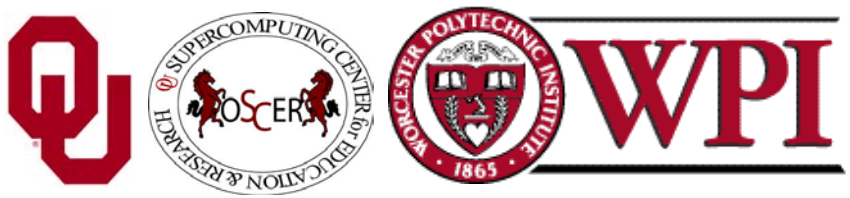
# Supercomputing in Plain English

## Teaching High Performance Computing to Inexperienced Programmers

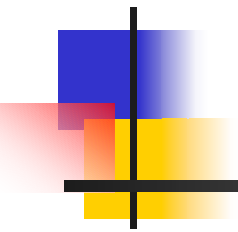
Henry Neeman, University of Oklahoma
Julia Mullen, Worcester Polytechnic Institute
Lloyd Lee, University of Oklahoma
Gerald K. Newman, University of Oklahoma

This work was partially funded by NSF-0203481.



# Outline

- Introduction
- Computational Science & Engineering (CSE)
- High Performance Computing (HPC)
- The Importance of Followup
- Summary and Future Work



# Introduction

# Premises

- Computational Science & Engineering (CSE) is an integral part of science & engineering research.
- Because most problems of CSE interest are large, CSE and High Performance Computing (HPC) are inextricably linked.
- Most science & engineering students have relatively little programming experience.
- Relatively few institutions teach either CSE or HPC to most of their science & engineering students.
- An important reason for this is that science & engineering faculty believe that CSE and HPC require more computing background than their students can handle.
- We disagree.

# The Role of Linux Clusters

- Linux clusters are much cheaper than proprietary HPC architectures (factor of 5 to 10 per GFLOP).
- They're largely useful for:
  - MPI
  - large numbers of single-processor applications
- MPI software design is not easy for inexperienced programmers:
  - difficult programming model
  - lack of user-friendly documentation – emphasis on technical details rather than broad overview
  - hard to find good help
- BUT: a few million dollars for MPI programmers is much much cheaper than tens or hundreds of millions for big SMPs – and the payoff lasts much longer.

# Why is HPC Hard to Learn?

- HPC technology changes very quickly:
  - Pthreads: 1988 (POSIX.1 FIPS 151-1) [1]
  - PVM: 1991 (version 2, first publicly released) [2]
  - MPI: 1994 (version 1) [3,4]
  - OpenMP: 1997 (version 1) [5,6]
  - Globus: 1998 (version 1.0.0) [7]
- Typically a 5 year lag (or more) between the standard and documentation readable by experienced computer scientists who aren't in HPC
  1. Description of the standard
  2. Reference guide, user guide for experienced HPC users
  3. Book for general computer science audience
- Documentation for novice programmers: very rare
- Tiny percentage of physical scientists & engineers ever learn these standards

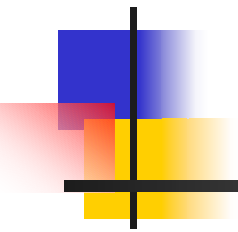
# Why Bother Teaching Novices?

- Application scientists & engineers typically know their applications very well, much better than a collaborating computer scientist would ever be able to.
- Because of Linux clusters, CSE is now affordable.
- Commercial code development lags behind the research community.
- Many potential CSE users don't need full time CSE and HPC staff, just some help.
- Today's novices are tomorrow's top researchers, especially because today's top researchers will eventually retire.

# Questions for Teaching Novices

- What are the fundamental issues of CSE?
- What are the fundamental issues of HPC?
- How can we express these issues in a way that makes sense to inexperienced programmers?
- Is classroom exposure enough, or is one-on-one contact with experts required?

# Computational Science & Engineering



# CSE Hierarchy

- Phenomenon
- Physics
- Mathematics (continuous)
- Numerics (discrete)
- Algorithm
- Implementation
- Port
- Solution
- Analysis
- Verification

# CSE Fundamental Issues

- Physics, mathematics and numerics are addressed well by existing science and engineering curricula, though often in isolation from one another.
- So, instruction should be provided on issues relating primarily to the later items – algorithm, implementation, port, solution, analysis and verification – and on the interrelationships between all of these items.
- Example: algorithm choice

  Typical mistake: solve a linear system by inverting the matrix, without regard for performance, conditioning, or exploiting the properties of the matrix.

# The Five Rules for CSE [8]

1. Know the physics.
2. Control the software.
3. Understand the numerics.
4. Achieve expected behavior.
5. Question unexpected behavior.

# Know the Physics

In general, scientists and engineers know their problems well – they know how to build the mathematical model representing their physical problem.

# Understand the Numerics

This area is less well understood by the scientific and engineering community. The tendency is toward old and often inherently serial algorithms.

At this stage, a researcher is greatly aided by considering two aspects of algorithm development:

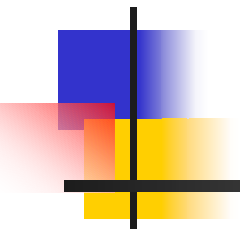
- Do the numerics accurately capture the physical phenomena?
- Is the algorithm appropriate for parallel computing?

# Achieve the Expected Behavior

The testing and validation of any code is essential to develop confidence in the results. Verification is accomplished by applying the code to problems with known solutions and obtaining the expected behavior.

# CSE Implies Multidisciplinary

- CSE is the interface between physics, mathematics and computer science.
- Therefore, finding an effective and efficient way for these disciplines to work together is critically important to success.
- However, that's not typically how CSE is taught; rather, it's taught in the context of a particular application discipline, with relatively little regard for computing issues, especially performance.
- But, performance governs the range of problems that can be tackled.
- Therefore, the traditional approach limits the scope and ambition of new practitioners.



