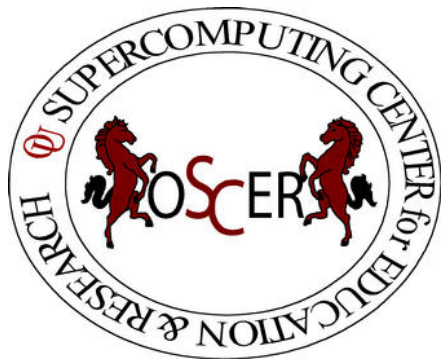
Supercomputing in Plain English

An Introduction to High Performance Computing

Part I: Overview: What the Heck is Supercomputing?

Henry Neeman, Director
OU Supercomputing Center for Education & Research



Goals of These Workshops

- To introduce undergrads, grads, staff and faculty to supercomputing issues
- To provide a common language for discussing supercomputing issues when we meet to work on your research
- **NOT**: to teach everything you need to know about supercomputing – that can't be done in a handful of hourlong workshops!

What is Supercomputing?

Supercomputing is the biggest, fastest computing **right this minute**.

Likewise, a **supercomputer** is the biggest, fastest computer right this minute.

So, the definition of supercomputing is constantly changing.

Rule of Thumb: a supercomputer is 100 to 10,000 times as powerful as a PC.

Jargon: supercomputing is also called High Performance Computing (HPC).

What is Supercomputing About?

Size

Speed

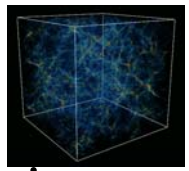
What is Supercomputing About?

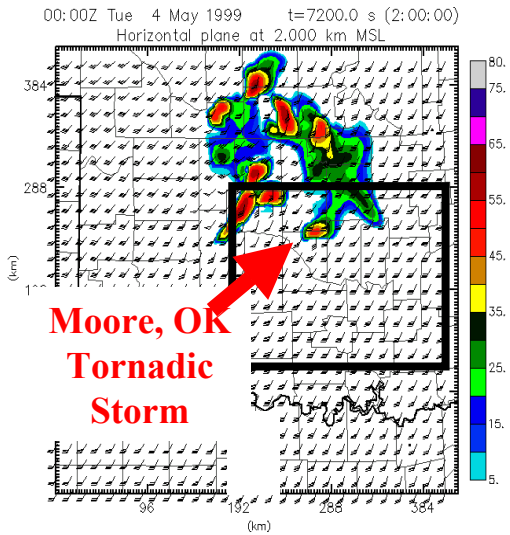
- **Size**: many problems that are interesting to scientists and engineers can't fit on a PC – usually because they need more than 2 GB of RAM, or more than 60 GB of hard disk.
- **Speed**: many problems that are interesting to scientists and engineers would take a very very long time to run on a PC: months or even years. But a problem that would take a month on a PC might take only a few hours on a supercomputer.

What is HPC Used For?

- Simulation of physical phenomena, such as
  - Weather forecasting
  - Star formation
  - Hydrocarbon reservoir simulation

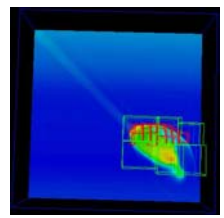
[15]





May 3 1999[1]

- Data mining: finding needles of information in a haystack of data, such as
  - Gene sequencing
  - Signal processing
  - Detecting storms that could produce tornados
- Visualization: turning a vast sea of data into pictures that a scientist can understand



[14]

What is OSCER?

- OSCER is a new multidisciplinary center within IT
- OSCER is for:
  - Undergrad students
  - Grad students
  - Staff
  - Faculty
- OSCER provides:
  - Supercomputing **education**
  - Supercomputing **expertise**
  - Supercomputing **resources**
    - Hardware
    - Software

Who at OU Uses HPC?

- Aerospace Engineering
- Astronomy
- Biochemistry
- Chemical Engineering
- Chemistry
- Civil Engineering
- Computer Science
- Electrical Engineering
- Industrial Engineering
- Geography
- Geophysics
- Management
- Mathematics
- Mechanical Engineering
- Meteorology
- Microbiology
- Molecular Biology
- OK Biological Survey
- Petroleum Engineering
- Physics
- Surgery
- Zoology

Note: some of these aren't using HPC yet, but plan to.

OSCER History

- Aug 2000: founding of OU High Performance Computing interest group
- Nov 2000: first meeting of OUHPC and OU Chief Information Officer Dennis Aebersold
- Feb 2001: meeting between OUHPC, CIO and VP for Research Lee Williams; draft white paper about HPC at OU released
- Apr 2001: Henry Neeman named Director of HPC for Department of Information Technology
- July 2001: draft OSCER charter released

OSCER History (continued)

- Aug 31 2001: OSCER founded; first supercomputing workshop presented
- Nov 2001: hardware bids solicited and received
- Dec 2001: OU Board of Regents approves purchase of supercomputers
- Apr & May 2002: supercomputers delivered
- Sep 12-13 2002: first OU Supercomputing Symposium
- Oct 2002: first paper about OSCER's education strategy published

What Does OSCER Do? Teaching

Supercomputing in Plain English

An Introduction to High Performance Computing

Henry Neeman, Director
OU Supercomputing Center for Education & Research

What Does OSCER Do? Rounds

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

OSCER Hardware: IBM Regatta

- 32 Power4 CPUs
- 32 GB RAM
- 200 GB disk
- OS: AIX 5.1
- Peak speed: 140 billion calculations per second
- Programming model:
  - shared memory
  - multithreading

OSCER Hardware: Linux Cluster

264 Pentium4 CPUs

264 GB RAM

2.5 TB disk

OS: Red Hat Linux 7.3

Peak speed: over a trillion calculations per second

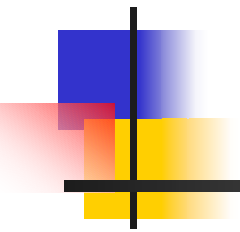
Programming model:

distributed multiprocessing

HPC Issues

- The tyranny of the storage hierarchy
- High performance compilers
- Parallelism: doing many things at the same time
  - Instruction-level parallelism: doing multiple operations at the same time within a single processor (e.g., add, multiply, load and store simultaneously)
  - Multiprocessing: multiple CPUs working on different parts of a problem at the same time
    - Shared Memory Multithreading
    - Distributed Multiprocessing
- Scientific Libraries
- Visualization

A Quick Primer on Hardware



Henry's Laptop

Dell Latitude C840[4]

- 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

Typical Computer Hardware

- Central Processing Unit
- Primary storage
- Secondary storage
- Input devices
- Output devices

Central Processing Unit

- Also called CPU or processor: the "brain"
- Parts
  - Control Unit: figures out what to do next -- e.g., whether to load data from memory, or to add two values together, or to store data into memory, or to decide which of two possible actions to perform (branching)
  - Arithmetic/Logic Unit: performs calculations – e.g., adding, multiplying, checking whether two values are equal
  - Registers: where data reside that are **being used right now**

Primary Storage

- Main Memory
  - Also called RAM ("Random Access Memory")
  - Where data reside when they're **being used by a program that's currently running**
- Cache
  - Small area of much faster memory
  - Where data reside when they're **about to be used** and/or **have been used recently**
- Primary storage is volatile: values in primary storage disappear when the power is turned off.

