ACM Parallel Computing Tech Pack

Journeyman's Programming Tour

November, 2010

Parallel Computing Committee

Paul Steinberg, Intel, *Co-Chair*
Matthew Wolf, CERCS, Georgia Tech, *Co-Chair*
Judith Bishop, Microsoft
Clay Breshears, Intel
Barbara Mary Chapman, University of Houston
Daniel J. Ernst, University of Wisconsin-Eau Claire
Andrew Fitz Gibbon, Shodor Foundation
Dan Garcia, University of California, Berkeley
Benedict Gaster, AMD
Katherine Hartsell, Oracle
Tom Murphy, Contra Costa College
Steven Parker, NVIDIA
Charlie Peck, Earlham College
Jennifer Teal, Intel

Special thanks to Abi Sundaram, Intel

Table of Contents

PARALLELISM COMPUTING
JOURNEYMAN’S PROGRAMMING TOUR

INTRODUCTION

In every domain the tools that allow us to tackle the big problems, and execute the complex calculations that are necessary to solve them, are computer based. The evolution of computer architecture towards hardware parallelism means that software/computational parallelism has become a necessary part of the computer scientist and engineer’s core knowledge. Indeed, understanding and applying computational parallelism is essential to gaining anything like a sustained performance on modern computers. Going forward, performance computing will be even more dependent on scaling across many computing cores and on handling the increasingly complex nature of the computing task. This is true irrespective of whether the domain problem is predicting climate change, analyzing protein folding, or producing the latest animated block-buster.

The Parallelism Tech Pack is a collection of guided references to help students, practitioners, and educators come to terms with the large and dynamic body of knowledge that goes by the name “parallelism”. We have organized it as a series of tours; each tour in the tech pack corresponds to one particular guided route through that body of knowledge. This particular tour is geared towards those who have some reasonable skills as practitioners of serial programming but who have not yet really explored parallelism in any coherent way. All of the tours in the Parallelism Tech Pack are living documents that provide pointers to resources for the novice and the advanced programmer, for the student and the working engineer. Future tours within the Tech Pack will address other topics.

The authors of this Tech Pack are drawn from both industry and academia. Despite this group's wide variety of experiences in utilizing parallel platforms, interfaces, and applications, we all agree that parallelism is now a fundamental concept for all of computing.

Scope of Tour: This tour approaches parallelism from the point of view of someone comfortable with programming but not yet familiar with parallel concepts. It was designed to ease into the topic with some introductory context, followed by links to references for further study. The topics presented are by no means exhaustive. Instead, the topics were chosen so that a careful reader should achieve a reasonably complete feel for the fundamental concepts and paradigms used in parallel computing across many platforms. Exciting areas like transactional memory, parallelism in functional languages, distributed shared memory constructs, and so on will be addressed in other tours but also should be seen as building on the foundations put forth here.

Online Readings
Herb Sutter. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's J. 33,* 3 (March). http://www.gotw.ca/publications/concurrency-ddj.htm.

James Laurus. 2009. Spending Moore's dividend. *Commun. ACM 52*, 5 (May). http://doi.acm.org/10.1145/1506409.1506425.

1. THE BASICS OF PARALLEL COMPUTING

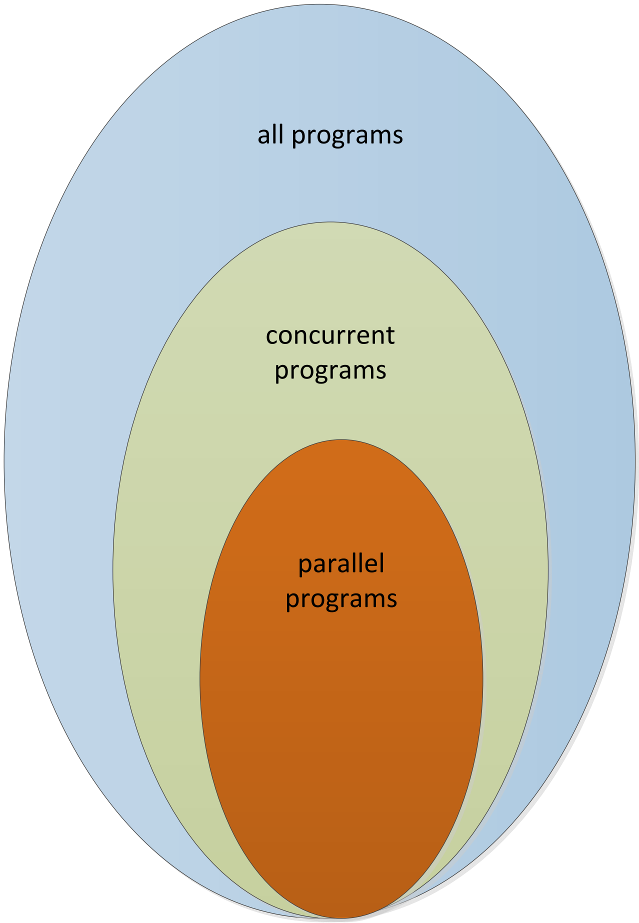
Parallelism is a property of a computation in which portions of the calculations are independent of each other, allowing them to be executed at the same time. The more parallelism exists in a particular problem, the more opportunity there is for using parallel systems and parallel language features to exploit this parallelism and gain an overall performance improvement. For example, consider the following pseudo code:

```
float a = E + A;
float b = E + B;
float c = E + C;
float d = E + D;
float r = a + b + c + d.
```

