A Training Roadmap for New HPC Users

Mark Richards and Scott Lathrop
National Center for Supercomputing Applications
Urbana, IL 61801
mdrichar,scott@ncsa.illinois.edu

ABSTRACT

Many new users of TeraGrid or other HPC resources are scientists or other domain experts by training and are not necessarily familiar with core principles, practices, and resources within the HPC community. As a result, they often make inefficient use of their own time and effort and of the computing resources as well. In this work, we present a training roadmap for new members of the HPC community. These new users benefit from a basic understanding of the the key concepts, technologies, and tools outlined in the roadmap, even if they are not required to develop immediate proficiency in all of them. In addition to providing a brief overview of a wide variety of topics, the roadmap includes links and pointers to numerous resources that users can refer to for their own training as the need arises.

Categories and Subject Descriptors

K.3.2 [**Computing and Education**]: Computer and Information Science Education—*computer science education, curriculum*

General Terms

Human Factors

Keywords

Training, Education

1. INTRODUCTION

High Performance Computing (HPC) may be viewed as the intersection between the frontiers of science and computing. Many HPC users have or are pursuing formal education in a scientific or engineering discipline and are self-taught programmers. Others may have a background in computer science where their primary training in principles of parallel programming comes from an introductory operating systems course. Few, if any, come to the HPC community with formal training in every area required for success.

A typical entry point is as a graduate student. An advisor requests that a student dig up a previous student's Fortran code and get it running on a supercomputer. For the new student, who is competent in solving small systems of partial differential equations and generating meaningful visualizations of the output in Matlab, the concept of compiling a Fortran program from source on a remote machine, submitting a batch job to run overnight, and then figuring out a way to process gigabytes of data and transfer it back to a personal machine for further analysis may be daunting. After eventually succeeding to run the existing code, the new student and faculty advisor may identify ways to extend the algorithm to handle new or larger datasets. This process may require writing new code in Fortran, using profiling tools to identify bottlenecks, employing optimization techniques to improve runtime efficiency, and parallel visualization tools to help understand the output. All of these requirements will likely stretch the student's capabilities.

Some users may experience frustration because of the need to use a command-line interface, deal with nebulous error messages from the compiler or linker, or grapple with unexpected numerical issues. Others are surprised and perplexed to find that after expending significant effort to parallelize an existing code, performance is actually worse on a parallel machine than on a single processor. Still others may be confused about what factors should inform their decision as to whether to target a code for use with GPGPUs or multi-core machines, whether to use Fortran or C, etc. Some may waste valuable time and effort implementing a basic numerical algorithm from scratch, unaware that optimized libraries for such tasks have been written by others and are freely available.

In many cases, useful explanations, tutorials, software tools, or numerical libraries are available online, but the users are simply unaware of them. They do not know what they do not know. If they only new the right keywords to type into a search engine, they might be able to find resources to help solve their problem relatively quickly. Instead, they spend hours retreading the mistakes of others or reinventing the wheel.

To this end, we have begun development of a training roadmap for new HPC users. The goal is to provide novices with a broad overview of the most basic principles, practices, tools, and technologies of HPC as soon as they get an account on an HPC system. We provide a web-based flowchart that outlines basic vocabulary and concepts related to HPC. Each node in the flowchart is hyperlinked to a brief textual description and collection of links to relevant online courses and tutorials, software tools, training videos, books, and papers that provide additional in-depth information about the topic. The goal is not to supplant formal training and official curricula but rather to make new users aware of quality resources that are more immediately accessible.

In the next section we outline the training roadmap. We then give a short overview of HPC vocabulary and principles that new users might reasonably consume in a single sitting.

2. TRAINING ROADMAP

Figure 1 shows a flowchart of some basic concepts and skills that are relevant to the HPC community. Not all skills are necessary for all users, but it is helpful for all users to understand the overall landscape. Many new users are first exposed to the world of high performance computing because of a specific need or application, and they may not appreciate the additional challenges that present themselves when applications are scaled to state-of-the-art systems. By familiarizing themselves with the roadmap early on, they will come to appreciate what they do not know. While the details of many of the topics may in fact not be immediately relevant to their work, our hope is that when they eventually come up against the challenges, questions, error messages, and bugs that are common to the HPC experience, they will remember that the roadmap exists and will return to find pointers to resources that will help them.

