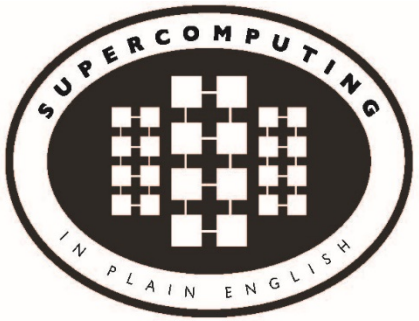
Supercomputing in Plain English

Shared Memory Multithreading

Henry Neeman, University of Oklahoma
Director, OU Supercomputing Center for Education & Research (OSCER)
Assistant Vice President, Information Technology – Research Strategy Advisor
Associate Professor, Gallogly College of Engineering
Adjunct Associate Professor, School of Computer Science
Tuesday February 27 2018



This is an experiment!

It's the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the phone bridge to fall back on.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

At OU, we will turn off the sound on all conferencing technologies.

That way, we won't have problems with **echo cancellation**.

Of course, that means we cannot hear questions.

So for questions, you'll need to send e-mail:

supercomputinginplainenglish@gmail.com

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Download the Slides Beforehand

Before the start of the session, please download the slides from the Supercomputing in Plain English website:

http://www.oscer.ou.edu/education/

That way, if anything goes wrong, you can still follow along with just audio.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Zoom

Go to:

http://zoom.us/j/979158478

Many thanks Eddie Huebsch, OU CIO, for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

YouTube

You can watch from a Windows, MacOS or Linux laptop or an Android or iOS handheld using YouTube.

Go to YouTube via your preferred web browser or app, and then search for:

`Supercomputing InPlainEnglish`

(**`InPlainEnglish`** is all one word.)

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Twitch

You can watch from a Windows, MacOS or Linux laptop or an Android or iOS handheld using Twitch.

Go to:

http://www.twitch.tv/sipe2018

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Wowza #1

You can watch from a Windows, MacOS or Linux laptop using Wowza from the following URL:

http://jwplayer.onenet.net/streams/sipe.html

If that URL fails, then go to:

http://jwplayer.onenet.net/streams/sipebackup.html

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Wowza #2

Wowza has been tested on multiple browsers on each of:

- Windows 10: IE, Firefox, Chrome, Opera, Safari
- MacOS: Safari, Firefox
- Linux: Firefox, Opera

We've also successfully tested it via apps on devices with:

- Android
- iOS

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Toll Free Phone Bridge

IF ALL ELSE FAILS, you can use our US TOLL phone bridge:

405-325-6688

684 684 #

NOTE: This is for **US** call-ins **ONLY**.

PLEASE MUTE YOURSELF and use the phone to listen.

Don't worry, we'll call out slide numbers as we go.

Please use the phone bridge **ONLY IF** you cannot connect any other way: the phone bridge can handle only 100 simultaneous connections, and we have over 1000 participants.

Many thanks to OU CIO Eddie Huebsch for providing the phone bridge..

Please Mute Yourself

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

(For YouTube, Twitch and Wowza, you don't need to do that, because the information only goes from us to you, not from you to us.)

At OU, we will turn off the sound on all conferencing technologies.

That way, we won't have problems with **echo cancellation**.

Of course, that means we cannot hear questions.

So for questions, you'll need to send e-mail.

PLEASE MUTE YOURSELF.

Questions via E-mail Only

Ask questions by sending e-mail to:

supercomputinginplainenglish@gmail.com

All questions will be read out loud and then answered out loud.

DON'T USE CHAT OR VOICE FOR QUESTIONS!

No one will be monitoring any of the chats, and if we can hear your question, you're creating an **echo cancellation** problem.

PLEASE MUTE YOURSELF.
PLEASE MUTE YOURSELF.

Onsite: Talent Release Form

If you're attending onsite, you **MUST** do one of the following:

- complete and sign the Talent Release Form,

OR

- sit behind the cameras (where you can't be seen) and don't talk at all.

If you aren't onsite, then **PLEASE MUTE YOURSELF.**

TENTATIVE Schedule

Tue Jan 23: Storage: What the Heck is Supercomputing?
Tue Jan 30: The Tyranny of the Storage Hierarchy Part I
Tue Feb 6: The Tyranny of the Storage Hierarchy Part II
Tue Feb 13: Instruction Level Parallelism
Tue Feb 20: Stupid Compiler Tricks
Tue Feb 27: Shared Memory Multithreading
Tue March 6: Distributed Multiprocessing
Tue March 13: **NO SESSION** (Henry business travel)
Tue March 20: **NO SESSION** (OU's Spring Break)
Tue March 27: Applications and Types of Parallelism
Tue Apr 3: Multicore Madness
Tue Apr 10: High Throughput Computing
Tue Apr 17: **NO SESSION** (Henry business travel)
Tue Apr 24: GPGPU: Number Crunching in Your Graphics Card
Tue May 1: Grab Bag: Scientific Libraries, I/O Libraries, Visualization

Thanks for helping!

- OU IT
  - OSCER operations staff (Dave Akin, Patrick Calhoun, Kali McLennan, Jason Speckman, Brett Zimmerman)
  - OSCER Research Computing Facilitators (Jim Ferguson, Horst Severini)
  - Debi Gentis, OSCER Coordinator
  - Kyle Dudgeon, OSCER Manager of Operations
  - Ashish Pai, Managing Director for Research IT Services
  - The OU IT network team
  - OU CIO Eddie Huebsch
- OneNet: Skyler Donahue
- Oklahoma State U: Dana Brunson

This is an experiment!

It's the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the phone bridge to fall back on.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Coming in 2018!