The first four assignments are independent of each other, and the expressions E+A, E+B, E+C, and E+D can all be calculated in parallel, that is, at the same time, which can potentially provide a performance improvement over executing them sequentially, that is, one at a time.

Parallel computing is defined as the simultaneous use of more than one processor to solve a problem, exploiting that program's parallelism to speed up its execution time.

***Is concurrency the same as parallelism*?** While concurrency and parallelism are related, they are not the same! Concurrency mostly involves a set of programming abstractions to arbitrate communication between multiple processing entities (like processes or threads). These techniques are often used to build user interfaces and other asynchronous tasks. While concurrency does not preclude running tasks in parallel (and these abstractions are used in many types of parallel programming), it is not a necessary component. Parallelism, on the other hand, is concerned with the execution of multiple operations in parallel, that is, at the same time. The following diagram shows parallel programs as a subset of concurrent ones, together forming a subset of all possible programs:



2. PARALLEL DECOMPOSITIONS

Introduction

There are a number of decomposition models that are helpful to think about when breaking *computation* into independent work. Sometimes it is clear which model to pick. At other times it is more of a judgment call, depending on the nature of the problem, how the programmer views the problem, and the programmer's familiarity with the available tool sets. For example, if you need to grade final exams for a course with hundreds of students, there are many different ways to organize the job with multiple graders so as to finish in the shortest amount of time.

Tutorials

The EPCC center at Edinburgh has a number of good tutorials. The tutorials most useful in this context are the following.

"Introduction to High Performance Computing" and "Decomposing the Potentially Parallel". http://www2.epcc.ed.ac.uk/computing/training/document_archive/.

Blaise Barney. "An Introduction to Parallel Computing". Lawrence Livermore National Labs. https://computing.llnl.gov/tutorials/parallel_comp/.

Videos
Introduction to Parallel Programming Video Lecture Series: Part 02 . "Parallel Decomposition Methods".

This video presents three methods for dividing composition, and pipelining. http://software.intel.com/enus/courseware/course/view.php?id=381.

Introduction to Parallel Programming Video Lecture Series: Part 04. "Shared Memory Considerations".

This video provides the viewer with a description of the shared-memory model of parallel programming. Implementation strategies for domain decomposition and task decomposition problems using threads within a shared memory execution environment are illustrated. http://software.intel.com/enus/courseware/course/view.php?id=249.

Task decomposition, sometimes called functional decomposition, divides the problem by the type of task to be done and then assigns a particular task to each parallel worker. As an example, to grade hundreds of final exams, all test papers can be piled onto a table and a group of graders can each be assigned a single question or type of question to score, which is the task to be executed. So one grader has the task of scoring all essay questions, another grader would score the multiple-choice questions, and another would score the true/false questions.

Videos
Introduction to Parallel Programming Video Lecture Series: Part 09. "Implementing a Task Decomposition".
http://software.intel.com/en-us/courseware/course/view.php?id=378.

This video describes how to design and implement a task decomposition solution. An illustrative example for solving the 8 Queens problem is used. Multiple approaches are presented with the pros and cons for each described. After the approach is decided upon, code modifications using OpenMP are presented. Potential data trace errors with a shared stack data structure holding board configurations (the tasks to be processed) are offered and a solution is found and implemented.

Data decomposition, sometimes called domain decomposition, divides the problem into elements to be processed and then assigns a subset of the elements to each parallel worker. As an example, to grade hundreds of final exams all test papers can be stacked onto a table and divided into piles of equal size. Each grader would then take a stack of exams and grade the entire set of questions.