The initial entry point in the roadmap is a collection of basic HPC concepts. These include a basic understanding of computer architectures, including the architecture of supercomputers. Additionally, users should understand principles of parallelization, efficiency, scalability, and the numerical issues that arise when dealing with floating point operations. Not all HPC users will need to program. Some may have or develop expertise in a particular application domain and may need to run applications on HPC machines but not necessarily modify source code. These users can still benefit from an understanding of the visualization tools that are available and of the tools and practices for manipulating large datasets.

Some users are accustomed to graphical user interfaces and will benefit by developing some familiarity with UNIX-style operating systems and the use of command-line interfaces and tools. Most HPC users will eventually need to write some code. Before delving deep into parallel software with all of its complexities, users should learn some basic programming skills in C or Fortran. They should also understand some best practices from the field of software engineering, particularly with respect to code structure, documentation, and testing.

Once users have developed basic programming skills, they can learn about more advanced tools involved in the software life cycle, including debuggers, code profilers, and third-party libraries. Some new HPC users may be experienced programmers who are familiar with such tools but will need to learn the additional complexities of debugging, profiling, and optimizing parallel codes. Before writing parallel programs, users should understand principles of shared memory and distributed memory systems so that they can make informed decisions about which avenues to pursue. Finally, the roadmap includes references to a variety of parallel programming languages and technologies.

The roadmap is hosted on a `hpcuniversity.org`. When users click on a node in the flowchart, they are taken to a brief description of the topic and a list of quality resources that they can utilize to learn more. The resources include professionally developed online tutorials (such MPI How-Tos), self-study courses, and videos of keynote addresses or live tutorials or lectures from NCSA's past Virtual Summer School for Computational Science and Engineering. Some of the links are simply pointers to the home pages of projects for tools like TAU, PerfSuite, or ATLAS. These are resources that new users would easily find themselves if they only knew that such tools existed and knew what terms to type into a search engine. Our hope is to increase awareness of such tools by providing this broad overview of the HPC space *before* the need for them arises.

3. HPC OVERVIEW

This section provides a broad overview of HPC that we believe new users might reasonably read through in one sitting soon after receiving access to an HPC system. Many important details are omitted for the sake of brevity, as the goal is to introduce new vocabulary and concepts rather than to promote mastery or a real depth of understanding in any particular area.

3.1 Fundamental HPC Concepts

The term *high performance computing* refers to the use of the most advanced computing resources to solve computationally intensive problems at the frontiers of science and engineering. As technology evolves, the processing power of today's supercomputers will become available in the mass-produced systems of tomorrow. Although the number of computations per second possible on the most advanced machines continues to increase rapidly, the key concepts and performance considerations for the high-end scientific and engineering applications tend to evolve more slowly over time.

On the one hand, increasing computational capabilities allow users to perform computations in minutes that used to take weeks or months on older machines. But commonly, the increased power is used to solve more challenging problems or run higher-fidelity problems in the same amount of time. For example, a weather forecasting tool may run simulations of cloud movement and behavior over a large geographical area. The simulation might model various attributes such as temperature, pressure, wind speed, and precipitation levels at each point on a grid. If the goal is to predict tomorrow's weather using data collected from atmospheric sensors over the last few hours or days, the simulation must necessarily run in at most a few hours. The results cannot be used to forecast tomorrow's weather if they are not available until the day after tomorrow. Both the time required to perform the computations and the accuracy of the predictions are related to the granularity of the grid used in the model. With a finer grid, accuracy improves but the simulation takes longer. As computers become faster, it makes sense to use the additional power to refine the grid and improve the accuracy, rather than to simply perform the same calculations faster.

Accuracy, speed, and efficiency are all key metrics in high performance computing. They are closely related.

