Supercomputing in plain English: Teaching high performance computing to inexperienced programmers

H. Neeman, [a] J. Mullen, [b] L. Lee [c] & G.K. Newman [c]
[a] *School of Computer Science, University of Oklahoma*
[b] *Computing and Communications Center, Worcester Polytechnic Institute*
[c] *School of Chemical Engineering & Materials Science, University of Oklahoma*

Abstract

Although the field of High Performance Computing (HPC) has been evolving rapidly, the development of standardized software systems – MPI, OpenMP, LAPACK, PETSc and others – has in principle made HPC accessible to a wider community. However, relatively few scientists and engineers take advantage of computational science and engineering (CSE) in their research, in part because of the considerable degree of sophistication about computing that HPC appears to require. Yet the fundamental concepts of HPC are fairly straightforward and have remained relatively stable over time. These facts raise an important question: can scientists and engineers with relatively modest computing experience learn HPC concepts well enough to take advantage of them in their research?

To answer this question, HPC educators must address several issues. First, what are the fundamental issues and concepts in CSE? Second, what are the fundamental issues and concepts in HPC? Third, how can these ideas be expressed in a manner that is clear to a person with relatively modest computing experience? Finally, is classroom exposure sufficient, or is guidance required to assist investigators in incorporating HPC into their research codes?

We discuss an effort, now underway, to develop materials that express sophisticated scientific computing concepts in a manner accessible to a broad audience. These materials, now partially completed, will address many of the above questions, employing plain English, analogies and narratives to target this population. In addition, we examine a programmatic approach that incorporates not only the use of these materials but also the crucial contribution of followup.

Send correspondence to Henry Neeman (hneeman@ou.edu)

1 Introduction

Computational science and engineering (CSE) and High Performance Computing (HPC) are inextricably linked, because of the tremendous computing resources required by many of the problems of greatest scientific and engineering interest. The CSE community has recognized, quite rightly, that its body of methodologies has reached a sufficient level of sophistication and power that it can now take its place beside theory and experimentation in a triad of research strategies, each with a roughly equal contribution to scientific understanding. However, this perspective is not shared among the academic community as a whole: while theory and experimentation are taught at every level of science and engineering education, computational approaches are much less frequently encountered, and when covered are often addressed in considerably less depth. We conjecture that a primary reason for this inconsistency is that cutting edge CSE appears to require substantial sophistication about computing, and since many scientists and engineers receive only modest training in this area, many practitioners and educators believe that CSE is beyond the reach of much of their community. The ongoing increase in computing power and capability has made many large scale CSE problems practicable, but the lack of training has left many avenues of investigation unexamined.

Moore's Law[22] has been borne out over the last several decades, leading to a situation in which high performance systems are available at very low prices, in large part because of the rise of Linux clusters. Built from commodity off the shelf parts and employing open source and other free or inexpensive software, these systems provide tremendous computational capability at staggeringly low prices, typically less than a dollar per megaflop.

Thus, the rise of Linux clusters has lead to a situation in which a great many researchers could employ CSE in their investigations, but for their lack of familiarity with these techniques. Yet many of these investigators could be the best people to engage in this kind of endeavor, because of both their strong scientific background and their strong desire to conduct cutting edge research. To ameliorate this situation, we believe that new pedagogical strategies must be developed, taking into account the capabilities of this target population, with their strong mathematics and science backgrounds but scant computing experience.

To achieve this goal, CSE and HPC educators must address several issues. First, what are the fundamental issues and concepts in CSE? Second, what are the fundamental issues and concepts in HPC? Third, how can these ideas be expressed in a manner that is clear to a person with relatively modest computing experience? Finally, is classroom exposure sufficient, or is guidance required to assist investigators in incorporating HPC in their research codes?

A helpful way to describe the fundamental issues of CSE is as a chain of abstractions: phenomenon, physics, mathematics (continuous), numerics (discrete), algorithm, implementation, port, solution, analysis and verification. In general, physics, mathematics and numerics are addressed well by existing science and engineering curricula – though often in isolation from one another – and therefore instruction should be provided on issues relating primarily to the later items, and on the interrelationships between all of them. For example, algorithm choice is a fundamental issue whose gravity is seldom appreciated by those who have less computing experience; a common mistake is to solve a linear system by inverting the matrix of coefficients, without regard for performance, conditioning, or exploitation of the properties of the matrix.

As for HPC, the literature tends to agree on the following fundamental issues: the storage hierarchy; instruction-level parallelism; high performance compilers; shared memory parallelism (e.g., OpenMP [1],[16]); distributed parallelism (e.g., MPI [21],[23]). The pedagogical challenge is to find ways to express the basic concepts with minimal jargon and maximal intuitiveness.

Although expressing these concepts in a manner appropriate for the target audience can be challenging, it does not need to be daunting. For example, the use of analogies and narratives to explain these concepts can capture the fundamental underlying principles without distracting the students with technical details. Once the students understand the basic principles, the details of how to implement HPC solutions are much easier to digest: we believe that first they should learn about the forest, and then come to understand the trees.

However, it is probably not reasonable, in many cases, to expect novice programmers to immediately understand how to apply HPC concepts to their science. We conjecture that acceptable results require, as a follow on, regular interactions with experienced HPC practitioners to provide the needed insight and practical advice to move research forward.

As for the smaller subpopulation that already enjoys considerable computing experience, we expect that the proposed pedagogical methods will not only provide a valuable perspective on the fields of CSE and HPC, but will also give this group experience with means of explaining their pursuits to people outside their discipline — a need too often overlooked in science and engineering education.

In August 2001, the University of Oklahoma (OU) established the OU Supercomputing Center for Education & Research (OSCER) [8] as a division of the Department of Information Technology, with a mandate to provide not only hardware and software, but more importantly HPC education and research facilitation. In particular, OSCER's mission is to teach the use of HPC in scientific computing to everyone on OU's three campuses that uses, or desires to use, HPC in their research or coursework.

To address this need, OSCER has instituted several new programs, including:

- a workshop series, originally presented in Fall of 2001 and being repeated in Fall of 2002 (and likely to become an annual event), titled "Supercomputing in Plain English" (originally titled "Supercomputing & Science");
- "rounds," in which OSCER personnel visit face to face with each individual research team — including undergraduates, graduates, faculty and staff — week by week, exchanging ideas and experiences to develop near- and long-term action plans for moving the HPC aspects of the research forward, with the members of the research team responsible for implementation;
- proposal facilitation, including partnering directly with individual research teams on specific projects, as well as providing text for proposals describing not only OSCER's resources but also OSCER's role in providing advising and mentoring for students involved in such projects.