# High Performance Computing

# OSCER

- OU Supercomputing Center for Education & Research
- OSCER is a new multidisciplinary center within OU's Department of Information Technology
- OSCER is for:
  - Undergrad students
  - Grad students
  - Staff
  - Faculty
- OSCER provides:
  - Supercomputing **education**
  - Supercomputing **expertise**
  - Supercomputing **resources**
    - Hardware
    - Software

# HPC Fundamental Issues

- Storage hierarchy
- Parallelism
  - Instruction-level parallelism
  - Multiprocessing
    - Shared Memory Multithreading
    - Distributed Multiprocessing
- High performance compilers
- Scientific libraries
- Visualization
- Grid Computing

# How to Express These Ideas?

- Minimal jargon
- Clearly define every new term in plain English
- Analogies
  - Laptop analogy
  - Jigsaw puzzle analogy
  - Desert islands analogy
- Narratives
- Interaction: instead of just lecturing, ask questions to lead the students to useful approaches
- Followup: not just classroom but also one-on-one
- This approach works not only for inexperienced programmers but also for CS students.

# Supercomputing in Plain English

## An Introduction to High Performance Computing

Henry Neeman, Director
OU Supercomputing Center for Education & Research


# HPC Workshop Topics

1. Overview
2. Storage Hierarchy
3. Instruction Level Parallelism
4. Stupid Compiler Tricks (high performance compilers)
5. Shared Memory Multithreading (OpenMP)
6. Distributed Multiprocessing (MPI)
7. Grab Bag: libraries, I/O, visualization

Sample slides from workshops follow.

# What is Supercomputing About?

## Size

## Speed

# What is the Storage Hierarchy?

[9]

**Fast, expensive, few**

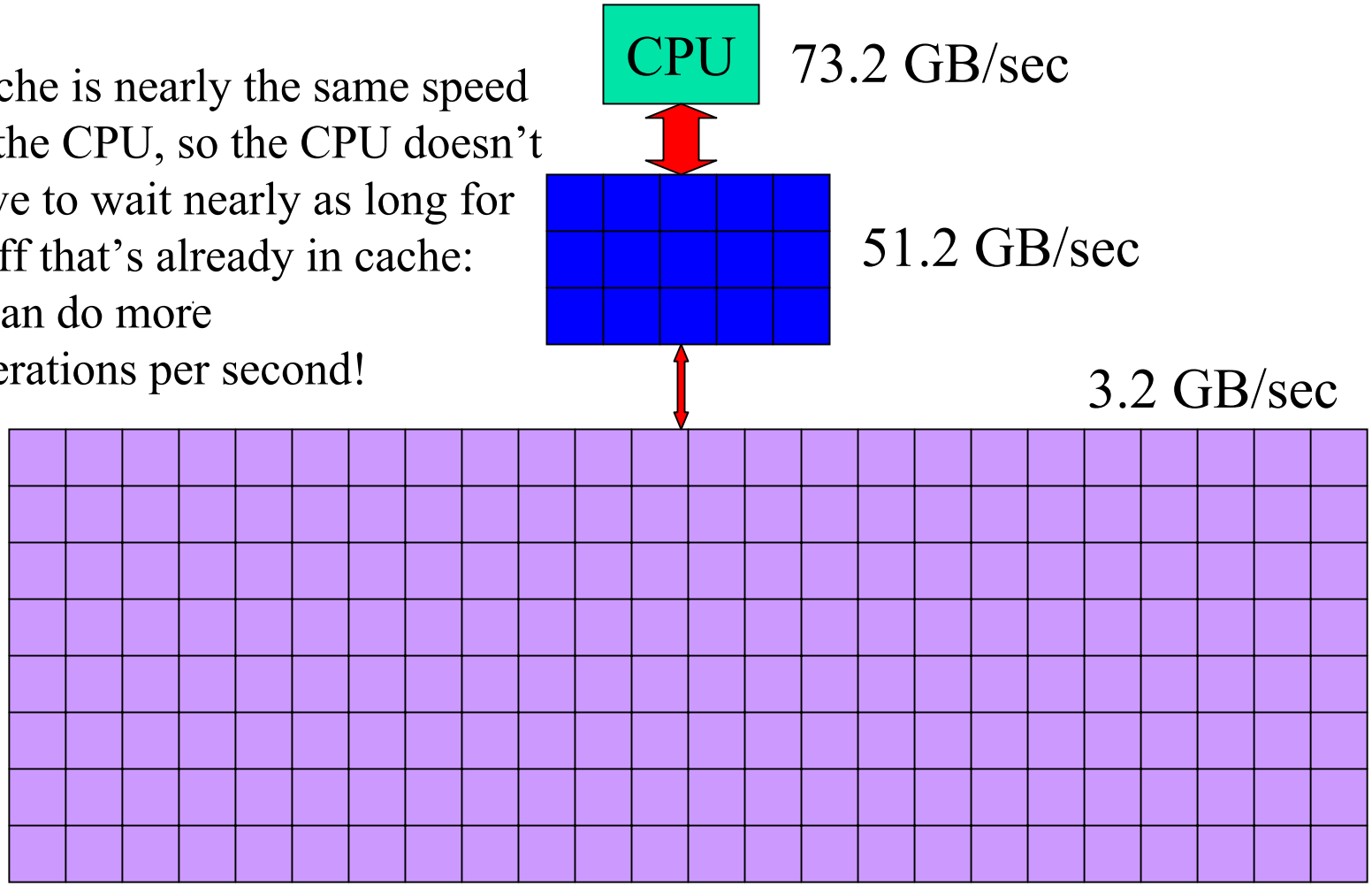
- Registers
- Cache memory
- Main memory (RAM)
- Hard disk
- Removable media (e.g., CDROM)
- Internet

**Slow, cheap, a lot**

[10]

# Why Have Cache?

Cache is nearly the same speed as the CPU, so the CPU doesn't have to wait nearly as long for stuff that's already in cache: it can do more operations per second!