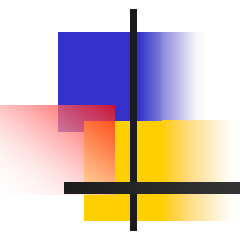
Secondary Storage

- Where data and programs reside that are going to be used **in the future**
- Secondary storage is non-volatile: values **don't** disappear when power is turned off.
- Examples: hard disk, CD, DVD, magnetic tape, Zip, Jaz
- Many are portable: can pop out the CD/DVD/tape/Zip/floppy and take it with you

Input/Output

- Input devices – e.g., keyboard, mouse, touchpad, joystick, scanner
- Output devices – e.g., monitor, printer, speakers

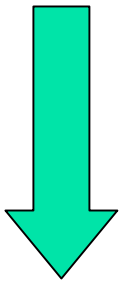
The Tyranny of the Storage Hierarchy



The Storage Hierarchy

[2]

Fast, expensive, few



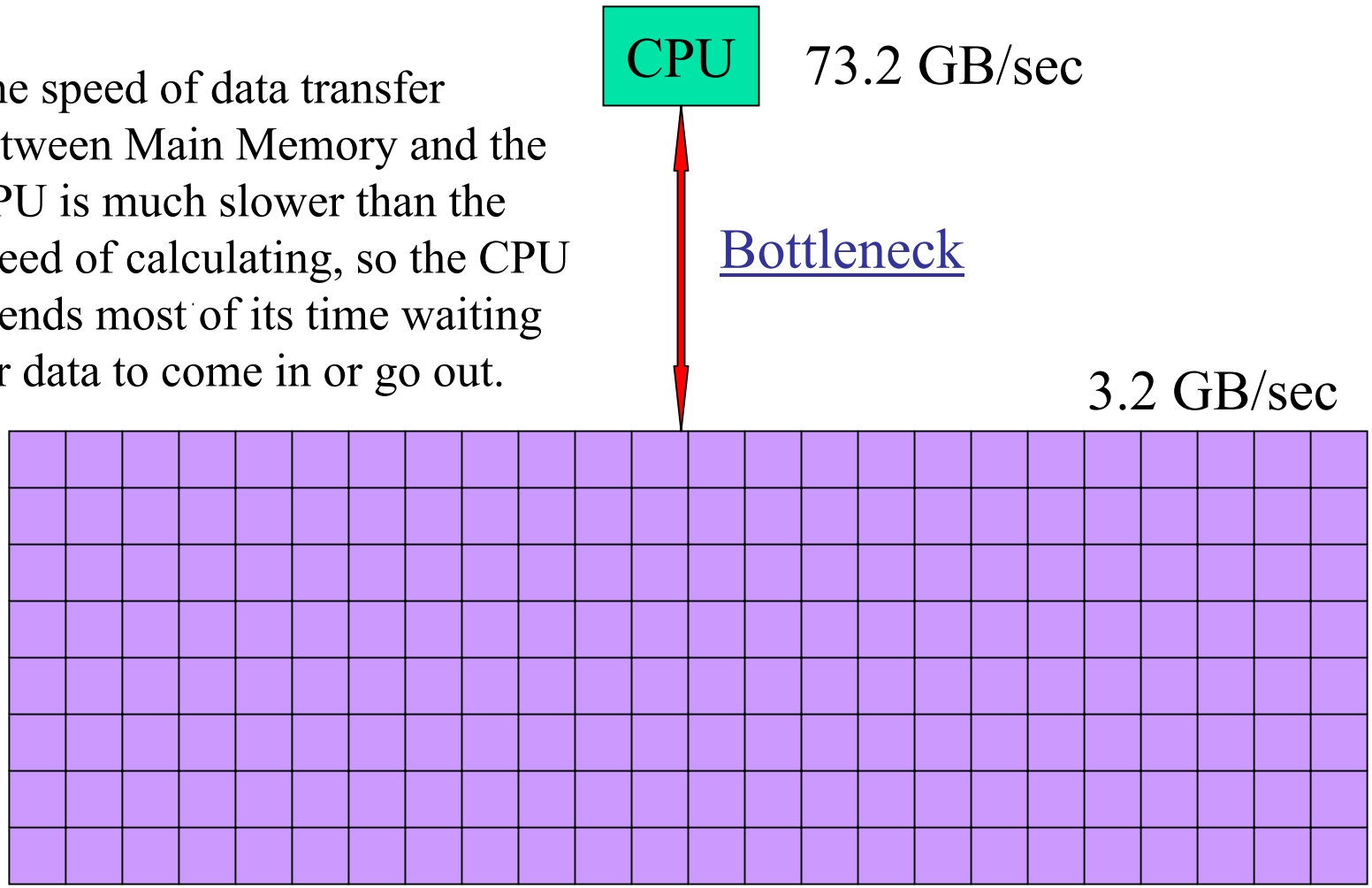
Slow, cheap, a lot

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

[3]