At Worcester Polytechnic University, the Parallel & Distributed Computing Applications Team has been providing a service similar to OSCER's HPC rounds, with the same purpose in mind: assisting researchers in the integration of computational science and HPC. These corresponding efforts at our two institutions has led us to a new collaboration. Under the NSF's Combined Research & Curriculum Development (CRCD) program, we are beginning a project to study the effectiveness of the teaching methods discussed here.

2 Computational Science & Engineering

The traditional approach to engineering and science education incorporates theory and experimentation to explore the physical and biological phenomena of the natural world. Experimentation allows the scientist to probe a given phenomenon, which in turn provides insight for deeper inquiry, as well as validation of the mathematical models that comprise the theory. Theoretical models that are derived from physical principles and mathematics provide a means of predicting anticipated phenomena and behavior.

With improvements in computing performance, a third method of scientific investigation has become available to researchers. Computer simulation bridges theory and experimentation: the former, in the guise of model and numerical method development; the latter, in that it allows one to probe and examine physical phenomena heretofore inaccessible, either because of location (e.g., hydrocarbon flow through rocks) or because of the intractable cost of instrumentation (e.g., ocean-atmosphere flows).

As with any experiment, the procedure must be performed carefully and accurately to assure reliable results. In the context of numerical simulation, this means that at every stage of the hierarchy of computational science, researchers must ensure that the approach they take is correct, accurate and appropriate: a good numerical method will not help bad mathematics, an efficient algorithm will not help an inappropriate numerical method, and so on. For large scale computations, the fact that a model, numerical method or algorithm works in principle is not enough; the choice must also incorporate sufficient parallelism to render the numerical experiment practicable. Therefore, for problems of considerable size, the strategy most likely to bear reliable results requires the integration of mathematics, computer science and application — and so cooperation between practitioners of these disciplines is a critical contributor to success. It is the overlap of these fields that has led to the generalization now referred to as CSE, and that is evident in the fundamental issues associated with the discipline:

- symbolic expression of a physical system by means of a continuous mathematical model;
- the discretization of the mathematical problem;
- the choice of numerical technique;
- implementation of the algorithm;
- verification of the code.

The first item is generally part of the training associated with an individual scientific field, while the second and third are the purview of mathematicians, the fourth is within the realm of computer science and the last relates to the specific field under consideration. Though researchers know their disciplines well and can recognize computational artifacts when they surface, few have the time and opportunity to investigate all the relevant disciplines associated with CSE.

As a result, the standard approach has been to teach formal concepts within a computational course associated with a particular field; for example, computational biological processes, computational fluid dynamics or computational mechanics. These courses generally cover issues such as accuracy and stability of various numerical schemes, methods of discretization in both time and space, and solution techniques. (For the sake of this discussion we use “solution” to mean the discrete approximation to the continuous solution.) In many cases, the primary

objective of the course is to enable students to think about these issues as they relate to the use of "canned-ware." Yet commercial software packages typically trail the achievements of the research community by several years, and the manner in which these systems are employed in classrooms can differ substantially from how they are used in a research context. Therefore, enlarging the population of scientific software developers is a crucial component in advancing the state of computational research. Clearly, a large portion of the material covered in these courses is pertinent to a variety of application areas, and furthermore, scientists and engineers with a background in calculus, differential equations and linear algebra, as well as at least modest experience with programming, have the tools necessary to understand the key features of computational science and parallel computing. However, it is rare that any of these kinds of courses present the issues associated with the development of code for parallel simulation.

The scientific investigations that our target population of researchers seek to undertake require large scale computing. At the same time, dual processor machines are common enough that they are becoming the workstations of choice. This confluence of need and available technology suggests that a new presentation of the issues of computing as they relate to scientific research is in order. The traditional paradigm, involving researchers referencing a standard set of computational methods commonly applied to their field, along with some numerical linear algebra, no longer provides investigators the exposure and experience required to easily use current computing power to good advantage. To bridge this gap, we propose to take a fresh view of each of the fundamental issues of CSE, recognizing that each falls within some domain of engineering, mathematics, science or computer science, but that all are required to fully prepare the student or researcher.

In general, the target audience is well schooled in applying the governing equations of their discipline to a small set of idealized cases that can be solved in closed form. The typical participant is capable of stating the assumptions and constraints of a physical system and of simplifying the system based on these conditions. Thus, they are capable of deriving the mathematical statement of the continuous problem, thereby satisfying the first fundamental issue of CSE.

For these students to develop simulation codes, they must next approximate the continuous problem by a discrete model. For problems expressed as partial differential equations, the most popular methods are finite difference and finite element schemes, although other approaches, such as boundary element and spectral element schemes, are employed by a smaller subset of the research community. Most textbooks begin with finite difference methods, developing the mathematics and analysis of hyperbolic, elliptic and parabolic equations, each of which require slightly different treatment[27]. Idealized problems, typically presented in one dimension, are used to illustrate a variety of different discretizations and the linear systems that arise from these strategies. At this level, little is said about how one solves the linear system, beyond basic ideas of linear algebra and the desire, often unachievable, to express the system as a tridiagonal matrix. The underlying assumption has been that computing is serial, so that a tridiagonal solver is computationally efficient and straightforward to implement. While this approach provides a pathway to discussion of a subset of the issues associated with computational science, a more comprehensive procedure is required if sophisticated research ability is the goal. For example, while parallel implementations of tridiagonal solvers exist, they only obtain parallel efficiencies of between 0.3 and 0.5 based on the choice of algorithm[29]; that is, they may be insufficient for certain

problems of scientific and engineering interest.

To achieve the goal of preparing students and researchers to utilize HPC and parallel resources, we suggest a top down approach. We begin not with the idealized problems but with the general statement of the physical problem. From this general statement we form smaller subproblems associated with well known and well understood mathematical equations; e.g. Helmholtz, Poisson and Laplace Equations. In certain disciplines, such as fluid dynamics, we still have a coupled system of equations and the focus must turn to algorithm and numerical discretization choice. For problems of this type, some sort of splitting is required to separate the physical mechanisms that are to be solved explicitly from those to be solved implicitly. In the field of fluid dynamics, there is the added constraint of pressure, which is a global quantity. Indeed, the global pressure solve forms the major bottleneck of computational fluid dynamics on parallel architectures.

