it will be a script executed in the coprocessor OS). VTune will access the coprocessor via SSH under the user's account and launch the script. The script may contain initialization of environment variables, staging of data and executables, passing of command line arguments and any other related tasks. The script that we used is shown in Listing 5.13.

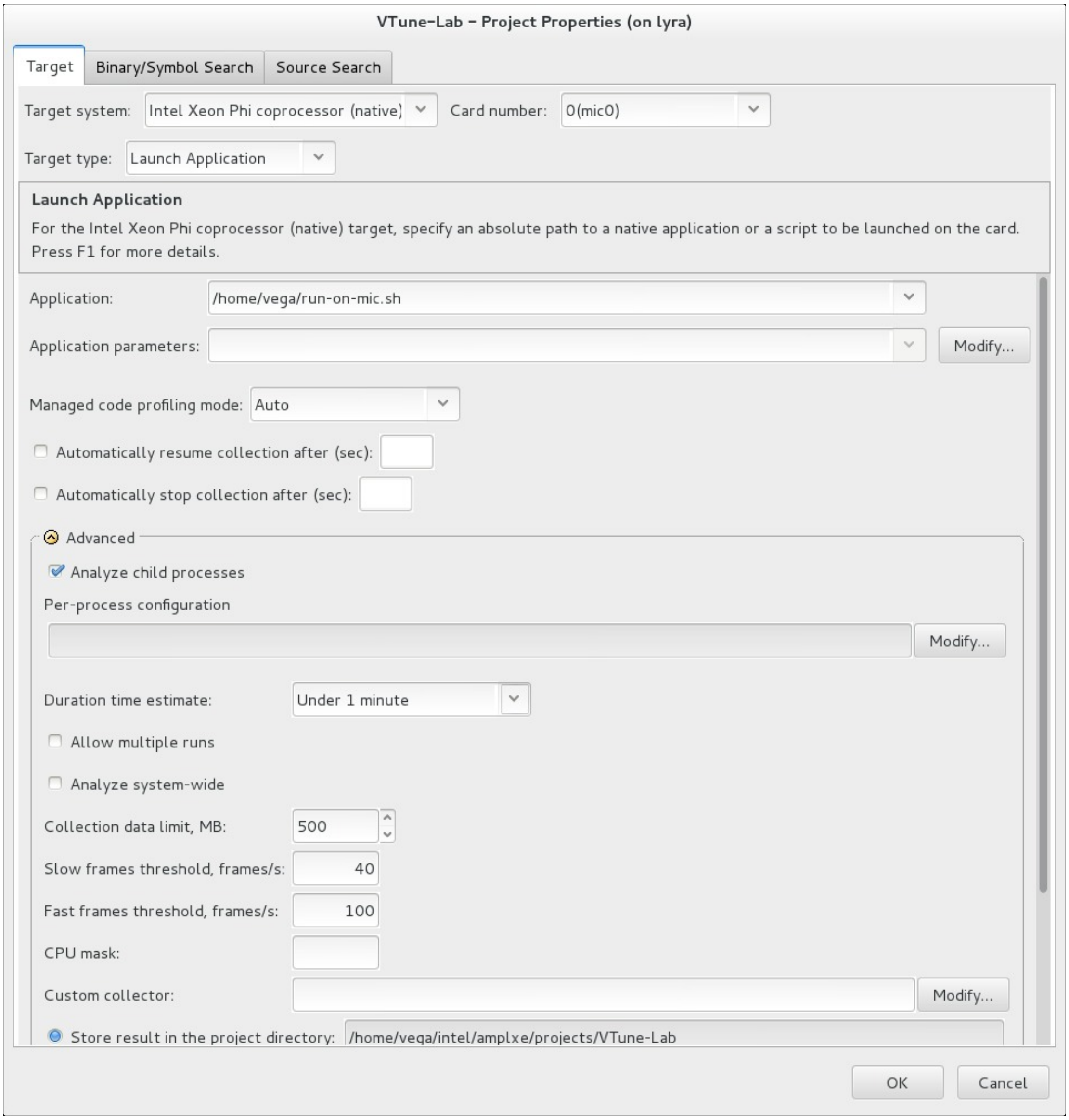
```
#!/bin/bash

# Set up general environment variables
source /opt/intel/parallel_studio_xe_2015/psxevars.sh
# Where to look for MIC architecture runtime libraries:
export LD_LIBRARY_PATH=$MIC_LD_LIBRARY_PATH

# Application-specific variables
export OMP_SCHEDULE=static

# Launch application
PATH_TO_EXECUTABLE=""
cd ~/${PATH_TO_EXECUTABLE}
./app-MIC
```

Listing 5.13: Script `run-on-mic.sh` launching the analyzed application on the coprocessor.