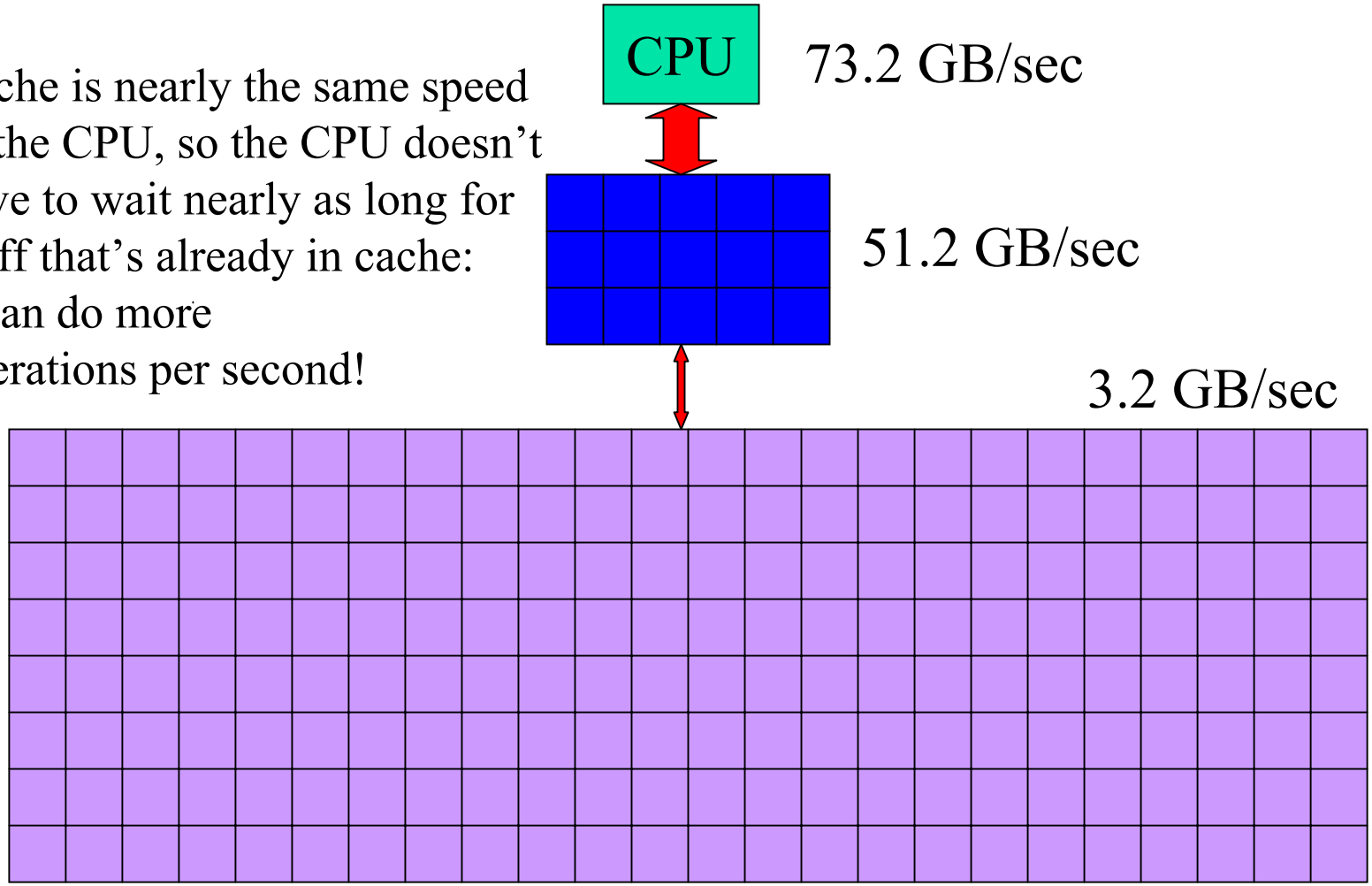
RAM is Slow

The speed of data transfer between Main Memory and the CPU is much slower than the speed of calculating, so the CPU spends most of its time waiting for data to come in or go out.



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, Again

Dell Latitude C840[4]

- 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

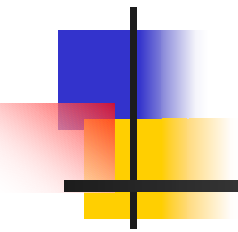
Henry's Laptop	Registers (Pentium 4 1.6 GHz)	Cache Memory (L2)	Main Memory (400 MHz DDR SDRAM)	Hard Drive	Ethernet (100 Mbps)	CD-RW	Phone Modem (56 Kbps)
Speed (MB/sec) [peak]	73,232[5] (3200 MFLOP/s*)	52,428 [6]	3,277 [7]	100 [8]	12	4 [9]	0.007
Size (MB)	304 bytes** [10]	0.5	512	30,000	unlimited	unlimited	unlimited
Cost ($/MB)	–	$1200 [11]	$1.17 [11]	$0.009 [11]	charged per month (typically)	$0.0015 [11]	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

Storage Use Strategies

- Register reuse: do a lot of work on the same data before working on new data.
- Cache reuse: the program is much more efficient if all of the data and instructions fit in cache; if not, try to use what's in cache a lot before using anything that isn't in cache.
- Data locality: try to access data that are near each other in memory before data that are far.
- I/O efficiency: do a bunch of I/O all at once rather than a little bit at a time; don't mix calculations and I/O.