Thus the first step in this approach requires the student to take a slightly more abstract view and state the continuous problem as a set of subproblems of the form **Lu = f**, where **L** is a matrix operator, and **u** and **f** are vectors. The student can now choose between a variety of discretization schemes, such as finite difference, finite element or spectral elements (h-p finite elements). It is clear that at this step we are at the interface between mathematics and science, but though this approach is well known within certain communities, it is not broadly presented at the undergraduate or early graduate student level.

It is at this point in the development that we turn to computer science. Inherent in the choice of appropriate numerical techniques are the issues associated with parallel computing, such as data locality, message passing and the minimization of global operations, as well as the issues of HPC, which are described below. The vast majority of scientists and engineers do not have the resources to explore multiple methods and to write parallel solver code. For this population, PETSc[13],[14],[12] and other toolkits offer the opportunity to explore a variety of fast, efficient parallel solvers. Therefore, once the physical system has been transformed to a system of subproblems, we encourage these researchers to explore the use of toolkits.

The final step in development is code validation. Unfortunately, this step is often not well understood by young researchers and graduate students. Though not trivial in practice, validation of scientific code is accomplished in a straightforward manner. Within each of the science and engineering disciplines, there are well known closed form solutions that can test several elements of a larger code. In addition, the interplay between experimentation and computation allows researchers to compare their results to physical data. It is important to understand that correctly completing a benchmark simulation is not enough; one should complete a resolution study in both time and space, and verify that the results match the expected accuracy. This verification step is often overlooked, and for time dependent problems applied to physical situations spanning large times, it is often unclear whether or not artifacts exist. Satisfying this final issue of computational science provides the confidence that the simulation code is robust and accurate; to be confident of performance we need to consider the fundamental HPC issues.

Students of HPC and parallel computing need a different approach. While we would hope that these students will have experience with numerical linear algebra, we believe that a top down approach provides better exposure to the issues of parallel computing. Indeed, as students begin to grapple with the issues of data locality, message passing and parallel computing, they need to consider these issues

from the beginning of the software development process, as they move from the continuous to the discrete model. Even without numerical linear algebra, the material should begin with the continuous model, break it into smaller sub-problems and discuss the solver techniques within the larger presentation on discretization.

With the creation of PETSc and other parallel toolkits, we best serve our target audience by presenting the issues surrounding the use of parallel solvers. From that perspective, we can assist them in the development and deployment of methods that produce matrices for which the efficient solvers are appropriate.

3 Fundamental Issues of High Performance Computing

HPC, as a discipline, is largely concerned with software design and implementation issues associated with achieving maximal performance for a given algorithm; the choice of the algorithm itself — for example, an iterative solver for a system of linear equations — is more properly the focus of computational science than of HPC. For the most part, the issues that most affect HPC are:

- the storage hierarchy [17], [19], [20], [30],
- instruction-level parallelism [17]. [18]. [19], [20], [28], [30],
- high performance compilers [17], [19], [28], [30],
- shared memory parallelism (e.g., OpenMP) [17], [19], [30],
- distributed parallelism (e.g., MPI) [17], [18].
- scientific libraries [28], [30],
- I/O [30],
- visualization [18].

In addition, the past several years have seen the rise of the issue of remote, heterogeneous (i.e., Grid-based) computing.

The pedagogical challenge for the discipline of HPC is to find means of expressing these basic concepts in a manner that is approachable by scientists and engineers who have strong mathematical and scientific backgrounds but modest software development experience. Thus, teaching strategies for HPC require minimal jargon and maximal intuitiveness.

3.1 OSCER Workshops Fall 2001

OSCER has been developing and presenting a prototype workshop series to serve this need. In this section, we discuss the strategies developed, the experiences encountered, and ideas for future improvements. The original slides for the first incarnation of this workshop series, presented in Fall 2001, can be found at
`http://www.oscer.ou.edu/education_2001fall.html`

The workshop series is split into seven sessions, each of which is presented to a mixed audience of undergraduate students, graduate students, faculty and staff, with a wide range of mathematical and scientific background, computing experience and application area of interest. Sessions are conducted in a loose, highly interactive style, with constant multiway dialogue during each session. The first session presents a broad overview of HPC as a discipline, and then the subsequent five sessions each cover a single topic: the storage hierarchy; instruction-level parallelism; high performance compilers; shared memory parallelism (e.g., OpenMP);

distributed parallelism (e.g., MPI). The final session is a "grab-bag" of topics presented in less depth: scientific libraries; I/O formats; visualization. Wherever possible, analogies and narratives are used to illustrate fundamental concepts.

3.1.1 Overview Session

The first session consists of brief looks at several of the topics that are to be covered in later sessions. The session begins with a statement of the goals of the series, a brief history of OSCER (in fall 2001, the first session took place on the day that OSCER officially opened for business), a broad definition of "supercomputing" and HPC, and the uses and users of HPC, particularly at OU.

Next, the basic issues underlying HPC, as described above, are outlined. Then, several of them are briefly presented, beginning with the storage hierarchy. For the benefit of those with modest computing experience, some very basic notions about computer organization are included, in particular a brief description of the CPU, primary storage (i.e., cache and RAM), secondary storage (hard disk, CDROM, etc), and I/O devices. Also incorporated here is a brief explanation of what cache does and how it helps performance. Immediately following is the Laptop Example.

The Laptop Example

The storage hierarchy pervades computing at all levels, from desktop to supercomputing. It can be described as a relationship between speed, price and size:

- fast storage is expensive and therefore computers tend to have little of it;
- slow storage is inexpensive and therefore computers tend to have a lot of it.

This point becomes more clear when applied to a familiar computer, so the laptop of the presenter is a straightforward choice, since it is visible to the attendees. In Fall 2001, the following table was presented (prices were accurate as of late August 2001, and therefore were already out of date for this publication):

Item	Peak Speed (MB/sec)	Size (MB)	Cost ($/MB)
Registers	16,800 [26]	112 bytes	unknown
Cache Memory (L2)	11,200 [25]	0.25	$400 [3]
Main Memory (100 MHz)	800 [4]	256	$1.17 [3]
Hard Drive	100 [6]	30,000	$0.009 [3]
Ethernet (100 Mbps)	12	unlimited	charged monthly
CD-RW	3.6 [5]	unlimited	$0.0015 [2]
Phone modem (56 Kbps)	0.007	unlimited	free (local call)

The table demonstrates a specific incarnation of the storage hierarchy, in terms that anyone who regularly uses a PC can understand. Most importantly, it shows the orders of magnitude of differences in performance between the various levels of the hierarchy: cache is 14 times faster than RAM, which is 8 times faster than a hard drive, which is 8 times faster than 100 Mbps ethernet, and so on. As a followon to the Laptop Example, a brief mention is made of storage use issues: register reuse, cache reuse, data locality and I/O efficiency.