# Henry's Laptop

## Dell Latitude C840[11]

- Pentium 4 1.6 GHz w/512 KB L2 Cache
- 512 MB 400 MHz DDR SDRAM
- 30 GB Hard Drive
- Floppy Drive
- DVD/CD-RW Drive
- 10/100 Mbps Ethernet
- 56 Kbps Phone Modem

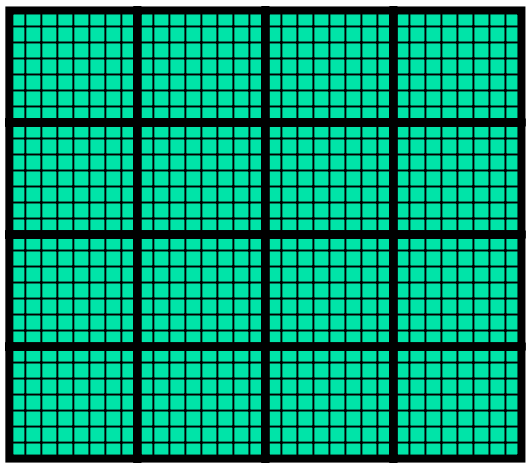
# Storage Speed, Size, Cost

| Henry's Laptop | Registers (Pentium 4 1.6 GHz) | Cache Memory (L2) | Main Memory (400 MHz DDR SDRAM) | Hard Drive | Ethernet (100 Mbps) | CD-RW | Phone Modem (56 Kbps) |
|---|---|---|---|---|---|---|---|
| Speed (MB/sec) [peak] | 73,232[12] (3200 MFLOP/s*) | 52,428 [13] | 3,277 [14] | 100 [15] | 12 | 4 [9] | 0.007 |
| Size (MB) | 304 bytes** [16] | 0.5 | 512 | 30,000 | unlimited | unlimited | unlimited |
| Cost ($/MB) | – | $1200 [17] | $1.17 [17] | $0.009 [17] | charged per month (typically) | $0.0015 [17] | charged per month (typically) |

* MFLOP/s: millions of floating point operations per second

** 8 32-bit integer registers, 8 80-bit floating point registers, 8 64-bit MMX integer registers, 8 128-bit floating point XMM registers