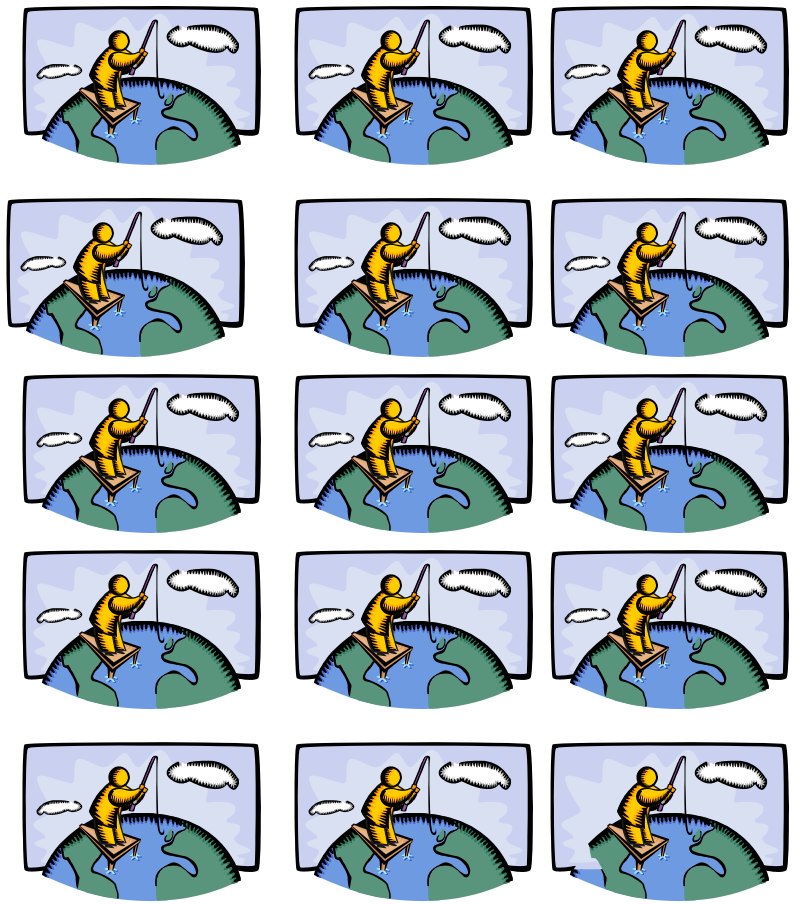


Parallelism

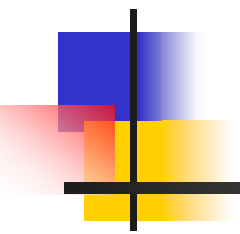
Parallelism

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

Less fish …



More fish!



Parallelism, Part I:

Instruction-Level Parallelism

DON'T PANIC!

Kinds of ILP

- Superscalar: perform multiple operations at the same time
- Pipeline: start performing an operation on one piece of data while finishing 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

What's an Instruction?

- Load a value from a specific address in main memory into a specific register
- Store a value from a specific register into a specific address in main memory
- Add two specific registers together and put their sum in a specific register – or subtract, multiply, divide, square root, etc
- Determine whether two registers both contain nonzero values ("**AND**")
- Jump from one sequence of instructions to another (branch)
- … and so on

DON'T PANIC!

Scalar Operation

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

How would this statement be executed?

1. Load **a** into register **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**

Does Order Matter?

```
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**

In the cases where order doesn't matter, we say that the operations are independent of one another.

Superscalar Operation

```
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**

So, we go from 8 operations down to 5.

Superscalar Loops

```
DO i = 1, n
   z(i) = a(i)*b(i) + c(i)*d(i)
END DO !! i = 1, n
```

Each of the iterations is completely independent of all of the other iterations; e.g.,

```
z(1) = a(1)*b(1) + c(1)*d(1)
```

has nothing to do with

```
z(2) = a(2)*b(2) + c(2)*d(2)
```

Operations that are independent of each other can be performed in parallel.

Superscalar Loops

```
for (i = 0; i < n; i++) {
   z[i] = a[i]*b[i] + c[i]*d[i];
} /* for i */
```

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

Example: IBM Power4

8-way Superscalar: can execute up to 8 operations at the same time[12]

- 2 integer arithmetic or logical operations, and
- 2 floating point arithmetic operations, and
- 2 memory access (load or store) operations, and
- 1 branch operation, and
- 1 conditional operation

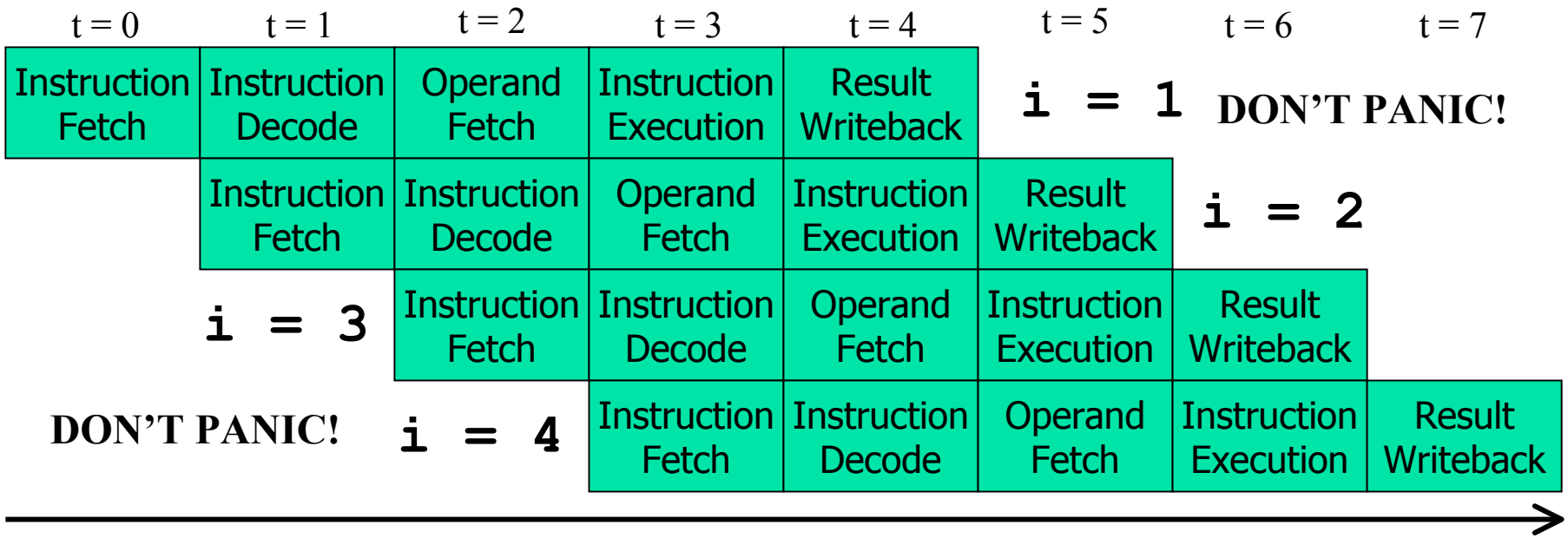
DON'T PANIC!

Pipelining

Pipelining is like an assembly line or a bucket brigade.

- An operation consists of multiple stages.
- After a set of operands complete a particular stage, they move into the next stage.
- Then, another set of operands can move into the stage that was just abandoned.

Pipelining Example



If each stage takes, say, one CPU cycle, then once the loop gets going, each iteration of the loop only increases the total time by one cycle. So a loop of length 1000 takes only 1004 cycles. [13]

Multiply Is Better Than Divide

In most (maybe all) CPU types, adds and subtracts execute very quickly. So do multiplies.

But divides take much longer to execute, typically 5 to 10 times longer than multiplies.

More complicated operations, like square root, exponentials, trigonometric functions and so on, take even longer.

Also, on some CPU types, divides and other complicated operations aren't pipelined.

Superpipelining

Superpipelining is a combination of superscalar and pipelining.