The executable `app-MIC` in this script is compiled with flags `-mmic` (to produce a native application) and `-g -O3` (to facilitate symbol recognition).

To analyze this application in VTune, we should create a new project (Figure 5.24). In the line "Target system", we chose "Intel Xeon Phi coprocessor(native)", which changes the layout of the prompt. Specifically, the option to indicate environment variables via the GUI is no longer there, which is where the usage of a wrapper script proves helpful.

With the new project configured, the workflow for analysis is similar to that for a host application, however, available analysis types must be chosen from the group "Knights Corner Platform" (see the sidebar menu in Figure 5.12).


VTune-Lab – Project Properties (on lyra)

Target | Binary/Symbol Search | Source Search

Target system: Intel Xeon Phi coprocessor (native) Card number: 0(mic0)

Target type: Launch Application

Launch Application
For the Intel Xeon Phi coprocessor (native) target, specify an absolute path to a native application or a script to be launched on the card. Press F1 for more details.

Application: /home/vega/run-on-mic.sh

Application parameters:

Managed code profiling mode: Auto

☐ Automatically resume collection after (sec):

☐ Automatically stop collection after (sec):

Advanced

☑ Analyze child processes

Per-process configuration

Duration time estimate: Under 1 minute

☐ Allow multiple runs

☐ Analyze system-wide

Collection data limit, MB: 500

Slow frames threshold, frames/s: 40

Fast frames threshold, frames/s: 100

CPU mask:

Custom collector:

● Store result in the project directory: /home/vega/intel/amplxe/projects/VTune-Lab

Figure 5.24: Setting up a VTune project for native application performance analysis on an Intel Xeon Phi coprocessor.

Offload Applications

For applications using an Intel Xeon Phi coprocessorin the offload mode, project setup is slightly different. The script that launches the application in this case (Figure 5.14) is executed in the host OS. If any environment variables need to be set, they should be passed to the offloaded code using the environment forwarding rules discussed in Section 2.2.9.