# Tiling



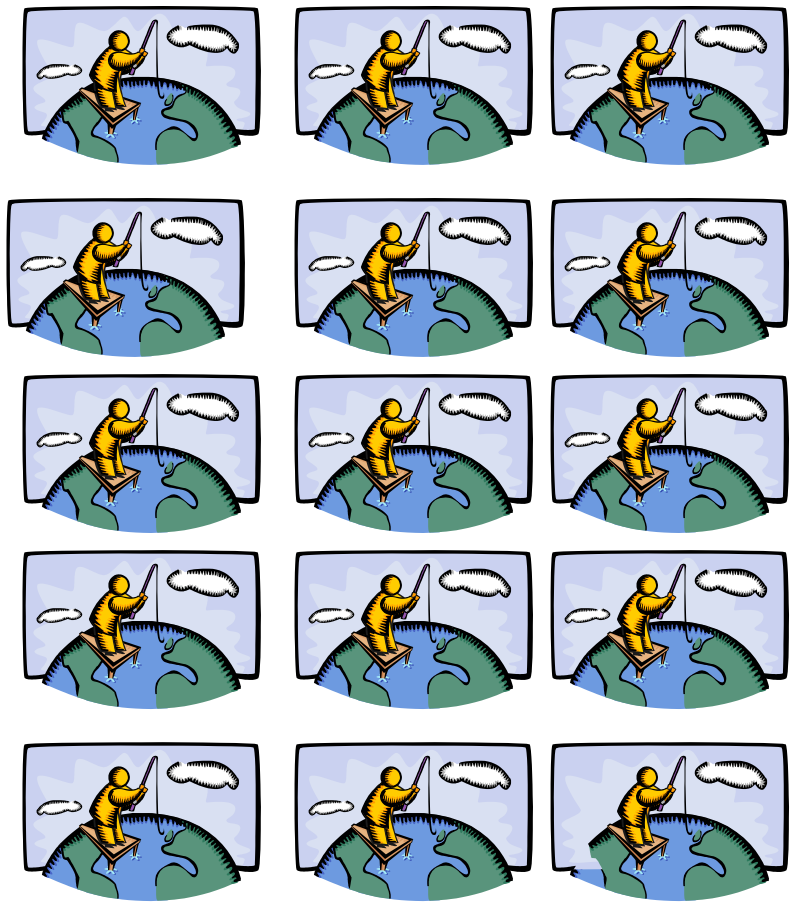
```
SUBROUTINE matrix_matrix_mult_tile (        &
 &              dst, src1, src2, nr, nc, nq, &
 &              rstart, rend, cstart, cend,  &
 &              qstart, qend)
  DO c = cstart, cend
    DO r = rstart, rend
      if (qstart == 1) dst(r,c) = 0.0
      DO q = qstart, qend
        dst(r,c)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

# Parallelism

Parallelism means doing multiple things at the same time: you can get more work done in the same time.

Less fish …



More fish!

# Instruction Level Parallelism

- Superscalar: perform multiple operations at the same time
- Pipeline: start performing an operation on one piece of data while continuing the same operation on another piece of data
- Superpipeline: perform multiple pipelined operations at the same time
- Vector: load multiple pieces of data into special registers in the CPU and perform the same operation on all of them at the same time

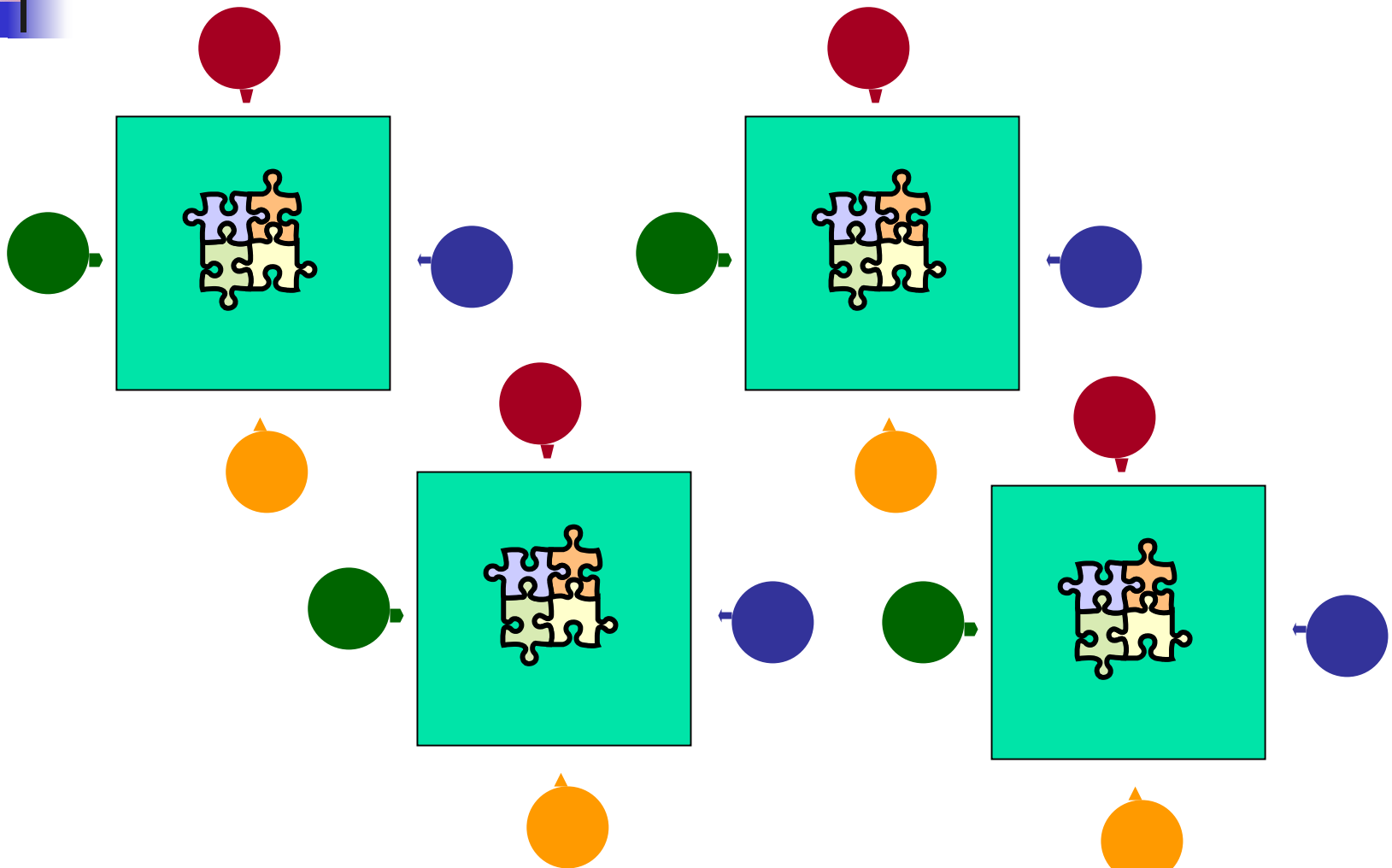
# Why You Shouldn't Panic

In general, the compiler and the CPU will do most of the heavy lifting for instruction-level parallelism.

# BUT:

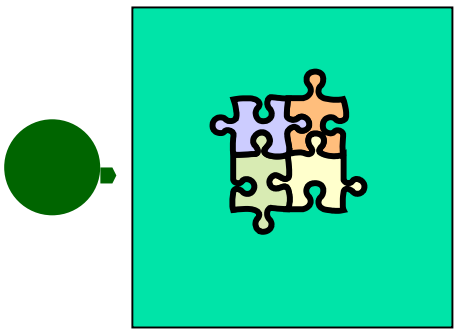
You need to be aware of ILP, because how your code is structured affects how much ILP the compiler and the CPU can give you.

# The Jigsaw Puzzle Analogy



# The Jigsaw Puzzle Analogy (2002)

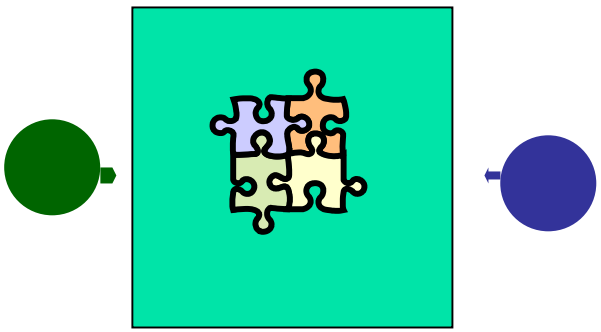
# Serial Computing



Suppose you want to do a jigsaw puzzle that has, say, a thousand pieces.

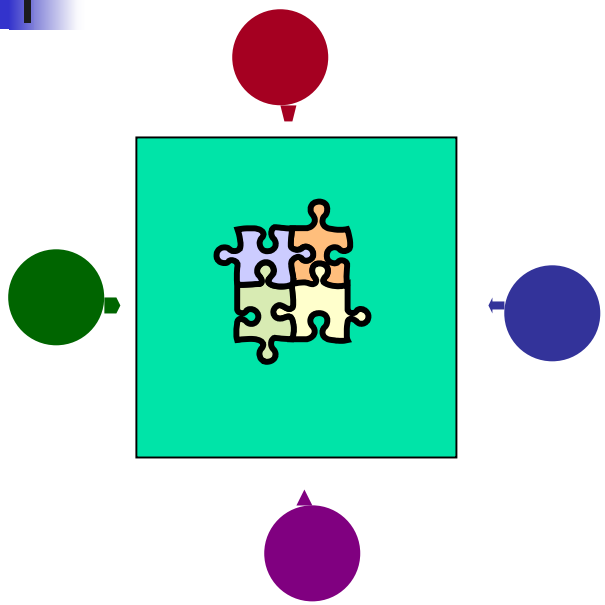
We can imagine that it'll take you a certain amount of time. Let's say that you can put the puzzle together in an hour.

# Shared Memory Parallelism



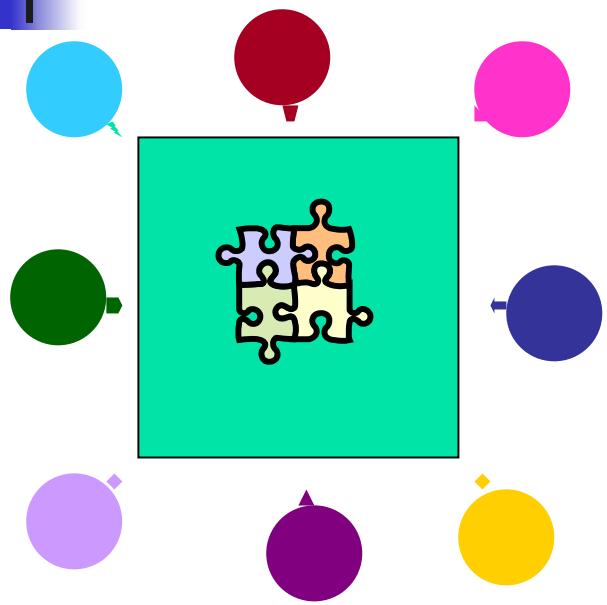
If Julie sits across the table from you, then she can work on her half of the puzzle and you can work on yours. Once in a while, you'll both reach into the pile of pieces at the same time (you'll contend for the same resource), which will cause a little bit of slowdown. And from time to time you'll have to work together (communicate) at the interface between her half and yours. The speedup will be nearly 2-to-1: y'all might take 35 minutes instead of 30.