So, a superpipeline is a collection of multiple pipelines that can operate simultaneously.

In other words, several different operations can execute simultaneously, and each of these operations can be broken into stages, each of which is filled all the time.

So you can get multiple operations per CPU cycle.

For example, a IBM Power4 can have over 200 different operations "in flight" at the same time.[12]

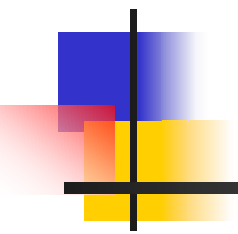
DON'T PANIC!

Why You Shouldn't Panic

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

BUT:

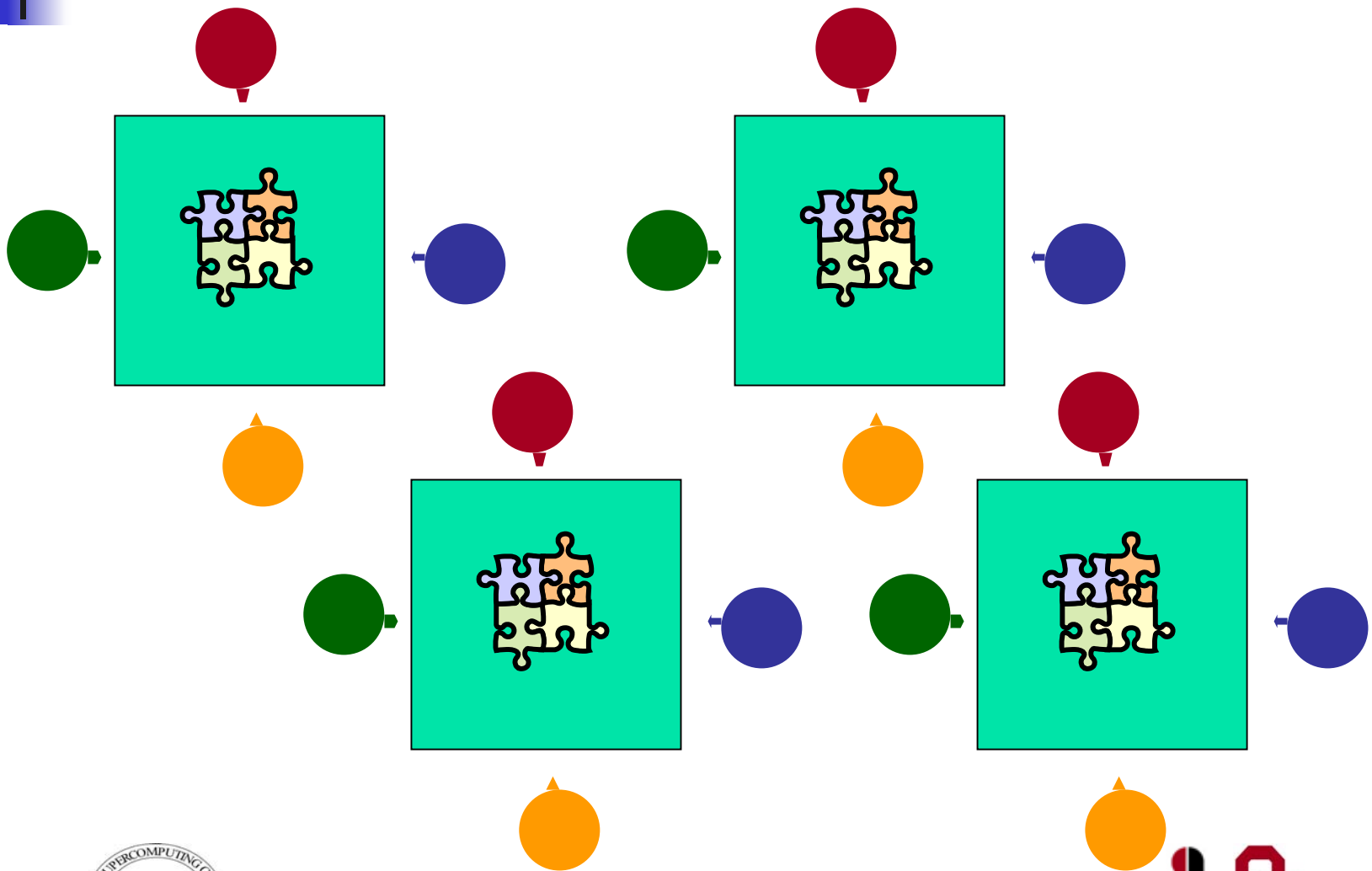
You need to be aware of ILP, because how your code is structured affects how much ILP the compiler and the CPU can give you.



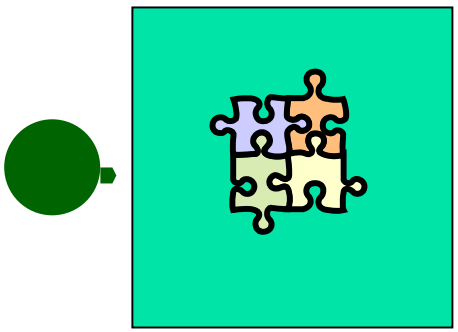
Parallelism, Part II:

Multiprocessing

The Jigsaw Puzzle Analogy



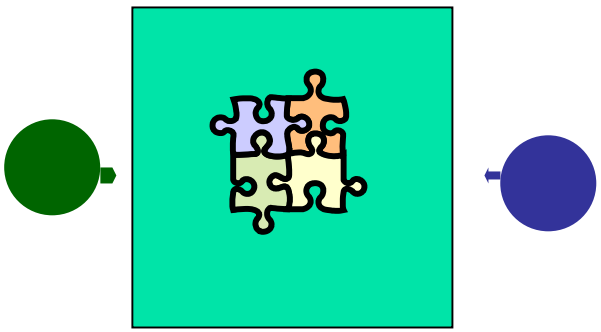
Serial Computing



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

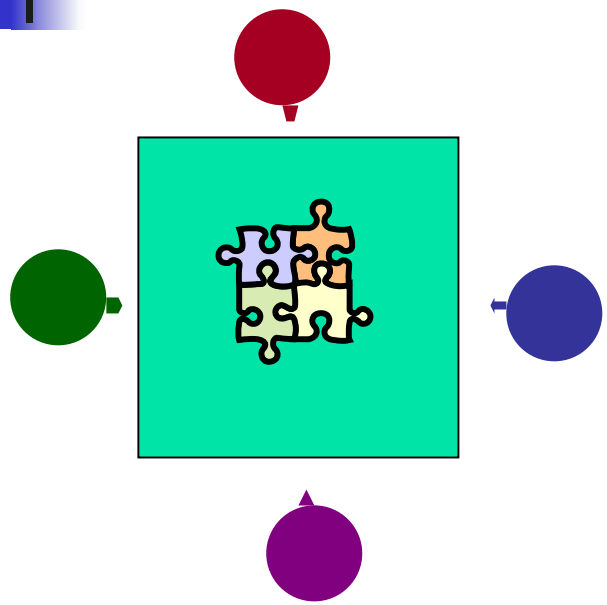
We can imagine that it'll take you a certain amount of time. Let's say that you can put the puzzle together in an hour.