Tutorials
Blaise Barney. "An Introduction to Parallel Computing". Lawrence Livermore National Lab. https://computing.llnl.gov/tutorials/parallel_comp/#DesignPartitioning.

Pipeline parallelism is a special form of task decomposition where the output from one process, or stage, as they are often called, serves directly as the input to the next process. This imposes a much more tightly coordinated structure on the program than is typically found in either plain task or data decompositions. As an example, to grade hundreds of final exams, all test papers can be piled onto a table and a group of graders arranged in a line. The first grader takes a paper from the pile, scores all questions on the first page and passes the paper to the second grader; the second grader receives a paper from the first grader and scores all the questions on the second page and passes the paper to the third grader, and so on, until the exam is fully graded.

3. PARALLEL HARDWARE

The previous section described some of the categories of parallel computation. In order to discuss parallel computing, however, we also need to address the way that computing hardware can also express parallelism.

Memory systems. From a very basic architecture standpoint, there are several general classifications of parallel computing systems:

In a ***shared memory system***, the processing elements all share a global memory address space. Popular shared-memory systems include multicore CPUs and many-core GPUs (Graphics Processing Unit).

In a ***distributed memory system***, multiple individual computing systems with their own memory spaces are connected to each other through a network.

These system types are not mutually exclusive. In ***hybrid systems***, in which modern computational clusters are classified, systems consist of distributed memory nodes, each of which is a shared memory system.

Processing characteristics. In a parallel application, calculations are performed in the same way they are in the serial case, on a CPU of some kind. However, in parallel computing there are multiple processing entities (tasks, threads or processes) instead of one. This results in a need for these entities to communicate values with each other as they compute. This ***communication*** happens across a network of some kind.

Coordination*,** such as managing access to shared data structures in a threaded environment, is also a form of communication. In either case, communication adds a ***cost to the runtime of a program, in an amount that varies greatly based on the design of the program. Ideally, parallel programmers want to minimize the amount of communication done (compared to the amount of computation).

Scalability*.** An important characteristic of parallel programs is their ability to scale, both in terms of the computing resources used by the program and the size of the data set processed by the program. There are two types of scaling we consider when analyzing parallel programs, strong and weak scaling. ***Strong scaling examines the behavior of the program when the size of the data set is held constant while the number of processing units increases**.** ***Weak scaling*** examines what happens when the size of the data set is increased proportionally as the number of processing units increases. Generally speaking, it is easier to design parallel programs that do well with weak scaling than it is to design programs that do well with strong scaling.

Heterogeneous architectures (e.g., IBM Cell Architecture, AMD's Fusion Architecture, and Intel's Sandy Bridge Architecture). Heterogeneous systems may consist of many different devices, each with its own capabilities and performance properties, all exposed within a single system. Such systems, while not new (embedded system on a chip designs have been around for over two decades), these architectures are becoming more prevalent in the mainstream desktop and supercomputing environments. This is due to the emergence of accelerators such as IBM Cell Broadband, and more recently the wide adoption of the General-purpose computing on graphics processing units (GPGPU) programming model, where CPUs and GPUs are connected to form a single system. NVIDIA's Compute Unified Device Architecture (CUDA) devices are the most common GPGPUs in use currently.

4. PARALLEL PROGRAMMING MODELS, LIBRARIES, AND INTERFACES

Introduction

This material is grouped by parallel programming model. The first section covers libraries and interfaces designed to be used in a shared memory model; the second covers tools for the distributed memory model; and the third covers tools for the GPGPU model. Another component of this tour covers hybrid models where two or more of these models may be combined into a single parallel application.

Shared memory model programming.

Posix threads are a standard set of threading primitives. Low-level threading method which underlies many of the more modern threading abstractions like OpenMP and TBB.

The following are some resources to assist you in understanding Posix threads better.

Tutorials
POSIX thread (pthread) libraries.
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

Books
David Butenhof. 1997. *Programming with POSIX Threads*, Addison-Wesley.
http://www.amazon.com/Programming-POSIX-Threads-David-Butenhof/dp/0201633922

This book offers an in-depth description of the IEEE operating system interface standard, POSIXAE (Portable Operating System Interface) threads, commonly called Pthreads. It's written for experienced C programmers, but assumes no previous knowledge of threads, and explains basic concepts such as asynchronous programming, the lifecycle of a thread, and synchronization. *Abbreviated Publisher's Abstract.*

Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. 1996. *Pthreads Programming: A POSIX Standard for Better Multiprocessing,* O'Reilly.
http://oreilly.com/catalog/9781565921153

POSIX threads, or pthreads, allow multiple tasks to run concurrently within the same program. This book discusses when to use threads and how to make them efficient. It features realistic examples, a look behind the scenes at the implementation and performance issues, and special topics such as DCE and real-time extensions. *Abbreviated Publisher's Abstract*.