# The More the Merrier?



Now let's put Lloyd and Jerry on the other two sides of the table. Each of you can work on a part of the puzzle, but there'll be a lot more contention for the shared resource (the pile of puzzle pieces) and a lot more communication at the interfaces. So y'all will get noticeably less than a 4-to-1 speedup, but you'll still have an improvement, maybe something like 3-to-1: the four of you can get it done in 20 minutes instead of an hour.

# Diminishing Returns



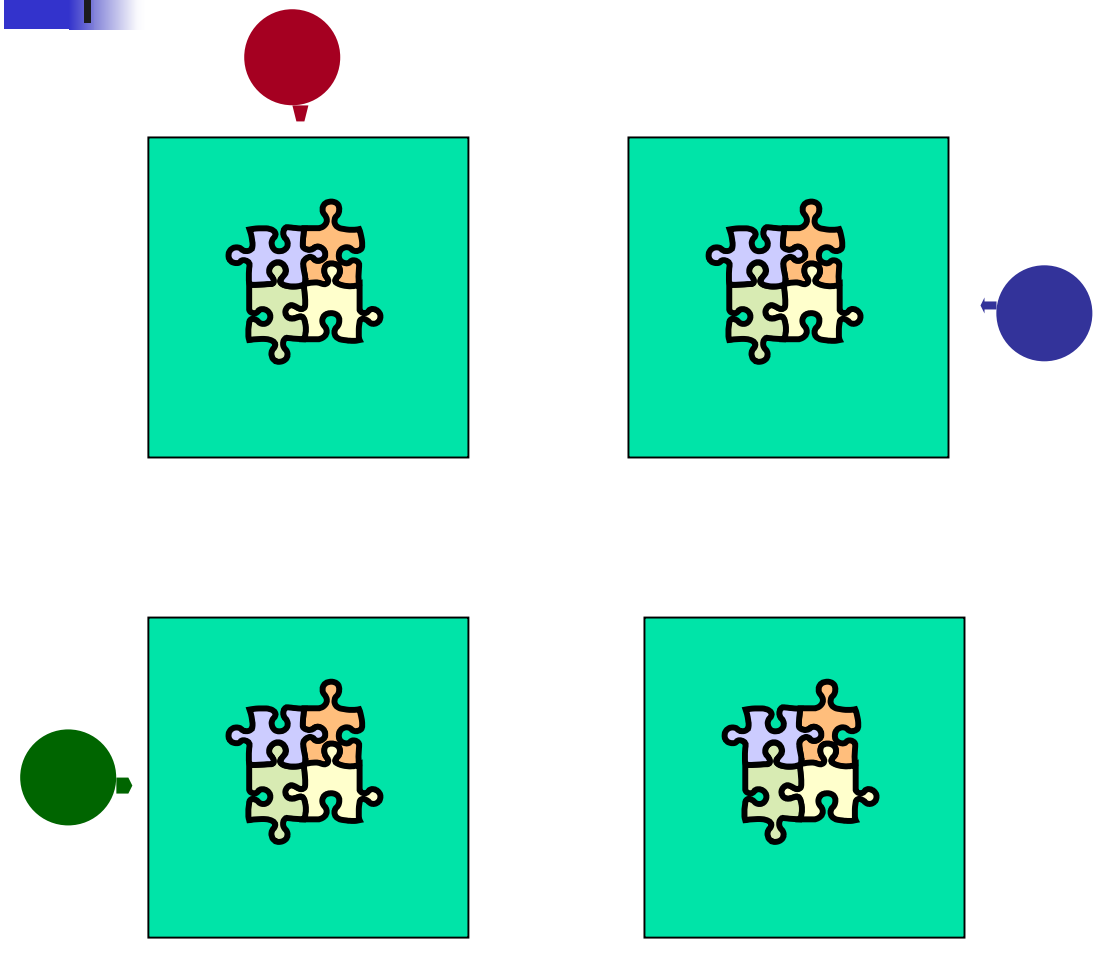
If we now put Cathy and Denese and Chenmei and Nilesh on the corners of the table, there's going to be a whole lot of contention for the shared resource, and a lot of communication at the many interfaces. So the speedup y'all get will be much less than we'd like; you'll be lucky to get 5-to-1.

So we can see that adding more and more workers onto a shared resource is eventually going to have a diminishing return.

# Distributed Parallelism

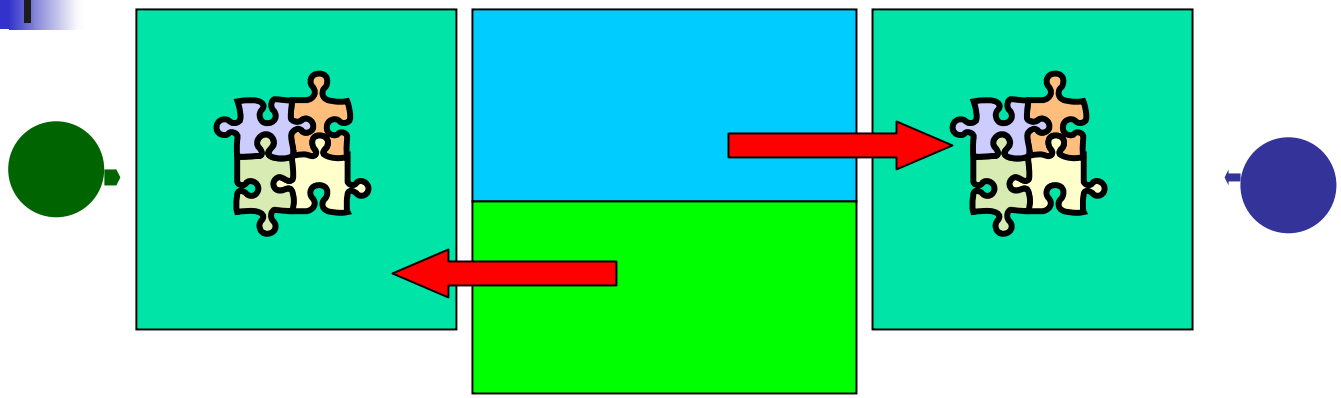
Now let's try something a little different. Let's set up two tables, and let's put you at one of them and Julie at the other. Let's put half of the puzzle pieces on your table and the other half of the pieces on Julie's. Now y'all can work completely independently, without any contention for a shared resource. **BUT**, the cost of communicating is **MUCH** higher (you have to scootch your tables together), and you need the ability to split up (decompose) the puzzle pieces reasonably evenly, which may be tricky to do for some puzzles.

# More Distributed Processors



It's a lot easier to add more processors in distributed parallelism. But, you always have to be aware of the need to decompose the problem and to communicate between the processors. Also, as you add more processors, it may be harder to load balance the amount of work that each processor gets.

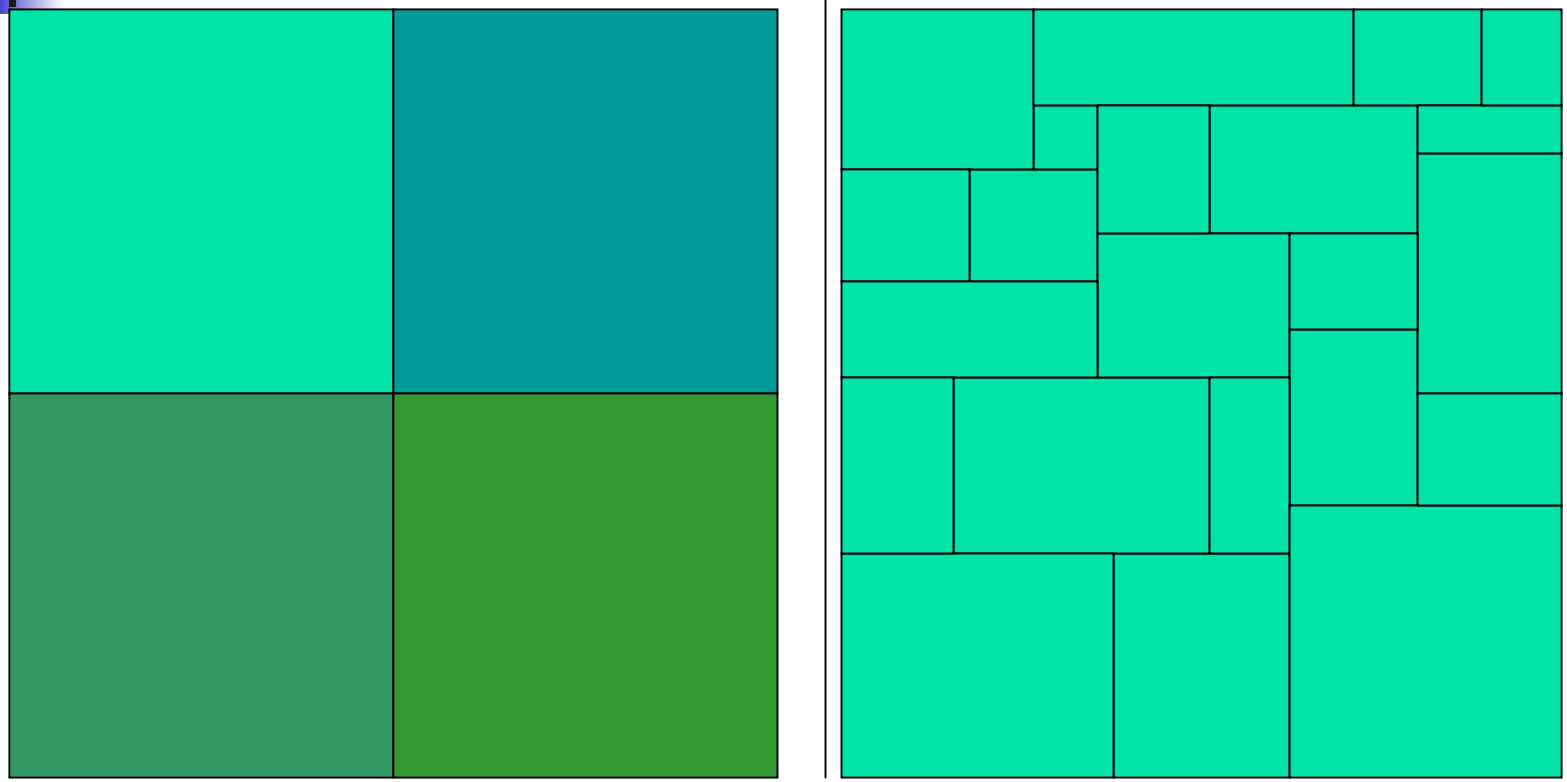
# Load Balancing



Load balancing means giving everyone roughly the same amount of work to do.

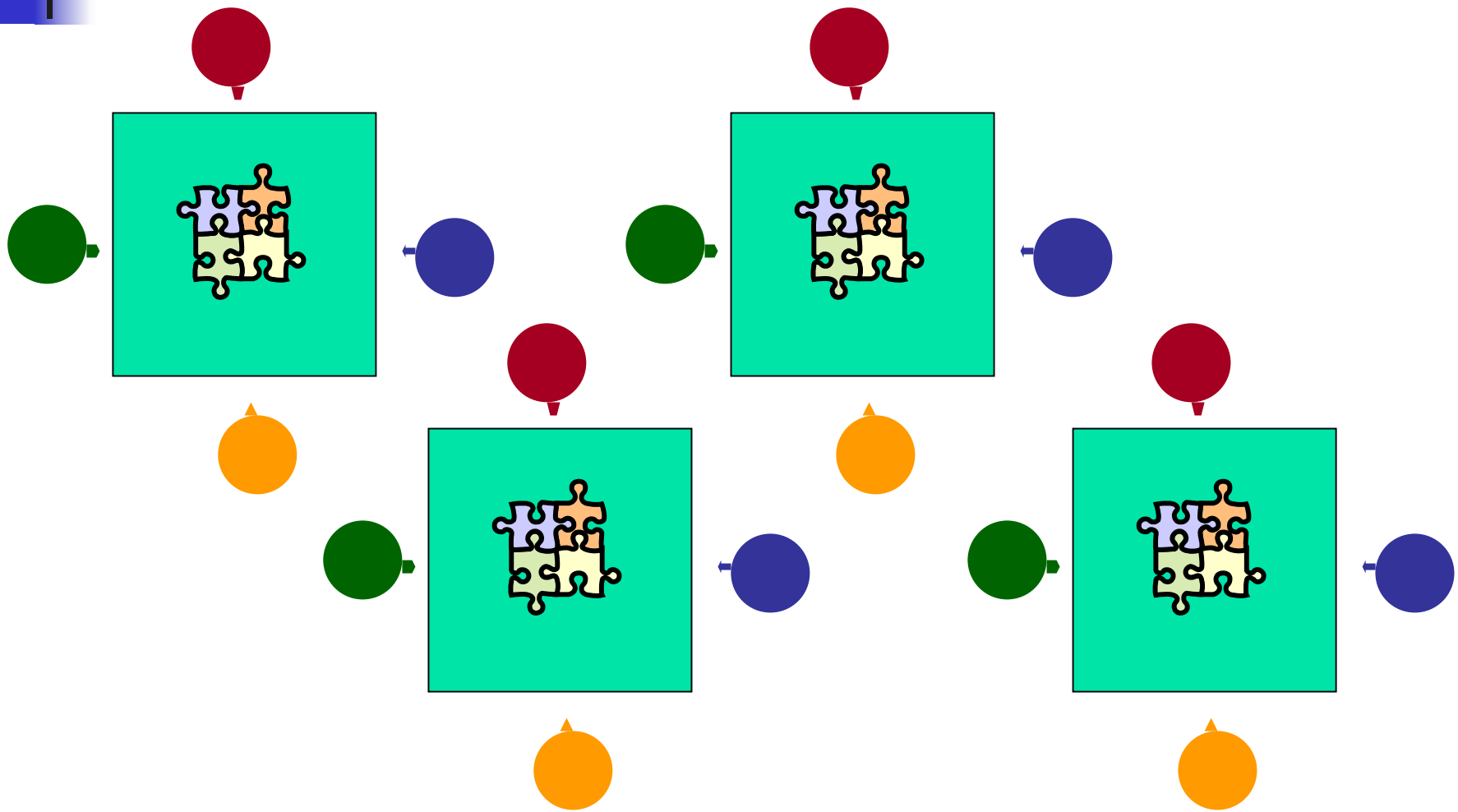
For example, if the jigsaw puzzle is half grass and half sky, then you can do the grass and Julie can do the sky, and then y'all only have to communicate at the horizon – and the amount of work that each of you does on your own is roughly equal. So you'll get pretty good speedup.

# Load Balancing



Load balancing can be easy, if the problem splits up into chunks of roughly equal size, with one chunk per processor. Or load balancing can be very hard.