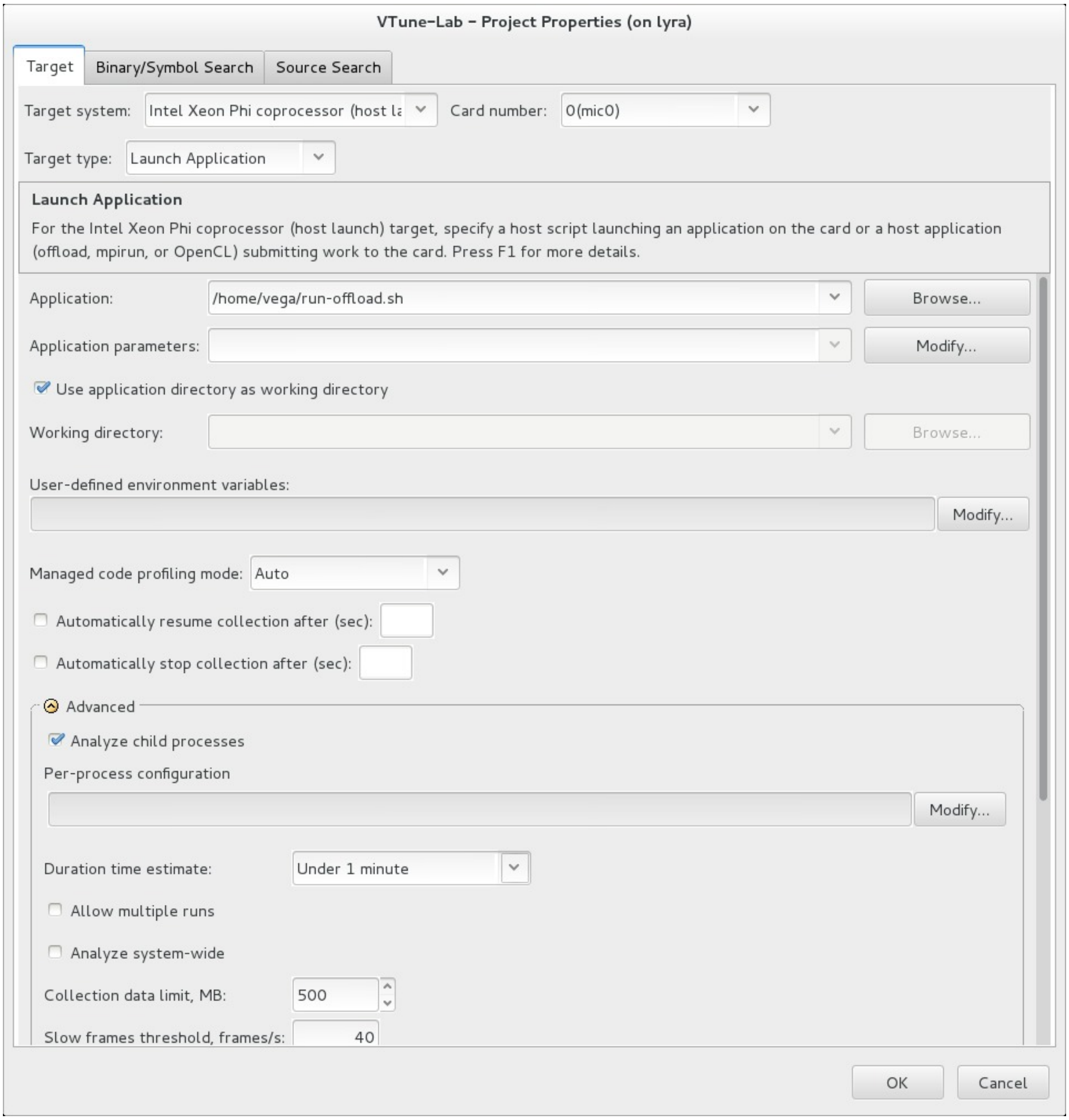
```
#!/bin/bash

# Parallel Studio environment variables:
source /opt/intel/parallel_studio_xe_2015/psxevars.sh

# Application-specific variables for offload
export MIC_ENV_PREFIX=PHI
export PHI_OMP_SCHEDULE=static

./app-OFF
```

Listing 5.14: Script `run-offload.sh` launching the analyzed offload application.

Finally, in VTune, the project is configured with “Target system” set to “Intel Xeon Phi coprocessor (host launch)”. This means that the application specified in the line “Application:” is launched on the host, however, results are collected on the coprocessor.

Like with native applications, the workflow of performance analysis is similar to that for host applications, however, analysis types must be chosen from the group “Knights Corner Platform”.

This concludes our brief tour of Intel VTune Amplifier XE. For more information and examples, refer to the User’s Guide and Tuning Guides and Performance Analysis Papers.


VTune-Lab – Project Properties (on lyra)

Target | Binary/Symbol Search | Source Search

Target system: Intel Xeon Phi coprocessor (host la
Card number: 0(mic0)

Target type: Launch Application

Launch Application
For the Intel Xeon Phi coprocessor (host launch) target, specify a host script launching an application on the card or a host application (offload, mpirun, or OpenCL) submitting work to the card. Press F1 for more details.

Application: /home/vega/run-offload.sh
Application parameters:
[✓] Use application directory as working directory
Working directory:
User-defined environment variables:
Managed code profiling mode: Auto
[ ] Automatically resume collection after (sec):
[ ] Automatically stop collection after (sec):

Advanced
[✓] Analyze child processes
Per-process configuration
Duration time estimate: Under 1 minute
[ ] Allow multiple runs
[ ] Analyze system-wide
Collection data limit, MB: 500
Slow frames threshold, frames/s: 40

Figure 5.25: Setting up a VTune project for offload application performance analysis on an Intel Xeon Phicoprocessor.

CHAPTER 6 Summary and Resources

Thank you for learning about Intel Xeon Phi coprocessor programming with "Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors" by Colfax International! We hope that, whatever scope of information you were looking for, you were able to find answers or pointers in this book. In this last brief chapter, we will summarize the key findings of our experience with Intel Xeon Phi coprocessor programming, and provide references for future learning.

6.1. Parallel Programming and Intel Xeon Phi Coprocessors

Parallel computing will be extremely important in the future on all levels of high performance computing, from workstation to exascale. The launch of the Intel Xeon Phi coprocessors changed the landscape of high performance computing by offering developers accelerated computing that works with familiar programming models and standard frameworks such as OpenMP and MPI.