3.1.1 Architectures

The "brain" of a basic computer is its central processing unit (CPU), which performs elementary arithmetic opera-

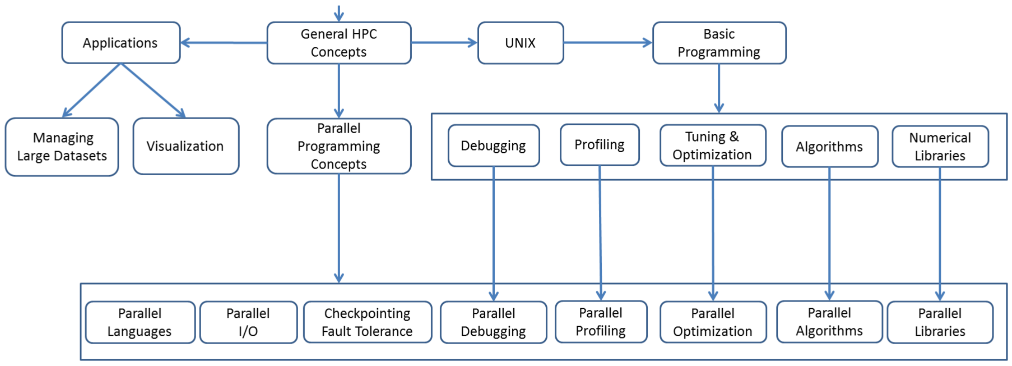

Figure 1: Flowchart for HPC Training Roadmap

tions (addition, subtraction, multiplication, division) and logical operations (AND, OR, NOT). Other basic mathematical operations like exponentials and trigonometric functions are reduced to or approximated by these simpler arithmetic operations. The maximum number of operations that can be performed in a second depends on the clock speed (measured in hertz [Hz], or cycles per second) and the number of independent functional units. Modern CPUs are often capable of performing more than one instruction per clock cycle, but achieving this efficiency requires careful programming.

The physical hardware that performs these operations typically requires the operand(s) to reside in the processor's *registers* or to be encoded in the instructions themselves. The operands may need to be retrieved from cache, main memory, disk, or the network. These elements form a natural *storage hierarchy* that is structured based on the inherent time/cost tradeoffs involved. The closer the data is to the functional units, the faster it can be accessed. Data already in registers can be used in an operation on the next clock cycle. Cache is typically organized into two or three levels of increasing size. Data in the lowest level (often called L1) can be retrieved (copied to a register) in a few (perhaps less than 10) cycles. Higher levels of cache (L2 and L3) are larger—typically a few megabytes—and require perhaps tens of cycles to access. Data from memory may require hundreds of cycles. Disks may have latencies on the order of milliseconds, several orders of magnitude slower than memory.

Increasing size and distance from the processor are also associated with a lower cost per bit. A $100 processor may have on the order of a few thousand bits of registers, where a $100 hard drive could have tens of trillions of bits of storage. A key principle in computing that is particularly prominent in many HPC applications is *locality*. Computations with high *temporal locality* reuse the same data multiple times within a short interval. *Spatial locality* refers to the idea that an application that uses any particular datum is likely to use data that are physically close to that datum soon. For example, since iterating over the elements of an array is a common process, a program that accesses an element at address a is likely to also access address $a+1$ and $a+2$ in the near future. These guiding principles inform the way computers are designed and the way software is written. A key to achieving good performance on a particular system is to order computations to minimize the time that the processor must wait for the data it needs.

A computer's operating system (OS) is the software that manages applications' access to system resources, including memory, disks, and networks. Operating systems manage the programs that run on the system as *processes*. A process includes an allocation of a block of memory to store information about a sequence of instructions to execute, including the instructions themselves, the call stack (i.e., state of active functions or subroutines), and the program's data. A process also maintains file descriptors for accessing resources on disk and a context that includes values of *environment variables* (used to communicate information about the filesystem, hardware, or operating system to the program) and permissions (security parameters for accessing system resources). Some processes may include multiple *threads*, or streams of instructions that can execute concurrently. The threads of a process maintain their own call stacks and instruction pointers but share other properties (memory space, environment, file descriptors) of the process.