# Hybrid Parallelism



# The Desert Islands Analogy

# An Island Hut

Imagine you're on an island in a little hut.

Inside the hut is a desk.

On the desk is a phone, a pencil, a calculator, a piece of paper with numbers, and a piece of paper with instructions.

# Instructions

The instructions are split into two kinds:

- Arithmetic/Logical: e.g.,
  - Add the 27th number to the 239th number
  - Compare the 96th number to the 118th number to see whether they are equal
- Communication: e.g.,
  - dial 555-0127 and leave a voicemail containing the 962nd number
  - call your voicemail box and collect a voicemail from 555-0063 and put that number in the 715th slot

# Is There Anybody Out There?

If you're in a hut on an island, you aren't specifically aware of anyone else.

Especially, you don't know whether anyone else is working on the same problem as you are, and you don't know who's at the other end of the phone line.

All you know is what to do with the voicemails you get, and what phone numbers to send voicemails to.

# Someone Might Be Out There

Now suppose that Julie is on another island somewhere, in the same kind of hut, with the same kind of equipment.

Suppose that she has the same list of instructions as you, but a different set of numbers (both data and phone numbers).

Like you, she doesn't know whether there's anyone else working on her problem.

# Even More People Out There

Now suppose that Lloyd and Jerry are also in huts on islands.

Suppose that each of the four has the exact same list of instructions, but different lists of numbers.

And suppose that the phone numbers that people call are each others'. That is, your instructions have you call Julie, Lloyd and Jerry, Julie's has her call Lloyd, Jerry and you, and so on.

Then you might all be working together on the same problem.

# All Data Are Private

Notice that you can't see Julie's or Lloyd's or Jerry's numbers, nor can they see yours or each other's.

Thus, everyone's numbers are private: there's no way for anyone to share numbers, except by leaving them in voicemails.

# Long Distance Calls: 2 Costs

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 connection charge is large, then you want to make as few calls as possible.

# Like Desert Islands

Distributed parallelism is very much like the Desert Islands analogy:

- Processors are independent of each other.
- All data are private.