Shared Memory Parallelism



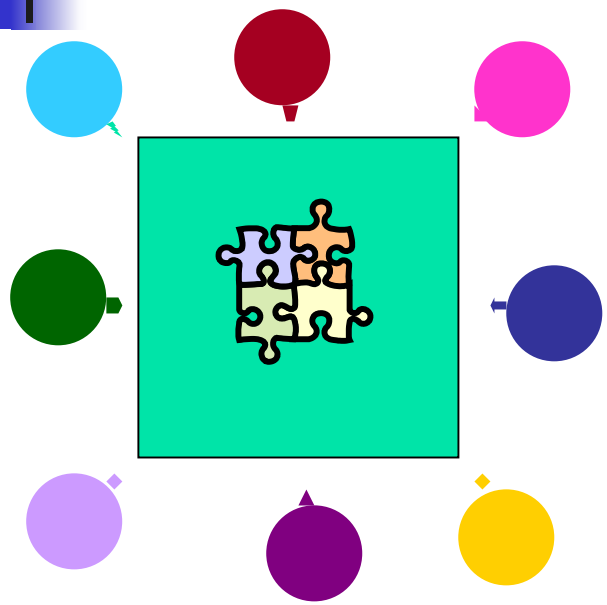
If Julie sits across the table from you, then she can work on her half of the puzzle and you can work on yours. Once in a while, you'll both reach into the pile of pieces at the same time (you'll contend for the same resource), which will cause a little bit of slowdown. And from time to time you'll have 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



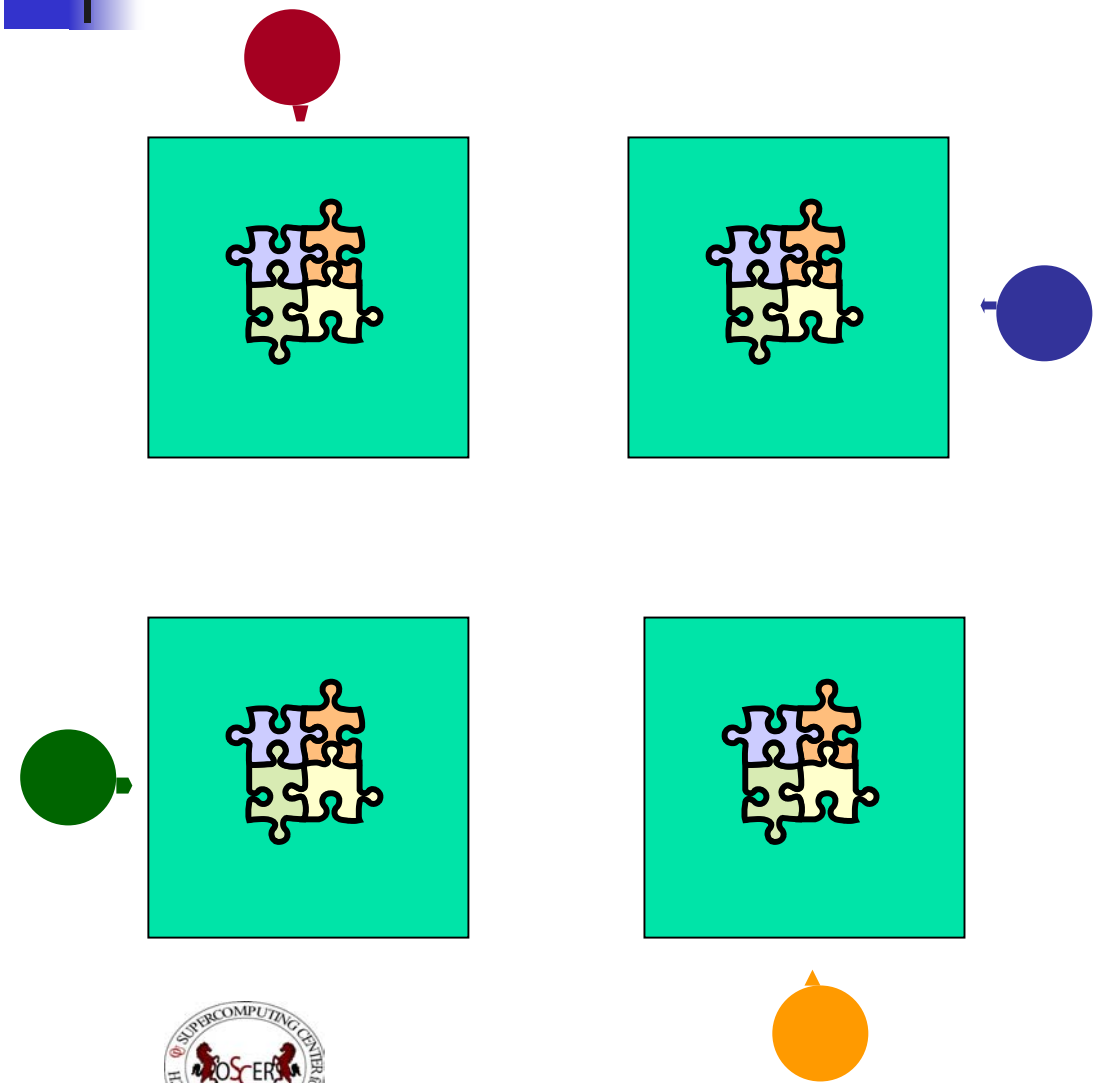
If we now put Cathy and Denese and Chenmei and Nilesh on the corners of the table, there's going to be a whole lot of contention for the shared resource, and a lot of communication at the many interfaces. So the speedup y'all get will be much less than we'd like; you'll be lucky to get 5-to-1.