In the programming and optimization examples presented throughout this book, we strived to convey two important messages:

1) Optimization methods that benefit applications for Intel Xeon Phi coprocessors usually also apply to Intel Xeon processors, and vice-versa.

2) High performance can be achieved without "ninja programming" [46]; instead, automatic vectorization and traditional parallel frameworks such as OpenMP and MPI can be used for optimization.

Consequently, an attractive feature of Intel Xeon Phi coprocessors is that the developer may develop and optimize the computational kernel code only once. That single source code can be used on today's Intel Xeon processors, Intel Xeon Phi coprocessors, future MIC architecture processors, and other platforms based on x86-like architecture. In this sense, applications designed for the MIC architecture using common programming methods are “future-proof”.

Even though it is easy to make an application *run* on an Intel Xeon Phi coprocessor, it is important to realize that it is not trivial to achieve *high performance*. At the same time, neither it is trivial to efficiently program multi-core general-purpose processors such as Intel Xeon CPUs. However, because of the continuity of programming and optimization methods, optimization for the CPU results in better performance on the MIC architecture, and vice-versa. That said, we concur with Intel’s James Reinders, who expresses the “double advantage” in this way [5]:

> The single most important lesson from working with Intel Xeon Phi coprocessors is this: the best way to prepare for Intel Xeon Phi coprocessors is to fully exploit the performance that an application can get on Intel Xeon processors first. Trying to use an Intel Xeon Phi coprocessor, without having maximized the use of parallelism on Intel Xeon processor, will almost certainly be a disappointment.
>
> . . .
>
> The experiences of users of Intel Xeon Phi coprocessors . . . point out one challenge: the temptation to stop tuning before the best performance is reached. . . . There ain’t no such thing as a free lunch! The hidden bonus is the “transforming-and-tuning” double advantage of programming investments for Intel Xeon Phi coprocessors that generally applies directly to any general-purpose processor as well. This greatly enhances the preservation of any investment to tune working code by applying to other processors and offering more forward scaling to future systems.

We hope that your experience with the Intel Xeon Phi coprocessors is as intellectually stimulating and enjoyable as it has been for us.

6.2. Supplementary Code for Practical Exercises (“Labs”)

This book is accompanied by a set of practical exercises, also referred to as “labs”. These exercises

- are the basis of some of the chapters in this book,
- may be used for self-study practice, and
- are used in the training course CDT 102 (see Section 6.3).

Downloading the Labs

You can download the latest version of the labs distributed under the MIT license from https://github.com/ColfaxResearch/HOW-Series-Labs

Terms of Use

Terms of use of the book and practical exercises are listed on page ii (between the title page and the page “About the Authors”).

System Requirements

The labs are designed for use in systems running a Linux* OS and based on an Intel Xeon processor.

Most labs are focused on using Intel Xeon Phi coprocessors, so having one or several coprocessors in the system enhances user experience.

All labs rely on Intel software development tools. At the very least, the Intel C++ compiler must be installed. Some of the labs also rely on Intel MKL, Intel MPI and Intel VTune Amplifier. See Section 1.2.7 for more information about these tools.

Orientation

The electronic archive containing the list of labs may be uncompressed into a directory tree shown in Figure 6.1.

```
labs
├── 2
│   ├── 2.01-native-basic
│   ├── 2.02-native-MPI
│   ├── 2.03-offload-basic
│   ├── 2.04-offload-asynchronous
│   ├── 2.05-shared-virtual-memory-basic
│   ├── 2.06-shared-virtual-memory-complex-objects
│   └── 2.07-benchmark-offload
├── 3
│   ├── 3.01-vectorization
│   ├── 3.02-OpenMP-basics
│   ├── 3.03-OpenMP-reduction
│   ├── 3.04-OpenMP-tasks
│   ├── 3.05-Cilk-Plus-basics
│   ├── 3.06-Cilk-Plus-reducers
│   ├── 3.07-Cilk-Plus-recursion
│   ├── 3.08-MPI-basics
│   ├── 3.09-MPI-reduce
│   └── 3.10-overview-matrix-matrix
├── 4
│   ├── 4.01-overview-nbody
│   ├── 4.02-vectorization-data-structures-coulomb
│   ├── 4.03-vectorization-tuning-lu-decomposition
│   ├── 4.04-threading-misc-histogram
│   ├── 4.05-threading-insufficient-parallelism-sweep
│   ├── 4.06-threading-scheduling-jacobi
│   ├── 4.07-threading-affinity
│   ├── 4.08-memory-tiling-matrix_x_vector
│   ├── 4.09-memory-loop-fusion-statistics
│   ├── 4.10-offload-double-buffering-dgemm
│   └── 4.11-MPI-load-balancing-asian-options
└── 5
    ├── 5.01-mkl
    └── 5.02-vtune
```

Figure 6.1: Directory tree of labs.