- Processes communicate by passing messages (like voicemails).
- The cost of passing a message is split into the latency (connection time) and the bandwidth (time per byte).

# The Importance of Followup

# Why Followup?

- Classroom exposure isn't enough, because in the classroom you can't cover all the technical issues, or how to think about parallel programming in the context of each of dozens of specific applications.
- So, experts have to spend time with student researchers (and, for that matter, faculty and staff researchers) one-on-one (or one-on-few) to work on their specific applications.
- But, the amount of time per research group can be small – maybe an hour a week for 1 to 2 years.

# OSCER Rounds

From left: Civil Engr undergrad from Cornell; CS grad student; OSCER Director; Civil Engr grad student; Civil Engr prof; Civil Engr undergrad

# Why Do Rounds?

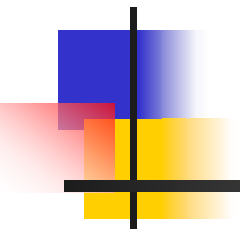
- "The devil is in the details" – and we can't cover all the necessary detail in 7 hours of workshops.
- HPC novices need expert help, but not all that much – an hour or so a week is typically enough, especially once they get going.
- Novices don't need to become experts, and in fact they can't: there's too much new stuff coming out all the time (e.g., Grid computing).
- But, someone should be an expert, and that person should be available to provide useful information.

# HPC Learning Curve

1. Learning Phase: HPC expert learns about the application; application research team learns how basic HPC strategies relate to their application
2. Development Phase: discuss and implement appropriate optimization and parallelization strategies
3. Refinement Phase: initial approaches are improved through profiling, benchmarking, testing, etc

Lots of overlap between these phases

# Summary and Future Work



# CSE/HPC Experts

- Most application research groups don't need a full time CSE and/or HPC expert, but they do need some help (followup).
- So, an institution with one or a few such experts can spread their salaries over dozens of research projects, since each project will only need a modest amount of their time.
- Thus, these experts are cost effective:
  - For each project, they add a lot of value for minimal cost.
  - Their participation in each project raises the probability of each grant proposal being funded, because the proposals are multidisciplinary, have enough CSE and/or HPC expertise to be practicable, and include a strong educational component.