The next topic touched on during this session is Instruction Level Parallelism (ILP), with a precis of the ILP material (Section 3.1.3, below).

The final major HPC topic addressed in the Overview session is multithread/ multiprocess parallelism, which is described using the Jigsaw Puzzle Analogy.

The Jigsaw Puzzle Analogy

The jigsaw puzzle analogy was presented "live" by using workshop attendees as physical models for shared memory and distributed parallelism. One or a few at a time, several attendees are asked to come to the front of the room to participate in putting together a jigsaw puzzle, to illustrate parallel programming concepts.

An Analogy for Shared Memory Parallelism

> *Suppose that Lloyd is working on a certain jigsaw puzzle, and that it takes him an hour to complete the puzzle.* (Here, a participant sits at a table in the front of the room, with an appropriate jigsaw puzzle.)
>
> *Now, suppose that Julie sits down at the table across from him, and works with him on the puzzle.* (Another participant sits at the table, across from the first.)
>
> *For the sake of argument, let's assume that half the puzzle is grass and the other half is sky. If Lloyd does the grass and Julie does the sky, how long will it take the two of them?*

Here, the audience will offer answers that will either be "half an hour" or "a little more than half an hour,." leading to a discussion of issues such as:

> *If Lloyd and Julie both try to grab a piece out of the pile at the same time, will that slow them down? What if they grab for the same piece at the same time?* (They demonstrate.)
>
> *Can Lloyd and Julie work completely independently, or will they need to work together at the horizon — that is, on the interface between their parts?*
>
> *Will these issues slow them down? How much? How long will it take the two of them, if one of them can do it in an hour?*

These questions lead to a discussion of fundamental concepts such as contention for shared resources (particularly shared memory) and communication between threads at the interfaces between subdomains.

> *Now suppose that Henry and Jerry sit at the table as well.* (Two more participants sit at the table, around the sides.)
>
> *Will there be more contention for the shared resource, or less, or the same amount? What about communication at the interfaces? How long will it take the four of them?*

By this time, the issues of contention and communication are becoming clearer.

> *Now suppose that four more people sit at the corners of the table. How long will it take the eight of them? Can we keep adding people around the table and continue to get speedup?*

At this stage, the discussion turns to the concept of diminishing returns.

Extending the Analogy to Distributed Parallelism

> *Now suppose that we have two tables, with Lloyd sitting at one and Julie at the other. And suppose we have some way to pull apart the pile of pieces so that Lloyd gets half the pile and Julie gets half the pile.* (The other participants step aside, and Julie goes to another table, with half the pile of pieces.)
>
> *Remembering that Lloyd can do the puzzle in an hour, how long will it take Lloyd and Julie now? What will they do differently, compared to sitting at the same table and sharing the same pile of pieces? Will they ever reach for the same piece? How will they do the horizon?*

These questions lead to a discussion of the tradeoff between contention for shared resources and the high cost of remote communication, and the need for Lloyd and Julie to bring together their halves of the puzzle, or even their tables, at the end of the process.

> *If Lloyd's part is easier than Julie's, will they be ready to do the horizon at the same time? If not, what will Lloyd do after he finishes his part? What will be the impact on speedup?*

Here, the discussion touches on load balancing and blocking under synchronous communication.

> *Now let's add a couple more tables, with Henry at one and Jerry at the other.*
>
> *Do we need to split up the pieces into smaller piles? What if there's no straightforward way to split up the pieces so that the piles are roughly equal size?*

These questions lead to a discussion of the differing levels of difficulty of data decomposition for various applications; that is, some problems decompose trivially (e.g., a uniform Cartesian mesh can be broken into roughly equal-sized chunks), while others are extremely difficult to decompose.

> *If we have four tables instead of two, how much time will be spent communicating: more, less or the same? Can we come up with ways to make communication more efficient? Can Lloyd and Julie communicate at the same time that Henry and Jerry do? Can we guarantee that they will, or will it depend on how we break up the piles of pieces?*
>
> *Can we keep adding more and more tables? What will be the impact on speedup?*

Extending the Analogy to Hybrid Parallelism

> *Suppose that we have several tables, all properly set up, but each table has several people at it. Each table can share a pile of pieces, but the tables are independent except for communicating at the interfaces between them.*
>
> *Would this be a good idea or a bad idea? What would be the impact on performance? What does this depend on? Would the group's overall performance be improved if each table has people who work really well together?*

Why Use HPC?

At the end of the Overview session, the attendees are asked the most fundamental question about the relationship between HPC and their research: "Why bother with HPC at all?"

> *It's clear that making effective use of HPC takes quite a bit of effort, both learning and programming. That seems like a lot of trouble to go to just to get your code to run faster.*
>
> *It's nice to have a code that used to take a day run in an hour. But if you can afford to wait a day, what's the point of HPC? Why go to all that trouble just to get your code to run faster?*

The answer:

> *What HPC gives you that you won't get elsewhere is the ability to do bigger, better, more exciting science.*

3.1.2 Storage Hierarchy Session

This session is begins with, and expands upon, the Laptop Example as presented in the Overview session (Section 3.1.1, above), and provides substantially more detail about computer organization issues that affect performance. Where possible, timing tests on simple code fragments demonstrate the principles being discussed.

After the Laptop Example, the topic of computer organization is examined, with greater detail about registers, cache and main memory, and about the relationships between them. The section on registers is concerned with basic issues: what registers are, how they are used, how many registers a typical CPU has. The section on cache describes what cache is, and how fast cache is compared to main memory. This is followed by a section briefly describing main memory. These three sections capture the distinctions between registers, cache and main memory:

- *Registers ... hold data that are being used right now ....*
- *[Cache is] a very special kind of memory where data reside that are about to be used ....*
- *[Main memory is] where data reside for a program that is currently running ....*

Then, the relationship between cache and main memory is described, including an explanation of cache lines and mapping strategies — direct, fully associative, set associative — that gives information on the advantages of each strategy. Finally, the benefit of having cache is explored.

The next section is on data locality, and begins by noting that, if a problem size is substantially larger than cache, then most of the data is outside of cache. After defining *cache hit* and *cache miss*, it is noted that:

> *If all of your data is small enough to fit in cache, then when you run your program, you'll get almost all cache hits ... which means that your performance might be excellent!*

Of course, most problems are not small enough to fit entirely in cache, especially the small caches of current commodity processors (e.g., 512 KB L2 cache on a Pentium 4). So, how can the cache hit rate be improved?