The preceding discussion of storage hierarchy and processes applies to many modern systems. High performance computer systems are built by aggregating CPUs, memory, disk, and network resources. A collection of otherwise independent commodity systems connected by an Ethernet network is often referred to as a *cluster*. The term *supercomputer* may refer to a more tightly coupled system with higher-end interconnectivity, such as Myrinet or InfiniBand. In practice, the terms are often used interchangeably.

The term *processor* historically has been used to refer to a single processing *core*: a CPU and its associated cache. A single processor was fabricated on a single chip and connected to a socket on a motherboard, which provides the physical interface to memory, disks, and networks. As semiconductor technologies have improved, an increasing number of transistors can be fabricated on a single chip, allowing for more registers and functional units on a processor. Increasing transistor density facilitates higher clock speeds. But higher clock speeds means higher power consumption and the need to dissipate more and more heat (to prevent the processor from overheating). This has caused the clock rates of commodity processors to plateau in the 2–3 GHz range. Due to this limitation of clock speed, hardware vendors have leveraged the increase in transistor density to put more processing cores onto a single silicon chip. These are referred to as *multi-core processors*. The operating system can assign processes to run concurrently on different cores. At the hardware level, each processing core typically includes a set of registers, functional units, and first level cache. The processing cores on a single chip may share higher levels of cache. In modern usage, the word processor may refer to the individual cores or to the whole set of cores on the single chip. The term *node* refers to a physical memory device together with the processors/cores that can access that memory directly. Supercomputers comprise multiple nodes, together with the network interconnect and disk drives that serve them.

A *many-core* system typically has hundreds of cores. These cores may not have the full functionality of the cores in traditional multi-core systems. For example, the hundreds of cores on a general-purpose graphics processing unit (GPGPU) may work synchronously to apply the same instructions to many blocks of data at the same time, rather than being able to run arbitrary instructions independent of the other cores.

As mentioned before program performance degrades drastically whenever processing elements are forced to wait for data to arrive from higher levels in the storage hierarchy. This problem is exacerbated in HPC systems when data must be transferred over the network interconnect. Two key concepts related to these communication bottlenecks are *latency* and *bandwidth*. These metrics are critical to determining the efficiency of different types of communication between processors; between a processor and its cache, memory, or disk; or between nodes on a network.

When one entity requests data from another, we refer to the requester as the *client* and the supplier as the *server*. Latency refers to the minimum amount of time that it takes for a single piece of data to travel from source to destination. Data transfers in computer systems may approach the speed of light, but even this represents a bottleneck given the clock rate of modern processors. Latency includes the time it takes for the physical transfer of a request message from client to server, the processing time required by the server to retrieve the requested data, and the time for the physical transfer of the data back to the client. Latencies may vary from less than a nanosecond between a processor and its cache to microseconds between nodes with high-end interconnect to milliseconds between nodes with an Ethernet connection. Bandwidth refers to the maximum rate at which data is transferred (e.g., gigabits per second). An analogy to irrigation is often used. Latency is the amount of time it takes for the first drop of water to make it from the faucet to the end of the hose. Bandwidth is the maximum rate of flow of water through the hose.

For small and infrequent messages, latency tends to be the main concern. For large or frequent messages, bandwidth can be the bigger concern, particularly in cases where the communication medium is shared. For example, processor cores may share a connection to main memory. Nodes in a cluster or supercomputer also share their interconnect. This can create competition and bottlenecks.

A supercomputer or cluster network may be viewed as a graph, with vertices representing the compute nodes, edges representing the communication channel between pairs of nodes, and weights on the edges representing the bandwidth between nodes. Graph algorithms can be used to analyze communication properties of the network. The communication links between nodes can be arranged to produce a graph with desirable properties. A key design goal is often to minimize the maximum number of communication links or "hops" between any pair of nodes.

System designers analyze costs and the expected needs of applications to determine the best system architecture, which includes the number and type of processing cores in each node, and the network of connections between nodes. Once the system is built, software developers are responsible for designing their applications in a way that best utilizes the system architecture and topology.