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

Distributed Parallelism

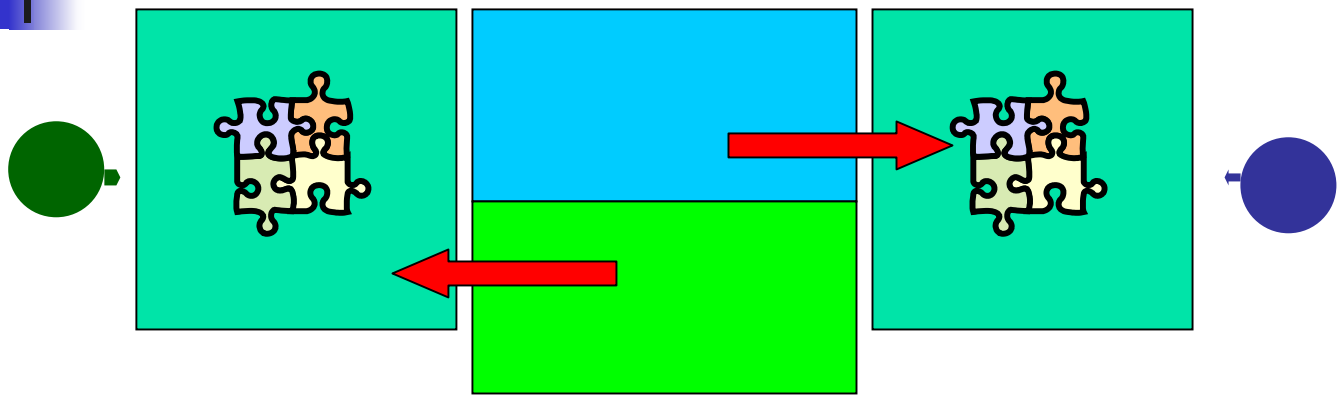
Now let's try something a little different. Let's set up two tables, and let's put you at one of them and Julie at the other. Let's put half of the puzzle pieces on your table and the other half of the pieces on Julie's. Now y'all can work completely independently, without any contention for a shared 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.