Joy Duffy. 2008. *Concurrent Programming on Windows*, Addison-Wesley.
http://www.amazon.com/Concurrent-Programming-Windows-Joe-Duffy/dp/032143482X

This book offers an in-depth description of the issues with concurrency, introducing general mechanisms and techniques, and covering details of implementations within the .NET framework on Windows. There are numerous examples of good and bad practice, and details on how to implement your own concurrent data structures and algorithms.

Win32 threads**,** also called Native Threading by Windows developers, are still the default method used by many to introduce parallelism into code in Windows environments. Native threading can be difficult to implement and maintain. Microsoft has a rich body of material available on the Microsoft Developer Network.

Material that provides more information about threads follows.

Online Resources
Microsoft Developer Network. An online introduction to Windows threading concepts.
http://msdn.microsoft.com/en-us/library/ms684841(VS.85).aspx.

Books

Johnson M. Hart. 2010. *Windows System Programming* (4th ed.), Addison-Wesley. http://www.amazon.com/Windows-Programming-Addison-Wesley-Microsoft-Technology/dp/0321657748

This book contains extensive new coverage of 64-bit programming, parallelism, multicore systems, and other crucial topics. Johnson Hart's robust code examples have been debugged and tested in both 32-bit and 64-bit versions, on single and multiprocessor systems, and under Windows 7, Vista, Server 2008, and Windows XP. Hart covers Windows externals at the API level, presenting practical coverage of all the services Windows programmers need, and emphasizing how Windows functions actually behave and interact in real-world applications. *Abbreviated Publisher's Abstract.*

Java: Since version 5.0 concurrency support has been a fundamental component of the Java specification. Java follows a similar approach to POSIX and other threading APIs, introducing thread creation and synchronization primitives into the language as a high-level API, through the package java.util.concurrent. There are a number of approaches to introducing parallelism into Java code but conventionally they follow a standard pattern. Each thread is created as an instance of the class *Thread,* defining a class that implements the runnable interface (think of this as the POSIX entry point for a function), that must implement the method *public void run().* Just like POSIX, the thread is terminated when the method returns.

There are a large number of resources to help with understanding Java threads better, and the following is just a small selection.

Tutorials

Oracle has a large number of Java online tutorials, and one that introduces Java threads. http://download.oracle.com/javase/tutorial/essential/concurrency/index.html.

For the developer new to Java and or concurrency, the Java for Beginners portal provides an excellent set of tutorials and one that is specifically on the threading model. http://www.javabeginner.com/learn-java/java-threads-tutorial.

Books

Scott Oaks and Henry Wong. 2004. *Java Threads*, O'Reilly. http://oreilly.com/catalog/9780596007829

This is a well-developed book that, while not the most up-to-date resource on the subject, provides an excellent reference guide.

OpenMP**:** OpenMP is a directive-based, shared memory parallel programing model. It is most useful for parallelizing independent loop iterations in both C and Fortran. New facilities in OpenMP 3.0 allow for independent tasks to execute in parallel. To be used, OpenMP must be supported by your compiler. The standard is limited in scope on the types of parallelism that you can implement, but it is easy to use and a good starting point for learning parallel programming. Some resources to help with understanding OpenMP better are listed below.

Tutorials
Blaise Barney. OpenMP Tutorial, Lawrence Livermore National Lab. https://computing.llnl.gov/tutorials/openMP/.

This excellent tutorial is geared to those who are new to parallel programming with OpenMP. Basic understanding of parallel programming in C/C++ or FORTRAN is assumed.

OpenMP exercises. Tim Mattson and Larry Meadows. Intel Corporation.
This tutorial provides an excellent introduction to OpenMP, including code and examples. http://openmp.org/mp-documents/OMP_Exercises.zip and http://openmp.org/mp-documents/omp-hands-on-SC08.pdf.

Getting Started with OpenMP. Text-based tutorial; read and learn with examples. http://software.intel.com/enus/articles/getting-started-with-openmp/ .

An Introduction to OpenMP 3.0**.** This deck contains more advanced techniques (e.g., inclusion of wait statements) that would need more explanation to be used safely. https://iwomp.zih.tu-dresden.de/downloads/2.Overview_OpenMP.pdf.