3.1.2 Parallelism, Efficiency, and Scalability

HPC applications are characterized by a high degree of parallelism. This mean that many parts of the overall computation may be executed at the same time. If an application cannot be broken down into parallelizable chunks of computation, it will not be able to take advantage of HPC resources. Let $T(1)$ be the time taken to execute a program on a single processor, and let $T(n)$ be the time taken to execute the program on n processors. In this context, the term processor may refer to the number of processes or threads independently executing. The speedup

$$S(n) = \frac{T(1)}{T(n)}$$

quantifies the change in execution time as the number of processors increases. When this number is close to n, the parallelization is efficient. In some cases, $S(n)$ can actually be greater than n; this is called a superlinear speedup. Such speedups are possible when communication costs are low and the parallelization of the computation significantly improves the cache performance. More typically, $S(n)/n$ gets smaller as n gets larger. In other words, the parallelization becomes less efficient as the number of processors increases. The inefficiency can come from many sources.

Limited Potential. It could be that much of the application is inherently sequential. This occurs when the result of some part of the computation must be known before another part can proceed. If, for example, half of a program's computation is inherently sequential, the maximum speedup achievable is 2, even if the parallelizable half of the program is distributed among an infinite number of processors. This principle of limited potential for speedup is sometimes referred to as Amdahl's Law.

Parallel overhead. Parallelization incurs some overhead. Some of this is due to bookkeeping and start-up costs involved in launching new processes and/or threads. Additional overhead is incurred when data is passed between processors. Both latency and bandwidth issues can reduce performance.

Redundant computation. Sometimes, a processor can avoid waiting for a result computed by another processor by redundantly computing the result itself. This may in fact be a worthwhile tradeoff, but the additional computation required will still constrain the maximum achievable speedup.

Load imbalance. Whenever a processor allocated to a program sits idle, performance is reduced. Processors may idle while they wait for data from other processors. They may also idle while they wait to be assigned pieces of work to do. Load imbalance occurs when some processors have more work to do than others, which forces some processors to idle. When seeking to increase performance through improved load balancing, users should remember that the goal is not necessarily to minimize the variance in the amount of work allocated to processors but rather to minimize the maximum load on any single processor.

Speculative loss. In searching for the best solution to a combinatorial search problem, the search space may be framed as a tree. In some cases, large portions of the search tree may be pruned based on the results from previous parts of the search. When the search is parallelized, it could be that some effort is expended in exploring parts of the search tree that would be pruned during a sequential execution. This is referred to a speculative loss.

If it is determined that a large fraction of an instance of a particular problem is inherently sequential, it might seem pointless to go through the effort of parallelizing it. However, as noted above, when available computing power becomes available, the goal is not necessarily to run existing applications faster but to solve bigger problems in the same amount of time. Sometimes the fraction of the code that is inherently sequential is reduced as the size of the problem increases. In these cases, it may still be efficient to run the code on a larger number of processors. Applications that do not suffer from intolerable inefficiency as the problem size increases are said to be scalable. The *isoefficiency* metric is used to quantify how much the size of a problem must grow to maintain constant efficiency (ratio of speedup to processors) as the number of processors increases.

Another important metric for HPC applications is accuracy. One source of inaccuracy may come from limitations in the computational model. As mentioned above, one might expect a weather model based on sensors spaced at one-meter intervals to be more accurate than a similar model based on sensors spaced at 10 meters. Another source of inaccuracy comes from numerical issues. Many problems of interest involve real-valued data, but computers approximate real numbers using a finite number of bits. Non-integer values are often expressed as 32-bit or 64-bit floating point values. The total number of values that can be represented is therefore 2^{32} or 2^{64}. The values that can be represented are not evenly spaced on the real number line. Particularly notable is the fact that the number of values with absolute value less than 1 is approximately the same as the number of values with absolute values greater than 1. A practical consequence is that associative laws do not apply to floating point arithmetic. That is, $a + (b + c) \neq (a + b) + c$, in general. This can present problems in scientific problems run on sequential machines, but the problem is exacerbated by parallel processing, when the order of execution may vary from run to run. This may result in the same program unexpectedly producing different results at different times when executed on the same input.