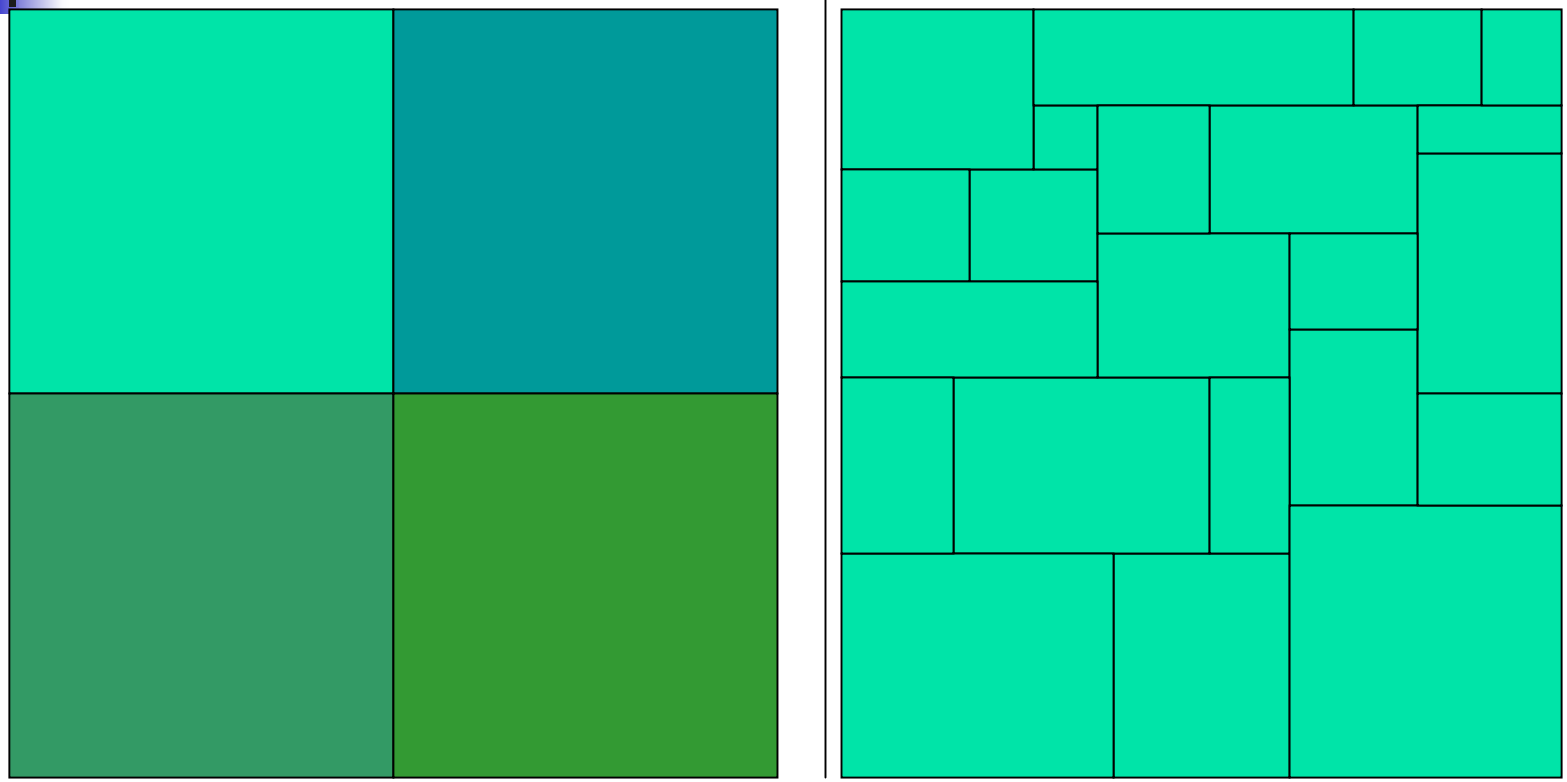
Load Balancing



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

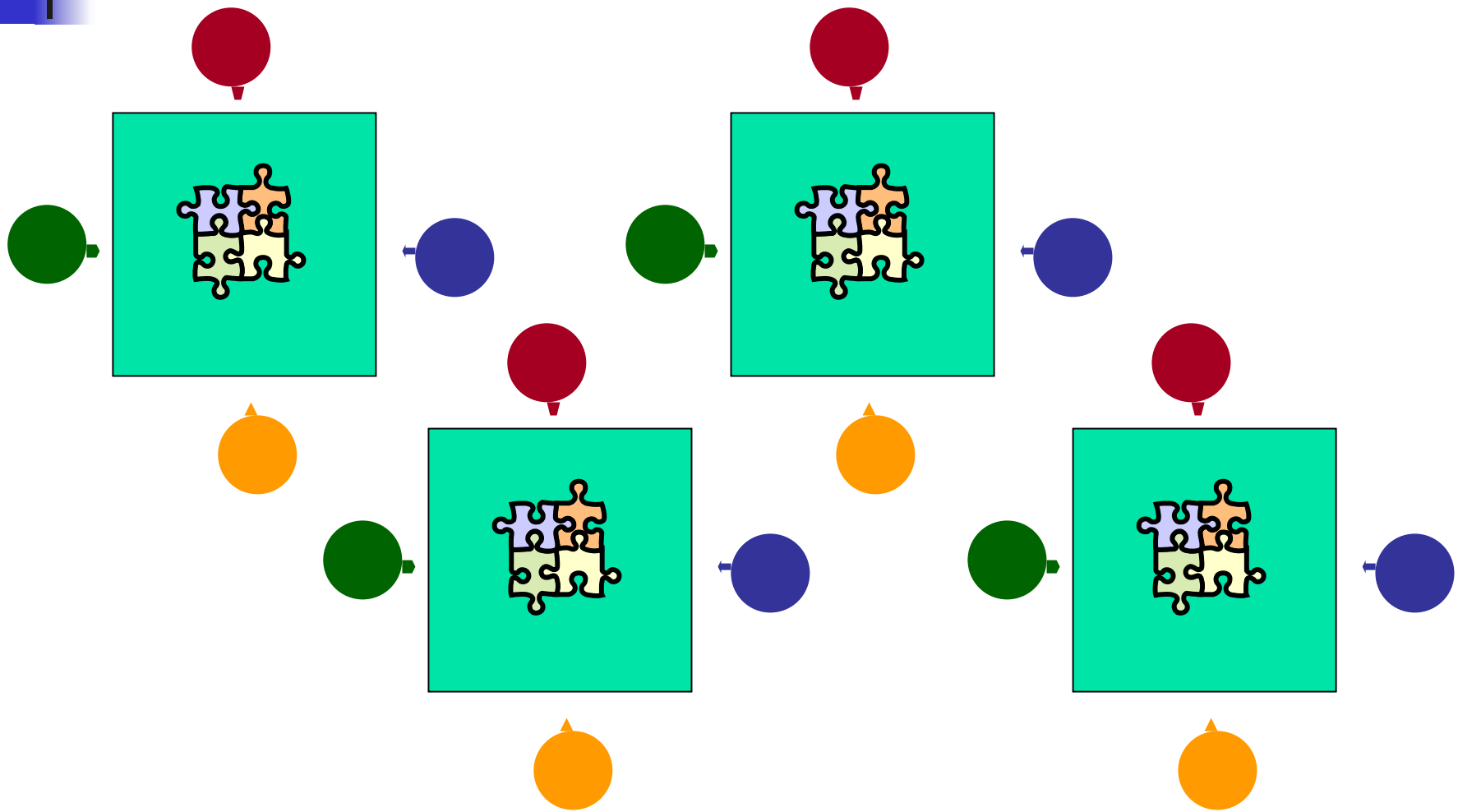
For example, if the jigsaw puzzle is half grass and half sky, then you can do the grass and Julie can do the sky, and then y'all only have to communicate at the horizon – and the amount of work that each of you does on your own is roughly equal. So you'll get pretty good speedup.

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.

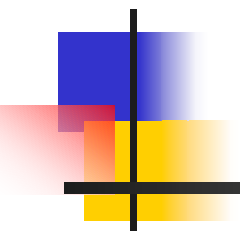
Hybrid Parallelism



Hybrid Parallelism is Good

- Some supercomputers don't support shared memory parallelism, or not very well. When you run a program on those machines, you can turn your program's shared memory parallelism off.
- Some supercomputers don't support distributed parallelism, or not very well. When you run a program on those machines, you can turn your program's distributed parallelism off.
- Some supercomputers support both kinds well.
- So, when you want to use the newest, fastest supercomputer, you can target what it does well without having to rewrite your program.

Why Bother?



Why Bother with HPC at All?

It's clear that making effective use of HPC takes quite a bit of effort, both learning how and developing software.

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?

Why HPC is Worth the Bother

- What HPC gives you that you won't get elsewhere is the ability to do bigger, better, more exciting science. If your code can run faster, that means that you can tackle much bigger problems in the same amount of time that you used to need for smaller problems.
- HPC is important not only for its own sake, but also because what happens in HPC today will be on your desktop in about 15 years: it puts you ahead of the curve.

Your Fantasy Problem

For those of you with current research projects:

1. Get together with your research group.
2. Imagine that you had an infinitely large, infinitely fast computer.
3. What problems would you run on it?

References

[1] "Update on the Collaborative Radar Acquisition Field Test (CRAFT): Planning for the Next Steps." Presented to NWS Headquarters August 30 2001.

[2] http://www.f1photo.com/

[3] http://www.vw.com/newbeetle/

[4] http://www.dell.com/us/en/bsd/products/model_latit_latit_c840.htm

[5] Richard Gerber, The Software Optimization Cookbook: High-performance Recipes for the Intel Architecture. Intel Press, 2002, pp. 161-168.

[6] http://www.anandtech.com/showdoc.html?i=1460&p=2

[7] ftp://download.intel.com/design/Pentium4/papers/24943801.pdf

[8] http://www.toshiba.com/taecdpd/products/features/MK2018gas-Over.shtml

[9] http://www.toshiba.com/taecdpd/techdocs/sdr2002/2002spec.shtml

[10] ftp://download.intel.com/design/Pentium4/manuals/24896606.pdf

[11] http://www.pricewatch.com/

[12] Steve Behling et al, *The POWER4 Processor Introduction and Tuning Guide*, IBM, 2001, p. 8.

[13] Kevin Dowd and Charles Severance, *High Performance Computing,* 2nd ed. O'Reilly, 1998, p. 16.

[14] See http://scarecrow.caps.ou.edu/~hneeman/hamr.html for details.

[15] Image by Greg Bryan, MIT: http://zeus.ncsa.uiuc.edu:8080/chdm_script.html