> *Use the same solution as in Real Estate:*
> *Location, Location, Location!*

This statement leads into a discussion of data locality, including temporal and spatial locality. As proof of the importance of locality, a simple code example is timed, comparing two similar loops, one that fills an array from start to finish, and another that fills the same array in a randomly permuted order. A graph of loop runtime for various array sizes is given, showing a factor of 6 to 8 improvement for the ordered fill versus the randomly permuted fill. (Actually, part of the performance difference is the result of pipelining, but this fact is not mentioned at this stage.)

Next, a much more compelling example is explored: matrix-matrix multiply. Initially, the naive, hand-coded version is presented, based simply on the mathematical definition of the matrix-matrix multiply operation:

```
  DO c = 1, nc
    DO r = 1, nr
      dst(r,c) = 0.0
      DO q = 1, nq
        dst(r,c) = dst(r,c) + src1(r,q) * src2(q,c)
      END DO ...
```

The performance of the hand-coded version of this routine is compared against the Fortran 90 intrinsic, `MATMUL`, and is shown to be substantially worse, 4 to 5 times slower for large matrices.

This example leads directly into a discussion of tiling. A new version of the code is presented, using tiling:

```
SUBROUTINE matrix_matrix_mult_tile (      &
 &           dst, src1, src2, nr, nc, nq, &
 &           rstart, rend, cstart, cend, qstart, qend)
  ...
  DO c = cstart, cend
    DO r = rstart, rend
      if (qstart == 1) dst(r,c) = 0.0
      DO q = qstart, qend
        dst(r,c) = dst(r,c) + src1(r,q) * src2(q,c)
      END DO ...
END SUBROUTINE matrix_matrix_mult_tile

  DO cstart = 1, nc, ctilesize
    cend = cstart + ctilesize - 1
    IF (cend > nc) cend = nc
    DO rstart = 1, nr, rtilesize
      rend = rstart + rtilesize - 1
      IF (rend > nr) rend = nr
      DO qstart = 1, nq, qtilesize
        qend = qstart + qtilesize - 1
        IF (qend > nq) qend = nq
          CALL matrix_matrix_mult_tile(       &
 &              dst, src1, src2, nr, nc, nq, &
 &              rstart, rend, cstart, cend,  &
 &              qstart, qend)
      END DO ...
```

This tiled version is timed using a variety of tile sizes (from the full problem size down to 2 x 2), on a variety of problem sizes (from 512 x 256 up to 2048 x 1024). All problem sizes show essentially the same behavior, with the performance curves having the same shape (in a log-log plot) regardless of problem size, and with the best performance found at tile sizes of 64 x 64 x 32 (32,768 bytes) through 32 x 32 x 32 (12,288 bytes) on a 700 MHz Pentium III. Smaller tile sizes than this take longer, leading to a discussion of the overhead associated with procedure calls. In addition, this example demonstrates that, in some cases, tiling can be as straightforward as wrapping the original, non-tiled version of an algorithm in a few loops that decompose the dataset into tiles.

The final sections of this session are concerned with slower technologies: hard disk, virtual memory and the Internet. The portion on hard disk explains that disk is slower than RAM because mechanical devices are slower than electronic devices; that is, objects move slower than electrons. Then, I/O strategies are discussed:

Read and write the absolute minimum amount.

- *Don't reread the same data if you can keep it in memory.*
- *Write binary instead of characters.*
- *Use optimized I/O libraries like NetCDF and HDF.*

The section on virtual memory gives an overview of what virtual memory is and how it works, and how to exploit data locality to minimize page faults. The section on the Internet simply cautions that one should avoid using it in situations in which performance is important. Finally, the storage use strategies presented in the Overview session (Section 3.1.1, above), are repeated, as a summary of the issues associated with the storage hierarchy.

3.1.3 Instruction-Level Parallelism Session

This session begins with an intuitive definition of instruction-level parallelism:

> *Instruction-Level Parallelism is a set of techniques for executing multiple instructions at the same time within the same CPU.*

The underlying assumption of this session is that many of the attendees have very modest computer science background — that is, as little as a single programming course — and thus they have little or no awareness of current processor architecture. So, the extent of their understanding of computer organization is as a simple Von Neumann model of purely serial computation — that is, execution of exactly one instruction at a time.

As a result, this session is anticipated to be the most daunting for many in the target population, not only because they lack sufficient background information to be aware of the advances in computer engineering that have enabled instruction-level parallelism, but also because their unfamiliarity with low-level constructs such as assembly language and microcoding poses challenges in explaining concepts such as cycles, superscalar execution and instruction pipelines.

Therefore, an important aspect of this workshop session is the introduction of these fundamental concepts in a manner that is accessible to those inexperienced in computing, but at the same time is not tedious for the computationally sophisticated. The solution that we have adopted is to present the most basic background information as quickly and concisely as possible — in our case, a mere three slides, describing the concept of ILP, giving examples of instructions, and defining "cycle" — with additional information scattered throughout the presentation, often

subtly. For example, rather than dwelling on the concept of an assembly-level instruction — or even identifying it with the term "assembly," which in this context is neither necessary nor especially helpful — we present brief program fragments along with their translations into pseudo-assembly:

```
z = a * b + c * d
```

1. Load `a` into `R0`
2. Load `b` into `R1`
3. Multiply `R2 = R0 * R1`
4. Load `c` into `R3`
5. Load `d` into `R4`
6. Multiply `R5 = R3 * R4`
7. Add `R6 = R2 + R5`
8. Store `R6` into `z`

Scalar Execution

In fact, the above example is used to introduce the concept of scalar execution, as a lead-in to concepts such as superscalar, pipeline, superpipeline and vector, because scalar execution is intuitive: it corresponds directly to many basic programming constructs in Fortran and C. Immediately following this example we present the same example, showing not only the above set of pseudo-assembly instructions but also, side-by-side, the same set of instructions in a different order — specifically, steps 1 through 3 are swapped with steps 4 through 6, and, within 4 through 6, steps 4 and 5 are swapped:

```
z = a * b + c * d
```

1. Load `a` into `R0`	1. Load `d` into `R4`
2. Load `b` into `R1`	2. Load `c` into `R3`
3. Multiply `R2 = R0 * R1`	3. Multiply `R5 = R3 * R4`
4. Load `c` into `R3`	4. Load `a` into `R0`
5. Load `d` into `R4`	5. Load `b` into `R1`
6. Multiply `R5 = R3 * R4`	6. Multiply `R2 = R0 * R1`
7. Add `R6 = R2 + R5`	7. Add `R6 = R2 + R5`
8. Store `R6` into `z`	8. Store `R6` into `z`

This side-by-side comparison illustrates the concept of independent sequences of instructions, which is perhaps the most fundamental property required to achieve parallel computation. Attendees are asked some fundamental questions:

- *If some of these instructions can have their order changed, can all of them?*
- *How can we tell when order matters and when it doesn't?*

These questions lead to an informal discussion of dependency analysis.