Other common numerical problems are the result of overflow (when an operation produces a value larger in magnitude than the largest representable floating point value), underflow (values too close to zero), rounding issues (e.g., using the equality operator to test whether a variable with a theoretical value of one is equal to the floating point value of 1.0 may return false). The standards for floating point arithmetic include special values for positive and negative infinity (e.g. +inf, -inf). Performing ill-defined operations, such as $+inf / - inf$, may produce a result of NaN (not a number). Novice HPC users should seek to achieve a basic understanding of numerical issues, particularly those that can result from the unpredictable sequence in which instructions may be executed.

Other surprises involve the improper use of pseudo-random number generators. Many HPC simulations involve random numbers, but computers often use deterministic sequences of numbers that exhibit some of the desirable properties of true random numbers. The starting point of a pseudo-random number sequence is determined by a seed value which the user can select. If the user is not careful, it may be the case that all processors involved in a computation will use the same pseudo-random sequence with unexpected (and likely undesirable) results.

3.2 UNIX

Because of their expense, HPC systems tend to be shared by multiple users who interact with them remotely. They often run UNIX-style operating systems, which emphasize command-line utilities over graphical user interfaces. While GUI applications allow for more intuitive use, command-line utilities are often more appropriate for customized or repetitive tasks. Rather than allowing interactive access to all users, HPC systems run in batch mode, where users submit jobs—requests for a particular program to run for a specific amount of time, which are scheduled to run in such a way that the necessary resources are completely dedicated to a single application for a single user for a fixed period of time. Once a program begins its scheduled run, it runs continuously and without interruption from the system or additional input from the user. This is in contrast to single-user systems that support multi-tasking, where one or a small number of processors alternate between several applications and services, potentially multiple times per second.

3.3 Basic Programming

Many HPC applications are written in a low-level programming language such as C or Fortran. HPC programmers will benefit from knowing at least one of these. Higher-level languages such as Java and Python are increasing in popularity and can be a viable option insofar as they are able to leverage optimized numerical libraries. These higher-level languages may also be used for prototyping. The HPC landscape is still dominated by C and Fortran codes; the compilers for these languages have been optimized for decades.

Before tackling the development of programs targeted to HPC platforms and applications, users should learn the basics of computer programming and software engineering. Over the life-cycle of a successful software package, the greatest expenses are incurred in its maintenance rather than its initial development. Programmers should write their code in a way that other human beings can read and understand. Meaningful variable names, appropriate code comments, and consistent application of formatting standards go a long way toward this end. At a minimum, programmers should learn basic principles of modular programming, such as the use of functions and subroutines. Other important concepts are dynamic memory allocation, management of data on the stack and heap, and principles of compilation and linking (including use of the `make` utility). A basic understanding of object-oriented principles—including encapsulation, inheritance, and polymorphism—is essential for users of C++ or Java.

Some programmers develop code using a text editor such as `emacs` or `vi`. These programs enable powerful code navigation as well as find and replace operations (using regular expressions), among other features. Other programmers prefer to use an integrated development environment, such as Eclipse. An up-front investment in learning some of the intermediate and advanced features in such tools pays big dividends in productivity over time.

3.4 Parallel Programming Concepts

Developing software for parallel machines brings additional challenges to the programmer. The different programming models and paradigms that are available on HPC systems force the developer to think in fundamentally different ways.

In the shared memory paradigm (exemplified by OpenMP), processors share access to the same physical memory and address space. A *data race* occurs when multiple processors attempt to access the same memory location at approximately the same time and at least one of those accesses is a write. The result can be that the wrong data is read and/or written. To avoid this problem, programs must take care to ensure *mutual exclusion* for shared data. A *critical section* is a segment of code in which data races can occur. Correct programs ensure that only one process can be in a critical section at any given time.