Videos
An Introduction to Parallel Programming: Video Lecture Series. http://software.intel.com/en-us/courseware/course/view.php?id=224.

This multipart introduction contains many units on OpenMP, and includes coding exercises and code samples.

Community sites
www.openmp.org. Contains the current and past OpenMP language specifications, lists of compilers that support OpenMP, references, and other resources.

Cheat sheet
FORTRAN and C/C++.
C++: http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf.
FORTRAN: http://www.openmp.org/mp-documents/OpenMP3.0-FortranCard.pdf.

Books

Barbara Chapman, Gabriele Jost, and Ruud van der Pas, 2007. *Using OpenMP, Portable Shared Memory Parallel Programming*, MIT Press.
ACM Members, read it here:
http://learning.acm.org/books/book_detail.cfm?isbn=9780262533027&type=24.

Michael J. Quinn, 1004. *Parallel Programming in C with MPI and OpenMP*, McGraw Hill.

This book addresses the needs of students and professionals who want to learn how to design, analyze, implement, and benchmark parallel programs in C using MPI and/or OpenMP. It introduces a design methodology with coverage of the most important MPI functions and OpenMP directives. It also demonstrates, through a wide range of examples, how to develop parallel programs that will execute efficiently on today's parallel platforms. *Abbreviated Publisher's Abstract.*

Breshears. 2009. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*, O'Reilly. http://oreilly.com/catalog/9780596521547

This book contains numerous examples of applied OpenMP code.
Written by an Intel engineer with over two decades of parallel and concurrent programming experience, *The Art of Concurrency* is one of the few resources to focus on implementing algorithms in the shared-memory model of multicore processors, rather than just theoretical models or distributed-memory architectures. The book provides detailed explanations and usable samples to help you transform algorithms from serial to parallel code, along with advice and analysis for avoiding mistakes that programmers typically make when first attempting these computations.

Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. 2001. *Parallel Programming in OpenMP*, Morgan-Kaufmann.

Aimed at the working researcher or scientific C/C++ or Fortran programmer, *Parallel Programming in OpenMP* both explains what the OpenMP standard is and how to use it to create software that takes full advantage of parallel computing. By adding a handful of compiler directives (or pragmas) in Fortran or C/C++, plus a few optional library calls, programmers can "parallelize" existing software without completely rewriting it. This book starts with simple examples of how to parallelize "loops"—iterative code that in scientific software might work with very large arrays. Sample code relies primarily on Fortran (the language of choice for high-end numerical software) with descriptions of the equivalent calls and strategies in C/C++. *Abbreviated Publisher's Abstract.*

Threading Building Blocks (TBB). Intel® Threading Building Blocks (Intel® TBB. TBB is a threading library used to introduce parallelism into C/C++. TBB is a relatively easy way to introduce loop-level parallelism, especially for programmers familiar with templated code. TBB is available both as an open source project and as commercial product from the Intel Corporation.

Tutorials
Intel Threading Building Blocks Tutorial.
http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Tutorial.pdf.
Written by Intel Corporation, this is a thorough introduction to the threading library. "This tutorial teaches you how to use Intel® Threading Building Blocks (Intel® TBB), a library that helps you leverage multicore performance without having to be a threading expert ".

Multicoreinfo.com brings together a number of TBB tutorials.
http://www.multicoreinfo.com/2009/07/parprog-part-6/.

Code examples
This TBB.org website contains a thorough set of coding examples.
http://www.threadingbuildingblocks.org/codesamples.php.

Code examples from a recent coding with TBB contest.
http://software.intel.com/en-us/articles/coding-with-intel-tbb-sweepstakes/.

Community sites
Contains product announcements (releases and updates), links to code samples, blogs, and forums on TBB. http://www.threadingbuildingblocks.org.

Intel site for commercial version of Intel Threading Building Blocks.
http://www.threadingbuildingblocks.com.

Books
James Reinders. 2007. *Intel Threading Building Blocks,* O'Reilly.
http://oreilly.com/catalog/9780596514808

This guide explains how to maximize the benefits of multicore processors through a portable C++ library that works on Windows, Linux, Macintosh, and Unix systems. With it, you'll learn how to use Intel Threading Building Blocks (TBB) effectively for parallel programming, without having to be a threading expert. Written by James Reinders, Chief Evangelist of Intel Software Products, and based on the experience of Intel's developers and customers, this book explains the key tasks in multithreading and how to accomplish them with TBB in a portable and robust manner. *Abbreviated Publisher's Abstract.*

Breshears. 2009. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*, O'Reilly. http://oreilly.com/catalog/9780596521547
This book contains numerous examples of applied TBB codes.