Each lab contains a tree structure similar to that in Figure 6.2.

```
labs
└─4
  └─4.05-threading-insufficient-parallelism-sweep
    ├─instructions.txt
    ├─main.cc
    ├─Makefile
    ├─worker.cc
    └─solutions
      ├─instruction-01
      │ ├─main.cc
      │ ├─Makefile
      │ └─worker.cc
      ├─instruction-02
      │ ├─main.cc
      │ ├─Makefile
      │ └─worker.cc
      └─instruction-03
        ├─main.cc
        ├─Makefile
        └─worker.cc
```

Figure 6.2: Typical directory and file tree of a lab.

Start working on each lab by viewing the file `instructions.txt` at the root of that tree. It guides the reader through the educational tasks that must be performed in this exercise. If you would like to see our solution, refer to the directory `solutions/` in the lab folder. It typically contains directories such as `instruction-01`, `instruction-02` with solutions to the corresponding assignments in `instructions.txt`.

Note that in some labs, the code must be compiled manually for the purpose of learning how to do it. In other labs, we provide a makefile, which is also capable of running the lab. To compile for the CPU architecture and execute, run the command `make run-cpu`. To compile for the MIC architecture and run, use `make run-mic`.

6.3. Colfax Developer Training

Colfax International offers multiple opportunities for getting trained to use Intel Xeon Phi coprocessors and Intel software development tools. Colfax Developer Training (CDT) is available in the following formats:

Course	Duration	Format	Description
CDT 101	1 day	Seminar	Intensive lecture-only course: an overview of parallel programming frameworks and optimization guidelines for multi-core CPUs (Intel Xeon) and many-core coprocessors (Intel Xeon Phi).
CDT 102	1 day	Labs	Instructor-led hands-on practical exercises: programming models, expressing parallelism, selected optimization topics.
CDT 401	4 day	Workshop	Immersive training with strong focus on performance optimization: lectures and hands-on exercises on systems with Intel Xeon Phi coprocessors.
CDT S01	Variable	Self-study	Remote access to systems equipped with Intel Xeon Phi coprocessors and Intel software development tools and access to this book and labs for self-guided training*.
CDT V01	Variable	Video course	Lectures of CDT 101 available on demand as a video course. *

* – coming soon (status as of April 2015).

Table 6.1: Colfax Developer Training

For information on booking the training, please visit http://www.colfax-intl.com/nd/xeonphi/training.aspx

6.4. Additional Resources

Books

We can recommend the following books for additional information on parallel programming and the Intel MIC architecture.

1) Another textbook on programming for the MIC architecture from Intel's senior engineers Jim Jeffers and James Reinders can be found in "Intel Xeon Phi Coprocessor High Performance Programming" [3]. The book has a Web site at http://lotsofcores.com/ [47]

2) A collection of case studies written by 28 high performance computing experts and edited by Reinders and Jeffers is "High Performance Parallelism Pearls" [48]. Source code for each case study is available at http://lotsofcores.com/.

3) A book focused on the architecture of the first generation of Intel Xeon Phi coprocessors is "Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers (Expert's Voice in Microprocessors)" by Rezaur Rahman [49].

4) For a solid foundation of traditional parallel programming methods with OpenMP and MPI, refer to "Parallel Programming in C with MPI and OpenMP" by Michael J. Quinn [20].

5) A new look at parallel algorithms and novel parallel frameworks are presented in "Structured Parallel Programming: Patterns for Efficient Computation" by Michael McCool, Arch D. Robinson and James Reinders [19]. The Web site of the book is http://parallelbook.com/ [50].