In the case of OpenMP, it may be possible to parallelize an existing sequential program by adding a few compiler directives to the program's loop structures. The programmer is responsible to identify which loops can be parallelized.

A key impediment to scalability in shared memory systems is the competition for memory bandwidth among the processors. Current shared memory systems may include up to tens of processors.

In the distributed memory paradigm, exemplified by MPI, each processor has its own memory space (although multicore nodes may in fact share the same physical memory). When processors need to share data, they do so through explicit message passing. Each processor receives a unique ID at the beginning of program execution. Although each processor executes the same program, these IDs can be used in branching constructs to force the processors to differentiate their behavior. For example, the branch for processor zero may include instructions to read a list of tasks from a file and send task descriptions to each of the other processors. The branches for other process IDs would include instructions to receive work from process zero.

Messages may be blocking or non-blocking. Blocking communication forces a sender or receiver to wait for feedback that the communication operation has been completed. Non-blocking communication allows a processor to interleave computation and communication, potentially reducing a processor's idle time and improving overall performance. Some applications may be programmed to use a hybrid model of both shared memory and distributed memory to take advantage of the system's architecture, which may include an internetwork of multi-core nodes.

A third parallel programming paradigm is the shared object model, exemplified by Charm++. In Charm++ the program is viewed as a collection of objects, each of which represents a piece of work. The Charm++ runtime system determines which objects execute on which physical processors; transfers may occur during program execution to improve load balancing.

Understanding the basic principles of parallel programming can help a developer decide which technologies and platforms are best suited to the target application.

3.5 Debugging

In the production of any software of more than trivial complexity, the ability to effectively use debugging tools is essential to the productivity of a developer. A debugger allows a programmer to step through the execution of a program, instruction by instruction, and to inspect the state of memory and the values of specific variables. This capability is invaluable in tracking down bugs. Debugging parallel applications presents additional challenges because the order in which instructions are executed can vary from run to run and because the programmer must track the execution of multiple simultaneously executing threads. This can be particularly challenging when programs execute on thousands of processors. The open-source GNU Project Debugger (GDB) can be used with programs written in C or Fortran (and some other languages). Data Display Debugger (DDD) is a graphical front-end that can enhance GDB's usability. Distributed Debugging Tool (DDT) is a debugger for parallel applications. Some HPC systems are supported by their own proprietary debuggers, such as Intel's TotalView.

3.6 Profiling

A code profiler is a tool that measures which blocks of code in a program actually consume the most processing time. Understanding where the bottlenecks are can help guide a developer's efforts in improving the efficiency of the code. Profiling parallel applications presents additional challenges, as communication costs between processors must be evaluated as well. The `gcov` and `gprof` utilities support basic profiling operations. PerfSuite and TAU are popular software tools for performing code profiling in a parallel setting.

3.7 Optimization

Profiling goes hand-in-hand with code optimization and tuning. Many of the same principles and strategies that apply in everyday computing also apply to the optimization of HPC codes. Increasing locality to improve cache performance is one example. In the parallel setting, there are additional avenues for optimization, such as trading off redundant computation and interprocessor communication, reallocating tasks among processors to improve load balancing and reduce idle time, or interleaving computation and communication. At a minimum, users should have a basic understanding of the optimization flags supplied by the compiler. Output from a profiler and understanding of system architecture and algorithm behavior help developers know what changes to make to their programs to improve overall performance.

3.8 Numerical Libraries

Many scientific applications require advanced mathematical operations such as solving systems of equations, finding eigenvalues or eigenvectors, linear and non-linear programming, Fourier transforms, sorting, and taking samples from probability distributions. Over the past few decades, researchers have devoted significant efforts to create broadly applicable and highly-tuned libraries to perform such operations. Prudent HPC users take advantage of these resources rather than "reinventing the wheel." A first step is to become familiar with scientific and numeric libraries in a sequential environment. This includes learning how to link to third-party libraries at build-time or runtime and how to manipulate the data in a program to conform to the library's published APIs. A basic understanding of the Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK) libraries is useful, even if these routines are not used directly. When intensive numerical computations themselves must be spread across numerous processors to achieve the required efficiency, additional complexity applies. Commonly used parallel numerical packages include ScaLAPACK, PETSc, and NAG.