***Distributed memory model programming*.** The preceding libraries and interfaces assume that the results from one thread of the overall computation can be made directly available to any other thread. However, some parallel hardware (such as clusters) forbid direct access from one memory space to another. Instead, processes must cooperate by sending each other messages containing the data to be exchanged.

***Message-passing interface (MPI)*.** MPI is a library specification that supports message passing between program images running on distributed memory machines, typically clusters of some type. A number of different organizations develop and support implementations of the MPI standard, which specifies interfaces for C/C++ and FORTRAN. However, bindings for Perl, Python, and many other languages also exist. MPI provides routines that manage the transmission of data from the memory space of one process to the memory space of another process. Distributed memory machines require the use of MPI or another message-passing library by the parallel program in order to use multiple processes running on more than one node.

Getting started: Start with the six basic commands:

MPI_Init() - Initialize the MPI world

MPI_Finalize() - Terminate the MPI world

MPI_Comm_rank() - Which process am I ?

MPI_Comm_size() - How many processes exist ?

MPI_Send() - Send data

MPI_Recv() - Receive data

Move on to more complex communication models as needed, that is, to collective communication (one->many, many->one, many->many); and or advanced communication techniques: synchronous vs asynchronous communication, blocking vs nonblocking communication.

Online Readings
Moodles with slides and code examples. NCSI parallel and distributed workshop. http://moodle.sc-education.org/course/category.php?id=17.

Tutorials
William Gropp, Rusty Lusk, Rob Ross, and Rajeev Thakur. 2005. Advanced MPI: I/O and One-Sided Communication. http://www.mcs.anl.gov/research/projects/mpi/tutorial/.

Super Computing in Plain English (SIPE).
http://www.oscer.ou.edu/Workshops/DistributedParallelism/sipe_distribmem_20090324.pdf.

Cheatsheets
http://wiki.sc-education.org/index.php/MPI_Cheat_Sheet.

Books
Peter Pacheco. 1997. *Parallel Programming with MPI. http://www.amazon.com/Parallel-Programming-MPI-Peter-Pacheco/dp/1558603395*

A hands-on introduction to parallel programming based on the Message-Passing Interface (MPI) standard, the de facto industry standard adopted by major vendors of commercial parallel systems. This textbook/tutorial, based on the C language, contains many fully developed examples and exercises. The complete source code for the examples is available in both C and Fortran 77. Students and professionals will find that the portability of MPI, combined with a thorough grounding in parallel programming principles, will allow them to program any parallel system, from a network of workstations to a parallel supercomputer. *Abbreviated Publisher's Abstract.*

***General-purpose GPU programming*.** In contrast to the threading models presented earlier, accelerator-based hardware parallelism (like GPUs) focuses on the fact that although results may be shareable, that is, can be sent from one part of a computation to another, the cost of accessing the memory may not be uniform; CPUs see CPU memory better than GPU memory, and vice versa.

***The Open Compute Language*.** (OpenCL) is an open standard for heterogeneous computing, developed by the Khronos OpenCL working group. Implementations are currently available from a broad selection of hardware and software vendors, including AMD, Apple, NVIDIA, and IBM.

OpenCL is intended as a low-level programming model, designed around the notion of a host application, commonly a CPU, driving a set of associated compute devices, where parallel computations can be performed.

A key design feature of OpenCL is its use of asynchronous command queues, associated with individual devices, that provide the ability to enqeue work (e.g., data transfers and parallel code execution), and build up complex graphs describing the dependencies between tasks.

The execution model supports both data-parallel and task-parallel styles, but OpenCL was developed specifically with an eye toward today's many-core, throughput, GPU-style architectures, and hence exposes a complex memory structure that provides a number of levels of software-managed memories. This is in contrast to the traditional single-address space model of languages like C and C++, backed by large caches on general-purpose CPUs.

The OpenCL standard is currently in its second iteration, at version 1.1, and includes both a C API, for programming the host, and a new C++ Wrapper API, added to 1.1 and intended to be used for OpenCL C++ development. By exposing multiple address spaces, OpenCL provides a very powerful programming model to access the full potential of many-core architectures, but this comes at the cost of abstraction!

This is particularly true in the case of performance portability, and it is often difficult to achieve good performance on two different architectures with the same source code. This can be even more evident between the different types of OpenCL devices (e.g., GPUs and CPUs). This should not come as a surprise, as the OpenCL specification itself states that it is a low-level programming language, and given that these devices can have very different types of compute capabilities, careful tuning is often required to get close to peak performance.