6) To gain a better understanding of computer architecture in general, and specifically the architecture of Intel Xeon Phi coprocessors, refer to "Compute Architecture: Quantitative Approach" by John L. Hennessy and David A. Patterson [4] and "Intel Xeon Phi Coprocessor System Software Developer's Guide", a publication by Intel [51].

Reference Guides

The following list is a collection of URLs for software development tool and programming framework reference guides.

1. Intel C++ Compiler User and Reference Guide [15]:
   https://software.intel.com/en-us/compiler_15.0_ug_c
2. Intel VTune Amplifier XE User's Guide [45]:
   https://software.intel.com/en-us/node/529797
3. Intel MPI Library Reference Manual [13]:
   https://software.intel.com/en-us/mpi-refman-lin-5.0.3-html
4. MPI routines on the ANL Web site [52]:
   http://www.mpich.org/static/docs/latest/
5. OpenMP specifications [53]:
   http://openmp.org/wp/openmp-specifications/

Online Resources

1) Colfax Research publications on Intel MIC architecture programming are available at http://colfaxresearch.com/

2) Intel Developer Zone has a portal on the MIC architecture with white papers, links to products, forums and case studies, and other essential information: https://software.intel.com/mic-developer

3) This online resource "Programming and Compiling for Intel Many Integrated Core Architecture" [30] contains condensed information about optimization of applications for the Intel Xeon Phi architecture: http://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture

4) Colfax International is preparing a video course based on the CDT 101 training course (see Section 6.3). When completed, the video course will be available at http://xeonphi.com/training/video.

Community Support

1) The forum “Intel Many Integrated Core Architecture” in the Intel Developer Zone is a great place to ask questions and exchange ideas:
https://software.intel.com/en-us/forums/intel-many-integrated-core
This forum gets contributions from developers working with Intel Xeon Phi coprocessors, and it is also monitored by Intel’s engineers involved in the development of the MIC architecture.

2) Another forum in the Intel Developer Zone, “Threading on Intel Parallel Architectures”
http://software.intel.com/en-us/forums/threading-on-intel-parallel-architectures
is a good place to communicate with peers about parallel programming, not necessarily in the context of the MIC architecture.

3) Find connections and stay updated on the latest news related to the MIC technology by joining the LinkedIn group “Parallel Computing with Intel Xeon Phi Coprocessors”:
http://www.linkedin.com/groups/Parallel-Computing-Intel-Xeon-Phi-4722265/about

Contact Us

If you have questions, ideas, suggestions, corrections, or need information about purchasing or test-driving computing systems with Intel Xeon Phi coprocessors, please refer to:

a) the Colfax International Web site http://www.colfax-intl.com/

b) page for Intel Xeon Phi: http://www.colfax-intl.com/nd/xeonphi/

c) or contact us at the following email address: phi@colfax-intl.com.

Bibliography

[1] Andrey Vladimirov and Vadim Karpusenko. Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation.
https://colfaxresearch.com/test-driving-intel-xeon-phi-coprocessors-with-a-basic-n-body-simulation/.

[2] Intel Xeon Phi Coprocessor Applications and Solutions Catalog.
https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-applications-and-solutions-catalog.

[3] Jim Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann, 1st edition, March 2013.

[4] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, 5th edition, 2011.

[5] James Reinders. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors.
http://software.intel.com/sites/default/files/blog/337861/reindersxeonandxeonphi-v20121112a.pdf.

[6] Intel Instruction Set Architecture Extensions.
https://software.intel.com/en-us/intel-isa-extensions.

[7] Joe Curley. The Faster Path to Discovery: New Details on the Intel Xeon Phi Product Family (webinar).
https://www.brighttalk.com/webcast/10773/116329.

[8] Andrey Vladimirov. Installing Intel MPSS 3.3 on Arch Linux.
https://colfaxresearch.com/installing-intel-mpss-3-3-in-arch-linux/.

[9] Intel Many Core Platform Software Stack (MPSS).
http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[10] Andrey Vladimirov and Vadim Karpusenko. Configuration and Benchmarks of Peer-to-Peer Communication over Gigabit Ethernet and InfiniBand in a Cluster with Intel Xeon Phi Coprocessors, 2014.
https://colfaxresearch.com/configuration-and-benchmarks-of-peer-to-peer-communication-over-gigabit-ethernet-and-infiniband-in-a-cluster-with-intel-xeon-phi-coprocessors/.

[11] Linux NFS Howto.
http://nfs.sourceforge.net/nfs-howto/.