3.9 Parallel I/O

Advances in I/O subsystems have lagged far behind the advances in processor technology. HPC applications may produce and consume terabytes of data. The result is that I/O is often the most debilitating bottleneck for HPC applications. Some problems can be ameliorated by aggregating read and write requests to a small number of processors to reduce competition for I/O resources. The Lustre filesystem is commonly used on Linux clusters and other HPC systems to help manage these challenges.

3.10 Parallel Programming Languages

Today's HPC users have access to a wide variety of programming languages and technologies. The most appropriate language for a particular project depends on the computing resources available and the nature of the problem itself. Many quality training materials exist for widely-used, mature languages such as MPI and OpenMP. Other languages included in our training roadmap are Charm++, CUDA, HPF, X10, UPC, and Co-array Fortran. In addition to providing links to tutorials and documentation for these languages, one of the goals of our training roadmap will be to help users understand the contexts in which each language is appropriate.

3.11 Checkpointing and Fault tolerance

Users of personal computers and scientific workstations have grown accustomed to the high reliability of their systems. Periodic backup of disk drives is a prudent step to avoid loss of data, but systems often enjoy a mean time between failure on the order of months or years. Failure is truly the exception. However, today's supercomputers may have hundreds of thousands of processors and thousands of disk drives. While each component may be relatively reliable independently, the probability of *some* failure in the aggregate system during the execution of any particular application is very high. Mean time between failure for the system as a whole is often on the order of a few hours or less. Therefore, the need for HPC programs to be able to recover from failures is particularly acute. *Checkpointing* is the practice of periodically writing data or saving the state of execution to disk in order to facilitate recovery in the case of failure. Other fault tolerance measures may include the ability for a system to detect at runtime that a processor has failed and to regenerate or migrate the work that that processor was responsible for to other processors.

3.12 Applications

HPC systems are used for a variety of scientific and engineering applications. On the one hand are applications of very large scale, such as simulations of the history of the universe since the big bang. These programs operate on scales of light years. At the other end of the spectrum are applications that model phenomena on nanometer scales, such as the folding behavior of proteins in cellular processes. These simulations may have timesteps on the order of femtoseconds. Other applications include weather and climate modeling, computational fluid dynamics, genomics, optimization of engineering artifacts (aircraft, bridges, plastics), combinatorial state space search, logic programming, rational drug design, and finance/trading. The dominant combination of algorithms, programming models, and architectures varies from one application domain to another.

3.13 Data Management

HPC applications may produce many terabytes of data. The ability to create, process, and transfer files of this magnitude presents special challenges. Tools like GridFTP and FileZilla facilitate the efficient transfer and management large files on a network. The Hierarchical Data Format (HDF5) standard and the associated libraries are mature and widely used tools for managing large amounts of numerical data.

3.14 Visualization

High performance computing applications often produce massive quantities of data that are most profitably understood and analyzed visually. In some cases, visualization tools are used to create animations of physical phenomenon over time, such as the spread of a tsunami after an earthquake. In other cases, visualizations help users make a qualitative analysis of high dimensional data. Popular parallel visualization tools include VisIt and ParaView.

4. CONCLUSION

New users in the high performance computing community are often highly competent individuals who are learning the ins and outs of the space on their own rather than through formal training. The training roadmap that we have outlined in this work will help them identify the key issues and challenges that they are likely to encounter as HPC users and guide them to high quality online resources that they can use to learn more. We believe that this roadmap will help to reduce the frustration that new users experience and to help them avoid making common mistakes or wasting valuable time and effort to solve problems that have already been solved by others.

The roadmap be featured online at `hpcuniversity.org`. We welcome feedback from the HPC community with respect to content that is included or omitted in the general overview and the resources that are recommended for each topic.

Acknowledgment.

Funding for this project is provided by the Department of Education FIPSE program through the Atlantis Program, award number P116J090044.