Superscalar Execution

Immediately following the side-by-side comparison, we introduce the concept of superscalar execution:

```
z = a * b + c * d
```

1. Load `a` into `R0` **AND** load `b` into `R1`
2. Multiply `R2 = R0 * R1` **AND**
   load `c` into `R3` **AND** load `d` into `R4`
3. Multiply `R5 = R3 * R4`
4. Add `R6 = R2 + R5`
5. Store `R6` into `z`

The advantage of this approach is that, without overtly focusing on formal definitions and details of computer organization, we have introduced superscalar execution in an intuitive manner. Given that many in our target population have an unsophisticated understanding of processor architecture, we speculate that the notion that a processor may be capable of multiple instructions simultaneously, though a new idea to many, will not be difficult to assimilate, nor will it strongly conflict with their preconceived notions of how processors behave, because for many of them their prior understanding is sufficiently vague and weak that the new information does not contradict an existing prejudice.

Loops

It is unlikely that this execution strategy — superscalar execution on a single statement — will impress most users, who will note that total runtime is reduced by only a few nanoseconds; specifically, the effective number of steps is reduced from 8 to 5. (For simplicity, we assume that all operations take the same amount of time, though we caution the attendees that in real life this is not the case.) At this point, we introduce loops.

Loops demonstrate the value of superscalar operation: in many cases, the individual iterations of a loop are independent of one another, so the superscalar units of the processor can work on them concurrently. Again, this concept is fairly intuitive for the target population, because we avoid such topics as scheduling, which are complicated without being, at this stage, especially enlightening.

Pipelining

On the one hand, pipelining can be explained intuitively, via analogies such as bucket brigades and assembly lines. On the other hand, the details of pipelining can be opaque, because the stages of the pipeline — instruction fetch, instruction decode, operand fetch and so on — are unfamiliar to many in the target population. Therefore, we avoid dwelling on the details of pipelining, focusing instead on pipelined performance, by showing benchmarks of loops that perform various operations. In addition, we discuss the kinds of constructs that can interrupt pipelining, such as index arrays, complicated loop bodies, premature loop exits, procedure calls and I/O. Finally, we briefly present superpipelining and vectorization, which are both relatively straightforward to explain.

Because of the comparatively high degree of technical detail in this session, our concern has been that participants would become lost or frustrated, so we insert into the presentation, at strategic moments, slides that simply read “Don’t panic!”[9]. We then explain why they shouldn’t panic:

> *In general, the compiler and the CPU will do most of the heavy lifting for instruction-level parallelism.*
>
> **BUT:**
>
> *You need to be aware of ILP, because how your code is structured affects how much ILP the compiler and the CPU can give you.*

In the context of instruction-level parallelism, the overarching issue for the target population is not the details of how ILP behaves, but rather the fact that, through sufficiently careful code implementation, they can induce a high performance compiler to produce a highly efficient executable. That is, they need not specifically concern themselves with targeting ILP, because compilers will do most of the work for them; rather, they should be aware of the kinds of constructs that compilers can optimize effectively, as well as those that stymie compilers. This discussion leads naturally into the next session, which presents high performance compilers and how to work well with them.

3.1.4 High Performance Compilers Session

This session consists of two major parts: dependency analysis and "stupid compiler tricks." The purpose of the session is to give the target audience strategies for improving performance by assisting the compiler in finding the best way to optimize a code. Also, this session sets the stage for discussions about parallelism. (Much of the material is derived from [17].)

In the section on dependency analysis, we define dependency analysis and distinguish between control dependencies and data dependencies. We also examine loop carried dependencies and reductions, and show timings that compare loops that do or do not have loop carried dependencies, demonstrating the extent to which these dependencies inhibit performance.

The section on "stupid compiler tricks" is split into two parts, on tricks that compilers play and on tricks to play with compilers. The tricks that compilers play include scalar optimizations (copy propagation, constant folding, dead code removal, strength reductions, common subexpression elimination, variable renaming), and loop optimizations (hoisting and sinking of loop invariant code, induction variable simplification, iteration peeling, loop interchange, unrolling, inlining). The tricks to play on compilers include compiler options (particularly optimization levels, including timing tests of various loops with and without optimization), profiling and hardware event counters. This section prepares participants for practical hands-on use of HPC, providing a context for discussion during rounds.

3.1.5 Shared Memory Parallelism Session

The session on shared memory parallelism begins with a contrast between the forest and the trees:

- *The Trees: We like multiprocessing because, as the number of processors working on a problem grows, we can solve our problem in less time.*
- *The Forest: We like multiprocessing because, as the number of processors working on a problem grows, we can solve bigger problems.*

After a brief look at jargon (threads versus processes), we examine Amdahl's Law[10], not as a formal construct, but as a means of examining the importance of parallelization throughout an application:

> *Rule of Thumb: When you write a parallel code, try to make as much of the code parallel as possible, because the serial part will be the limiting factor on parallel speedup.*

We probe this rule by examining some benchmarks of an example parallel code exhibiting sublinear speedup.

Next, we touch on granularity and parallel overhead, followed by repeating the shared memory portion of the Jigsaw Puzzle Analogy (above). Immediately after, we discuss the fork-join model of multithreading, as a lead-in to OpenMP.

About half of this session is devoted to OpenMP, which has become the *de facto* standard for shared memory parallel programming in the HPC community. Because most of the concepts are very new to the target audience, we begin with the notion of a compiler directive and show some OpenMP examples, thereby introducing an implied outline of this portion of the session.

We then introduce a first OpenMP program, which is a simple "Hello, world." A sample run, which includes setting the `OMP_NUM_THREADS` environment variable, not only shows parallel execution but captures nondeterministic ordering, with several runs showing the outputs of various threads in different orders.

Next, we examine the `PARALLEL DO` directive, because for most of the target audience, much of what they need is covered by parallel loops. This leads naturally to discussion of concepts such as chunks, private versus shared data, load balancing, synchronization and barriers, critical sections and reductions. Finally, we examine a simple strategy for parallelizing an existing serial code: placing parallel loop directives above the most important loops in the code.

3.1.6 Distributed Parallelism Session

The distributed parallelism session begins with the Desert Islands Analogy.