Online Resources
OpenCL 1.1 Specification (revision 33, June 11, 2010).
http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

OpenCL 1.1 C++ Wrapper API Specification (revision 4, June 14, 2010).
http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf

OpenCL 1.1 Online Manual Pages.
http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/

OpenCL Quick Reference Card. http://www.khronos.org/opencl/.

Tutorial
An excellent beginners' "hello world" tutorial introduction using OpenCL 1.1's C++ API.
http://developer.amd.com/GPU/ATISTREAMSDK/pages/TutorialOpenCL.aspx

Videos
ATI Stream OpenCL Technical Overview Video Series.
http://developer.amd.com/DOCUMENTATION/VIDEOS/OPENCLTECHNICALOVER

VIEWVIDEOSERIES/Pages/default.aspx

This five-part video tutorial series provides an excellent introduction to the basics of OpenCL, including its execution and memory models, and the OpenCL C device programming language.

Community sites

The Khronos Group's main page, http://www.khronos.org, keeps track of major events around OpenCL and its other languages such as OpenGL and WebGL, along with some useful discussion forums on these. http://www.khronos.org/opencl.

The websites http://www.beyond3d.com and Mark Harris' http://www.gpgpu.org are full of information about many-core programming, in particular the modern GPUs of AMD and NVIDIA, and provide vibrant discussions on OpenCL and CUDA (the details of this language will follow), among other interesting areas and topics.

There is an ever-growing set of examples that can be found all over the web, and each of the major vendors provides excellent examples with their corresponding SDKs.

CUDA. Compute Unified Device Architecture (CUDA) is NVIDIA's parallel computing architecture that enables acceleration in computing performance by harnessing the power of the GPU (graphics processing unit). CUDA is in essence a data-parallel model, sharing a lot in common with the other popular GPGPU language OpenCL, where kernels (similar to functions) are executed over a 3D iteration space; each index is executed concurrently, possibly in parallel.

Online Resources

NVIDIA maintains a collection of featured tutorials, presentations, and exercises on the CUDA Developer Zone. http://developer.nvidia.com/object/cuda_training.html.

Online Readings

For details on NVIDIA CUDA hardware and the underlying programming models, the following articles are relevant:

Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A unified graphics and computing architecture, *IEEE Micro 28*, 2 (March), 39–55.

NVIDIA GF 100. 2010. http://www.nvidia.com/object/IO_89569.htm. (White paper.)

Code examples

The CUDA SDK includes numerous code examples, along with CUDA versions of

popular libraries (cuBLAS and cuFFT).
http://developer.nvidia.com/object/cuda_3_1_downloads.html

Books
David B. Kirk and Wen-Mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann.
http://www.amazon.com/dp/0123814723?tag=wwwnvidiacomc-20&camp=14573&creative=327641&linkCode=as1&creativeASIN=0123814723&adid=1DT2S034DXS37V3K5FFY.

This is a recent and popular textbook for teaching CUDA. This book shows both student and professional alike the basic concepts of parallel programming and GPU architecture. Various techniques for constructing parallel programs are explored in detail. Case studies demonstrate the development process, which begins with computational thinking and ends with effective and efficient parallel programs. *Abbreviated Publisher's Abstract.*

***Hybrid parallel software architectures*.** Programs which use a hybrid parallel architecture combine two or more libraries/models/languages (see Section 3) into a single program; the motivation for this extra complexity is to allow a single parallel program image to harness additional computational resources.

The most common forms of hybrid models combine MPI with OpenMP or MPI with CUDA. MPI/OpenMP is appropriate for use on cluster resources where the nodes are multicore machines; MPI is used to move data and results among the distributed memories, and OpenMP is used to leverage the compute power of the cores on the individual nodes. MPI/CUDA is appropriate for use on cluster resources where the nodes are equipped with NVIDIA's GPGPU cards. Again, MPI is used to move data and results among the distributed memories and CUDA is used to leverage the resources of each GPGPU card.

Online Resources
MPI/OpenMP: The Louisiana Optical Network Initiative (LONI) has a nice tutorial on building hybrid MPI/OpenMP applications. It can be found at https://docs.loni.org/wiki/Introduction_to_Programming_HybridApplications_UsingOpenMP_and_MPI. This includes pointers to LONI's OpenMP as well as MPI tutorials.