  - The more projects an expert participates in, the broader their range of experience, and so the more value they bring to each new project.
- In a sense, the experts' job is to make themselves obsolete, but to a specific student or project rather than to their institution – "there's plenty more where that came from."

# OU CRCD Project

- Develop CSE & HPC modules
- Teach CSE & HPC modules within nanotechnology course
- Assessment
  - Surveys
    - Pre & post test
    - Attitudinal
  - Programming Project
    - We develop parallel Monte Carlo code.
    - We remove the parallel constructs.
    - Students (re-)parallelize the code, under our supervision and mentoring.
- CSE & HPC modules ported to other courses to ensure broad applicability

# References


[1] S.J. Norton, M. D. Depasquale, *Thread Time: The MultiThreaded Programming Guide*, 1st ed, Prentice Hall, 1996, p. 38.
[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994. http://www.netlib.org/pvm3/book/pvm-book.ps
[3] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard.* 1994. http://www.openmp.org/specs/mp-documents/fspec10.pdf
[4] P.S. Pacheco, Parallel Programming with MPI. Morgan Kaufmann Publishers Inc., 1997.
[5] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface." 1997. http://www.openmp.org/specs/mp-documents/fspec10.pdf
[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel Programming in OpenMP. Morgan Kaufmann Publishers Inc., 2001.
[7] Globus News Archive. http://www.globus.org/about/news/
[8] Robert E. Peterkin, personal communication, 2002.
[9] http://www.f1photo.com/
[10] http://www.vw.com/newbeetle/
[11] http://www.dell.com/us/en/bsd/products/model_latit_latit_c840.htm
[12] R. Gerber, The Software Optimization Cookbook: High-performance Recipes for the Intel Architecture. Intel Press, 2002, pp. 161-168.
[13] http://www.anandtech.com/showdoc.html?i=1460&p=2
[14] ftp://download.intel.com/design/Pentium4/papers/24943801.pdf
[15] http://www.toshiba.com/taecdpd/products/features/MK2018gas-Over.shtml
[16] http://www.toshiba.com/taecdpd/techdocs/sdr2002/2002spec.shtml
[17] ftp://download.intel.com/design/Pentium4/manuals/24896606.pdf
[18] http://www.pricewatch.com/
[19] K. Dowd and C. Severance, *High Performance Computing,* 2nd ed. O'Reilly, 1998, p. 16.