[12] Michael Hebenstreit. Configuring Intel Xeon Phi coprocessors inside a cluster.
https://software.intel.com/en-us/articles/configuring-intel-xeon-phi-coprocessors-inside-a-cluster.

[13] Intel MPI 5.0.3 Reference Manual for Linux OS*.
https://software.intel.com/en-us/mpi-refman-lin-5.0.3-html.

[14] Aart Bik. *The software vectorization handbook. Applying multimedia extensions for maximum performance*. Intel Press, 2006.

[15] Intel C++ Compiler XE 15.0 Reference.
https://software.intel.com/en-us/node/522786.

[16] Intel intrinsic guide.
https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[17] Vadim Karpusenko and Andrey Vladimirov. How to Write Your Own Blazingly Fast Library of Special Functions for Intel Xeon Phi Coprocessors .
https://colfaxresearch.com/write-your-own-blazingly-fast-library-of-special-functions-for-intel-xeon-phi-coprocessors/.

[18] Andrey Vladimirov. Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors.
https://colfaxresearch.com/multithreaded-transposition-of-square-matrices-with-common-code-

for-intel-xeon-processors-and-intel-xeon-phi-coprocessors/.

[19] Michael McCool, Arch D. Robinson, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[20] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[21] Ryo Asai and Andrey Vladimirov. Large Fast Fourier Transforms with FFTW 3.3 on Terabyte-RAM NUMA Servers.
https://colfaxresearch.com/intel-cilk-plus-for-complex-parallel-algorithms-enormous-fast-fourier-transforms-efft-library/.

[22] Andrey Vladimirov. Cluster-Level Tuning of a Shallow Water Equation Solver on the Intel MIC Architecture.
https://colfaxresearch.com/cluster-level-tuning-of-a-shallow-water-equation-solver-on-the-intel-mic-architecture/.

[23] Andrey Vladimirov. Performance to Power and Performance to Cost Ratios with Intel Xeon Phi Coprocessors (and why 1x Acceleration May Be Enough).
https://colfaxresearch.com/performance-to-power-and-performance-to-cost-ratios-with-intel-xeon-phi-coprocessors-and-why-1x-acceleration-may-be-enough/.

[24] Martyn J. Corden and David Kreitzer. Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?
http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler.

[25] Wendy Doerner. Advanced Optimizations for Intel MIC Architecture, Low Precision Optimizations.
http://software.intel.com/en-us/articles/advanced-optimizations-for-intel-mic-architecture-low-precision-optimizations.

[26] Units in the last place - Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Unit_in_the_last_place.

[27] Andrey Vladimirov. Arithmetics on Intel's Sandy Bridge and Westmere CPUs: not all FLOPs are Created Equal.
https://colfaxresearch.com/arithmetics-on-intels-sandy-bridge-and-westmere-cpus-not-all-flops-are-created-equal/.

[28] ROOT, a Data Analysis Framework.
http://root.cern.ch/.

[29] Andrey Vladimirov. Auto-Vectorization with the Intel Compilers: is Your Code Ready for Sandy Bridge and Knights Corner?, 2012.
https://colfaxresearch.com/auto-vectorization-with-the-intel-compilers-is-your-code-ready-for-sandy-bridge-and-knights-corner/.

[30] Intel. Programming and compiling for intelÂő many integrated core architecture.
https://software.intel.com/en-us/articles/programming-and-compiling-for-intel-many-integrated-core-architecture.

[31] Andrey Vladimirov. Fine-Tuning Vectorization and Memory Traffic on Intel Xeon Phi Coprocessors: LU Decomposition of Small Matrices.
https://colfaxresearch.com/fine-tuning-vectorization-and-memory-traffic-on-intel-xeon-phi-coprocessors-lu-decomposition-of-small-matrices/.

[32] Nicolas Butler. Concurrency Hazards: False Sharing.
http://simplygenius.net/Article/FalseSharing.

[33] Rob Farber. Power Profiling Shows Simple Changes To Save Megawatts of Power On Leadership Supercomputers.
http://www.techenablement.com/power-profiling-shows-simple-changes-save-megawatts-power-leadership-supercomputers/.

[34] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers.
https://www.cs.virginia.edu/stream/.

[35] Karthik Raman. Optimizing Memory Bandwidth on Stream Triad. https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad.

[36] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009. http://dx.doi.org/doi:10.1145/1498765.1498785.