MPI/CUDA: The National Center for Supercomputing Applications (NCSA) has a tutorial that includes information about this; see the section "Combining MPI and CUDA" in
http://www.ncsa.illinois.edu/UserInfo/Training/Workshops/CUDA/presentations/tutorial-CUDA.html.

Parallel Languages

We touch here only briefly on the topic of inherently parallel languages. There are a variety of efforts, ranging from extensions to existing languages to radically new approaches. Many, such as Cilk or UPC, can be relatively easily understood in terms of the libraries and techniques described above. A fuller discussion of the variety of language efforts and tools will be in a further Tech Pack. Because it is sufficiently different, however, a quick look at how parallelism is incorporated into functional languages is helpful.

Concurrent ML and Concurrent Haskell: Functional programming languages, such as Standard ML and Haskell, provide strong foundation for building concurrent and parallel programming abstractions, for the single reason that they are declarative. Declarative in a parallel world, i.e. avoiding the issues (e.g. race conditions) that updating a global state for a shared memory model can cause, provides for a strong foundation to build concurrency abstractions.

Concurrent ML is a high-level message-passing language that supports the construction of first-class synchronous abstractions called events, embedded into Standard ML. It provides a rich set of concurrency mechanisms built on the notion of spawning new threads that communicate via channels.

Concurrent Haskell is an extension to the functional language Haskell for describing the creation of threads, that have the potential to execute in parallel with other computations. Unlike Concurrent ML, Concurrent Haskell provides a limited form of shared memory, introducing MVars ("mutable variables") which can be used to atomically communicate information between threads. Unlike more relaxed shared memory models (e.g. see OpenML and OpenCL in the following text), Concurrent Haskell's runtime system ensures that the operations for reading from and writing to MVars occur atomically.

Tutorials

Simon Peyton Jones and Satnam Singh. A Tutorial on Parallel and Concurrent Programming in Haskell Lecture Notes from Advanced Functional Programming Summer School 2008.
http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/AFP08-notes.pdf

Books

John H. Reppy. 1999. *Concurrent ML*, Cambridge University Press.

Simon Peyton Jones. 2007. Beautiful Concurrency. In *Beautiful Code*; edited by Greg Wilson, O'Reilly. http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/index.htm#beautiful

5. TOOLS

There is a variety of tools available to assist programmers in creating, debugging, and running parallel codes. This section summarizes the categories of tools; a more exhaustive list of tools that run on different hardware and software platforms will be included in a subsequent addition to the Tech Pack.

Compilers

Many of the compiler families today, both commercial and open-source, directly support some form of explicit parallelism. (OpenMP, threads, etc.).

Auto-parallelization

The holy grail of many, core support would be a compiler that could automatically extract parallelism at compile time. Unfortunately, this is still a work in progress. That said, a number of compilers can add utility through vectorization and the identification of obvious parallelism in simple loops.

Online Resources
http://en.wikipedia.org/wiki/Automatic_parallelization.

Thread debuggers

Intel Thread Checker: http://software.intel.com/en-us/intel-thread-checker/.

Intel Parallel Inspector: http://software.intel.com/en-us/intel-parallel-inspector/.

Microsoft Visual Studio 2010 tools: http://www.microsoft.com/visualstudio/en-us/.

Hellgrind. http://valgrind.org/docs/manual/hg-manual.html is a Valgrind tool for detecting synchronization errors in C, C++ and FORTRAN programs that use the POSIX pthreads threading primitives. The main abstractions in POSIX pthreads area set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Tuners/performance profilers

Intel® VTune™ Performance Analyzer & Intel® Thread Profiler 3.1 for Windows. The Thread Profiler component of Vtune helps tune multithreaded applications for performance. The Intel Thread Profiler timeline view shows what the threads are doing and how they interact. http://software.intel.com/en-us/intel-vtune/.

Intel® Parallel Amplifier. A tool to help find multicore performance bottlenecks without needing to know the processor architecture or assembly code. http://software.intel.com/en-us/intel-parallel-amplifier/.

Microsoft Visual Studio 2010 tools. http://www.microsoft.com/visualstudio/en-us/.

gprof: the GNU Profiler. http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html.

Memory tools

Hoard: http://www.hoard.org/. The Hoard memory allocator is a fast, scalable, and memory-efficient memory allocator. It runs on a variety of platforms, including Linux, Solaris, and Windows. Hoard is a drop-in replacement for malloc() that can *dramatically improve application performance, especially for multithreaded programs running on multiprocessors.* No change to your source is necessary. Just link it in or set just one environment variable.