The Desert Islands Analogy

The purpose of this analogy is to capture the Single Program Multiple Data (SPMD) programming model that underlies MPI, and that therefore is one of the primary means of obtaining parallelism on clusters. The primary focus of this analogy is on the seeming incongruity between the isolation of processes from one another — that is, the fact that all data are private to their local process — and the cooperative nature of their behavior. This analogy requires substantially more exposition than the jigsaw puzzle analogy.

> *Suppose that Lloyd is in a little hut on his favorite desert island.* (Here, Lloyd would be asked to name his favorite island, and that name would be substituted throughout the discussion.)
>
> *In his little hut are a few things: a desk, a chair, a phone, a pencil, a calculator, and two sheets of paper, one with instructions, the other with numbers.*
>
> *There are a couple different kinds of instructions: some that involve calculations on the list of numbers, and some that involve voicemails.*
>
> *For example, a calculation instruction might say:*
>
> - *add the 137th number to the 928th and put the result in the 673rd slot;*
> - *compare the 54th number to the 816th, and:*
>   - *if they're the same, perform a certain sequence of instructions;*
>   - *if they're different, perform another sequence of instructions.*
>
> (These kinds of instructions would be recognizable by anyone with even a semester of programming experience.)

On the other hand, a voicemail instruction might say:

- *dial 555-0127 and leave a voicemail containing the 962nd number;*
- *call your voicemail box and collect a voicemail from 555-0063, and place the number contained in that voicemail into the 163rd slot.*

If Lloyd is in a hut on an island, is he specifically aware of anyone else? Does he know whether anyone else is working on the same problem as he is? Does he know who's at the other end of the phone line?

Now, suppose Julie is on another island somewhere, in the same kind of hut, with the same kind of equipment. (Here, Julie would likewise be asked to name her favorite island.) *Suppose that she has the same list of instructions as Lloyd, but a different set of numbers on her sheet and a different set of phone numbers for getting and leaving voicemails.*

Is Julie specifically aware of anyone else? Does she know whether anyone else is working on the same problem as she is? Does she know who's at the other end of the phone line? Does she know that Lloyd is working from the same set of instructions, but with different data and phone numbers?

Now, suppose that Henry and Jerry are also in huts on islands, with the same instructions but different data and phone numbers. And suppose that the phone numbers that the four of them call are each others': Lloyd leaves voicemails for and gets voicemails from Julie, Henry and Jerry, and so on.

Could the four of them be working on the same problem together, even though they're not specifically aware of each other?

This last question draws attention to a counterintuitive parallel programming paradigm: several seemingly independent processes that, through proper algorithm design, operate in concert to solve a problem far larger than the workload of any one of them.

Notice that Lloyd can't see Julie's data or Henry's or Jerry's, Julie can't see any of the other three's, and so on. That is, everyone's data are private: there's no way for anyone to share data, except by leaving voicemails.

A fundamentally important issue in message passing is the twin costs of passing a message: latency and bandwidth. Fortunately, long distance telephone charges are an excellent analogy.

When you make a long distance phone call, you typically have to pay two costs:

- *Connection charge: the fixed cost of connecting your phone to someone else's, even if you're only connected for a second;*
- *Per-minute charge: the cost per minute of talking, once you're connected.*

If the connect charge is large, then you want to make as few calls as possible, and to exchange as much information as possible during each call.

From this analogy, we can immediately map to distributed parallelism concepts: independent processes, private data, communication via message passing, latency and bandwidth. In addition, we present the concept of load balancing, both for its own sake and also to introduce the concept of data decomposition.

This discussion is followed by a brief overview of MPI, with modest code examples to illustrate basic concepts. Here, MPI terminology is introduced, along with a small subset of MPI routines. As with the Shared Memory Parallelism, above, we discuss the issue of indeterminant ordering of parallel execution. We also examine the encapsulation of messages, as well as collective communications. Finally, we end with an example: coverting a serial Monte Carlo application, which is embarrassingly parallel, to an MPI application, and use it as a springboard to examine asynchronous communication and communication hiding.

3.1.7 Grab Bag Session: Scientific Libraries, I/O and Visualization

This session examines three disparate topics, but in substantially less depth than the other sessions.

As an introduction to this session, the chain of computational science abstractions is presented, to provide context for the discussion of scientific libraries.

Scientific Libraries

Scientific libraries are presented to discourage many of the bad habits described in Section 2. Specifically, the section covers:

1. The availability of solver libraries
2. The "do's and don'ts" of solving systems of linear equations

   Don'ts*:*

   - **Don't** *invert the matrix (*$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$*). That's much more costly than solving directly.*
   - **Don't** *write your own solver code. There are people who devote their whole careers to writing solvers. They know a lot more about writing solvers than we do.*

   Do's*:*

   - **Do** *use standard, portable solver libraries.*
   - **Do** *use a version that's tuned for the platform you're running on, if available.*
   - **Do** *use the information that you have about your system to pick the most efficient solver.*
3. Knowing the properties of the matrix (e.g., symmetric, positive definite, banded, sparse);
4. Specific solver libraries (LAPACK[11], ScaLAPACK[15], PETSc).

LAPACK is used to illustrate these principles. It is briefly described, and then an example of LAPACK use is shown, to give the audience a sense of the assemble-solve-disassemble approach. Then, ScaLAPACK and PETSc are briefly discussed, followed by a brief look at the circumstances under which each of the three libraries is appropriate.

I/O Libraries

I/O libraries are presented in order to alert participants about two issues: performance and portability. The first of these issues is critical at runtime, because a badly chosen I/O method will slow a code down substantially. The second is important in the long run, because data stored in a non-standard format can become unreadable when it's ported to a new platform — and especially when the original platform is decommissioned.

This topic begins with text output, showing that the ASCII approach, while fully portable, is very inefficient for data with many significant figures. This approach is contrasted with outputting native binary, which can be substantially more efficient, but which has two costs: readability and portability. However, readability is not a major issue, because one can always write a program to output the binary data in text, and because large data files — the kind for which I/O performance is important — aren't typically examined by eye, because of their sheer size. However, binary portability is a substantial problem, leading into a mention of portable formats such as NetCDF[24] and HDF[7]. The advantages of this approach are presented.

Visualization

The final section, on visualization, is fairly straightforward and intuitive: the standard case is made for visualization (that contemporary scientific datasets are too large to look at value by value), and then examples of several species of visualization are presented: contour lines, slice planes, isosurfaces, streamlines and volume rendering.