[37] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999. http://supertech.csail.mit.edu/papers/Prokop99.pdf.

[38] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science*, 1999. http://doi.ieeecomputersociety.org/10.1109/SFFCS.1999.814600.

[39] Rakesh Krishnaiyer. Compiler prefetching for Intel Xeon Phi coprocessors. https://software.intel.com/sites/default/files/managed/7f/1f/5.3_Prefetching_on_MIC_6.pdf.

[40] Andrey Vladimirov and Vadim Karpusenko. Heterogeneous Clustering with Homogeneous Code: Accelerate MPI Applications Without Code Surgery Using Intel Xeon Phi Coprocessors, 2013. https://colfaxresearch.com/heterogeneous-clustering-with-homogeneous-code-accelerate-mpi-applications-without-code-surgery-using-intel-xeon-phi-coprocessors/.

[41] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing, 1998. http://supertech.csail.mit.edu/papers/steal.pdf.

[42] Crash Course on Programming and Optimization with Intel Xeon Phi Coprocessors at SC14. https://colfaxresearch.com/crash-course-on-

programming-and-optimization-with-intel-xeon-phi-coprocessors-at-sc14/.

[43] Intel Math Kernel Library 11.2 Reference Manual. https://software.intel.com/en-us/mkl_11.2_ref.

[44] Intel Math Kernel Library Link Line Advisor. http://software.intel.com/sites/products/mkl/MKL_Link_Line_Advisor.html.

[45] Intel VTune Parallel Amplifier XE 2015 Help for Linux* OS. https://software.intel.com/en-us/node/529213.

[46] Changkyu Kim et al. Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-closing-ninja-gap-paper.pdf.

[47] Jim Jeffers and James Reinders. Web Site for the book "Intel Xeon Phi Coprocessor High Performance Programming". http://www.lotsofcores.com/.

[48] Jim Jeffers and James Reinders. *High Performance Parallelism Pearls: Multi-core and Many-core Programming Approaches*. Morgan Kaufmann, 1st edition, November 2014.

[49] Rezaur Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers (Expert's Voice in Microprocessors)*. Apress, 1st edition, September 2013.

[50] Michael McCool, Arch D. Robinson, and James Reinders. Web Site for the book "Structured Parallel Programming: Patterns for Efficient Computation". http://parallelbook.com/.

[51] Intel. Intel Xeon Phi Coprocessor System Software Developers Guide. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide.

[52] Web Pages for MPI Routines at the Argonne National Laboratory Seb Site.
http://www.mpich.org/static/docs/latest/.

[53] OpenMP Specifications.
http://openmp.org/wp/openmp-specifications/.

IT IS ALL ABOUT OPTIMIZING PARALLELISM

Parallelism has long been a nonnegotiable requirement of all high performance computing applications in supercomputer sites. These days, parallel computing is also becoming commonplace in smaller computing environments: private clusters, workstations and portable computers. Computer architectures grow in size (more compute nodes in a cluster, more cores on a chip) and evolve in depth (wider SIMD registers, deeper pipelines). Harnessing this rocketing growth of hardware capabilities to tackle new fascinating computing problems requires that software developers continually learn to optimize their applications to utilize all available levels and scope of hardware parallelism.

In Parallel Computing and Optimization with Intel Xeon Phi Coprocessors, Colfax International presents to high performance application developers the state-of-the-art programming paradigms and best optimization practices for modern computing platforms based on the Intel multi-core and Many Integrated Core (MIC) architectures.

In this example-based intensive guide to programming Intel Xeon Phi coprocessors, you will find:

- An overview of Intel MIC processors of the first generation (Knights Corner) and second generation (Knights Landing);

- Introduction to task- and data-parallel programming with MPI, OpenMP, Intel Cilk Plus, and automatic vectorization with Intel C and C++ compiler;

- Extensive discussion of high performance application optimization on the Intel Xeon and Intel Xeon Phi platforms, including scalar optimizations, improvement of SIMD operations, multithreading, efficient cache utilization, communication control, and scaling across heterogeneous distributed-memory computing systems;

- A discussion of system administration tasks for workstations and clusters with Intel Xeon Phi coprocessors;

- Supplementary code for practical exercises (self-study "labs") comprising 30 guided exercises with solutions, also used in the Colfax Developer Training program.

COLFAX

www.colfax-intl.com