4 The Importance of Followup

Obviously, communicating the breadth, depth and complexity of HPC in a modest number of workshops is impossible, and therefore doing so is not a goal of this approach. (Indeed, this point is raised in the Overview workshop.) Instead, researchers need substantial followup, both concurrent with and subsequent to the workshops, in order to apply HPC expertise to specific applications, but without requiring application scientists to become HPC experts in order to begin making progress.

Over the past year, OSCER has established HPC rounds, in which OSCER personnel work directly with research teams, meeting regularly (e.g., weekly) with several different groups to exchange ideas about applying HPC and computational science to their specific projects. Typically, this process occurs in three phases:

- a learning phase, in which the OSCER representative learns about the application and the research team learns how basic HPC strategies relate to their application;
- a development phase, in which appropriate optimization and parallelization strategies are discussed and implemented;
- a refinement phase, in which the initial methods are improved through profiling and other analysis techniques, leading to greater speedup, portability and so on.

In fact, these three phases overlap considerably, and in particular the learning phase is continual.

The importance of this approach cannot be overstated, and especially important is the availability of one or a few persons whose role is to develop and maintain expertise on HPC and computational science issues. This base of expertise can serve as a springboard to improving and expanding the nature, breadth and depth of computational research being conducted at an entire institution, at minimal cost to any individual research group.

A typical PhD-granting institution has a dozen or more research groups that can benefit from this kind of consultation, and a quick back-of-the-envelope calculation shows that 90 minutes a week of this person's time (combining teaching and rounds) is roughly half a month of Full Time Equivalent per year. So, if each of these research groups incorporates into their external funding proposals a line item for half a month of the HPC expert's time, a sizable portion (perhaps all) of this person's time can be funded externally. The burden on each of the individual teams is minimal, while each benefits substantially from this person's presence. Furthermore, this process spirals upward, as the HPC expert gains experience across a wide range of applications and computational approaches. Because this person is directly engaged in several projects, they receive credit on relevant publications in diverse topics, leading not only to their enhanced professional development but also to an improved reputation not only for themselves but also for their institution, thereby leading to increased probability of funding for any given project in which they participate.

5 Summary and Future Work

The ideas discussed in this paper are the springboard for a new project being conducted under the National Science Foundation's Combined Research and Curriculum Development program. Our goal is to apply these pedagogical strategies by incorporating this content into engineering and physical science coursework, taught not only by members of the project team but more importantly by instructors outside the team. This exercise will serve as a crucial testbed of the conjectures presented here.

One of the lynchpins of this project will be to present the materials described here, in a suitable form, to a nanotechnology course taught at OU by Lee, Newman and Neeman. In addition to tracking student attitudes and experiences, we will also provide a Monte Carlo simulation code for them to parallelize, working with the students to help them develop an appropriate solution. To achieve this, we will first parallelize the code ourselves, and then remove the parallel constructs, thereby guaranteeing that the code is in a form appropriate for parallelization. A similar approach will be taken by Mullen in a CFD course taught at WPI. Finally, the materials will be used by an instructor outside the project team in a course over which the project team has no control.

The approach to CSE and HPC education and research described in this paper — an admittedly cursory introduction to complicated, intricate material that uses plain English, analogy and narrative, combined with regular face-to-face interaction with an expert — has already borne fruit on a number of projects underway at OU, and is expected to meet with substantial success over the next several years. We believe that this strategy is not only effective but also cost effective, and we expect that it will become more popular within the scientific computing community in years to come.

6 Acknowledgments

This material is based on work supported by the National Science Foundation under Grant No. NSF-0203481.

References

[1] OpenMP C and C++ application program interface, version 1.0. OpenMP Architecture Review Board, October 1998.

[2] buy.com website. http://www.buy.com/retail/computers/category.asp?loc=484, Aug 2001.

[3] Dell computer corp webpage. http://www.dell.com/, Aug 2001.

[4] Intel 820 chipset performance brief webpage. http://developer.intel.com/design/chipsets/820/820perform.htm, Aug 2001.

[5] Toshiba sd-r2002 cd-rw/dvd-rom specifications webpage. http://www.toshiba.com/taissdd/techdocs/sdr2002/2002spec.shtml, Aug 2001.

[6] Toshiba super slimline 20 gb: Mk2018gas webpage. http://www.toshiba.com/taissdd/products/features/MK2018gas-Over.shtml, Aug 2001.

[7] NCSA HDF webpage. http://hdf.ncsa.uiuc.edu/, Aug 2002.

[8] OU Supercomputing Center for Education & Research webpage. http://www.oscer.ou.edu/, Aug 2002.

[9] D. Adams. *The More Than Complete Hitchhiker's Guide*. Longmeadow Press, Stamford CT, 1986.

[10] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30(8):483–485, 1967.

[11] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. http://www.netlib.org/lapack/, 1999.

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

[13] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1998.

[14] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, 1998.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK Users' Guide. http://www.netlib.org/scalapack/, 1997.

[16] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Diego CA, 2001.

[17] K. Dowd and C. Severance. *High Performance Computing*. O'Reilly & Associates, Inc., Sebastopol CA, 2nd edition, 1998.

[18] L. D. Fosdick, E. R. Jessup, C. J. C. Schauble, and G. Domik. *An Introduction to High-Performance Scientific Computing*. The MIT Press, Cambridge MA, 1996.

[19] R. Gerber. *The Software Optimization Cookbook: High-performance Recipes for the Intel Architecture*. Intel Press, United States, 2002.

[20] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. Society for Industrial and Applied Mathematics, Philadelphia PA, 2001.

[21] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 2nd edition, 1999.

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

[23] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., San Francisco CA, 1997.

[24] R. K. Rew, G. P. Davis, S. Emmerson, and H. Davies. *NetCDF User's Guide for C: An Interface for Self-Describing, Portable Data, Version 3*. Jun 1997.

[25] A. L. Shimpi. Intel Pentium 4 1.7GHz: Does the prophecy hold true? http://www.anandtech.com/showdoc.html?i=1460&p=2, Apr 2001.

[26] R. Standish. Introduction to High Performance Computing. http://www.ac3.com.au/edu/hpc-intro/node6.html, Oct 2001.

[27] J. C. Tannehill, D. A. Anderson, and R. H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor and Francis, Philadelphia PA, 2nd edition, 1997.

[28] W. Triebel, J. Bissell, and R. Booth. *Programming Itanium-based Systems*. Intel Press, United States, 2001.

[29] E. F. Van De Velde. *Concurrent Scientific Computing*. Springer-Verlag, New York, 1994.

[30] K. R. Wadleigh and I. L. Crawford. *Software Optimization for High Performance Computing*. Prentice Hall PTR, New Jersey, 2000.