- Coalition for Advancing Digital Research & Education (CADRE) Conference: Apr 17-18 2018 @ Oklahoma State U, Stillwater OK USA
  https://hpcc.okstate.edu/cadre-conference
- Linux Clusters Institute workshops
  http://www.linuxclustersinstitute.org/workshops/
  - Introductory HPC Cluster System Administration: May 14-18 2018 @ U Nebraska, Lincoln NE USA
  - Intermediate HPC Cluster System Administration: Aug 13-17 2018 @ Yale U, New Haven CT USA
- Great Plains Network Annual Meeting: details coming soon
- Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency Aug 5-10 2018, U Oklahoma, Norman OK USA
- PEARC 2018, July 22-27, Pittsburgh PA USA
  https://www.pearc18.pearc.org/
- IEEE Cluster 2018, Sep 10-13, Belfast UK
  https://cluster2018.github.io
- **OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2018, Sep 25-26 2018 @ OU**
- SC18 supercomputing conference, Nov 11-16 2018, Dallas TX USA
  http://sc18.supercomputing.org/

Outline

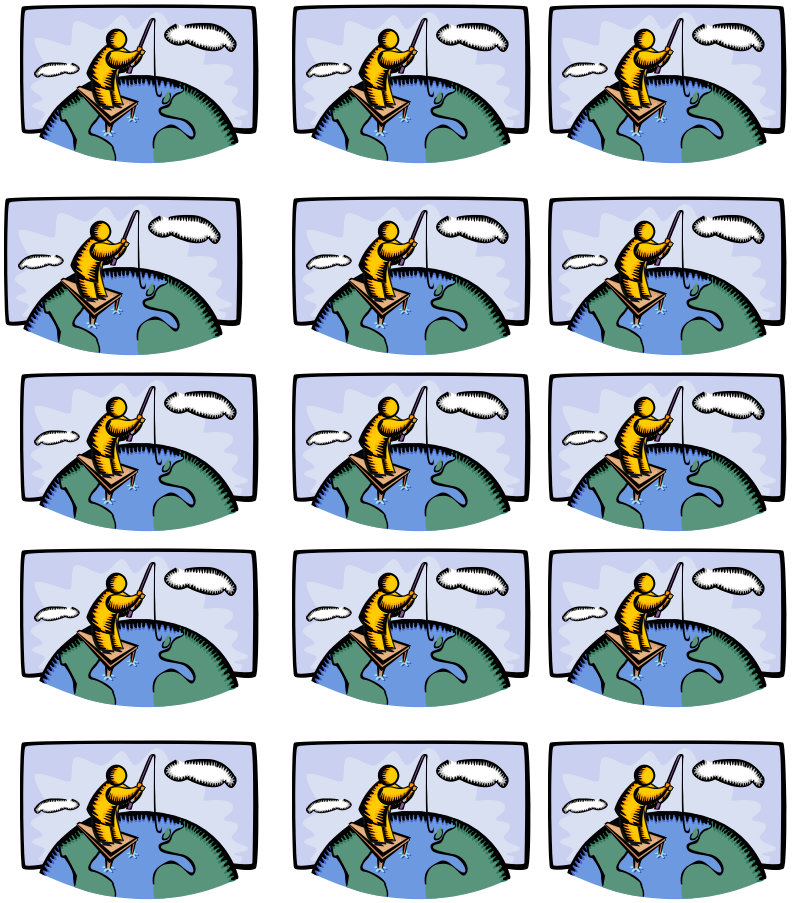
- Parallelism
- Shared Memory Multithreading
- OpenMP

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!

What Is Parallelism?

Parallelism is the use of multiple processing units – any combination of multiple CPU chips, and/or multiple CPU cores within an individual CPU chip, and/or multiple components within an individual CPU core – to solve a single problem (or a collection of related problems), and in particular the use of multiple processing units operating simultaneously on different parts of the problem.

The different parts could be different tasks, or the same task on different pieces of the problem's data.

Common Kinds of Parallelism

- Instruction Level Parallelism
- Shared Memory Multithreading (for example, OpenMP)
- Distributed Multiprocessing (for example, MPI)
- Accelerator Parallelism (for example, CUDA, OpenACC)
- Hybrid Parallelism
  - Distributed + Shared (for example, MPI + OpenMP)
  - Shared + GPU (for example, OpenMP + OpenACC)
  - Distributed + GPU (for example, MPI + CUDA)

Why Parallelism Is Good

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

Parallelism Jargon

- ***Threads*** are execution sequences that share a single memory area ("***address space***")
- ***Processes*** are execution sequences with their own independent, private memory areas

… and thus:

- ***Multithreading***: parallelism via multiple **threads**
- ***Multiprocessing***: parallelism via multiple **processes**

Generally:

- Shared Memory Parallelism is concerned with **threads**, and
- Distributed Parallelism is concerned with **processes**.

Jargon Alert!

In principle:

- "shared memory parallelism" ➔ "multithreading"
- "distributed parallelism" ➔ "multiprocessing"

In practice, sadly, the following terms are often used interchangeably:

- Parallelism
- ***Concurrency*** (includes both parallelism and time slicing)
- Multithreading
- Multiprocessing

Typically, you have to figure out what is meant based on the context.

Amdahl's Law

In 1967, Gene Amdahl came up with an idea so crucial to our understanding of parallelism that they named a **Law** for him:

$$S = \frac{1}{(1 - F_p) + \frac{F_p}{S_p}}$$

where:

- S is the overall speedup achieved by parallelizing a code;
- F_p is the fraction of the code that's parallelizable;
- S_p is the speedup achieved in the parallel part.[1]

Amdahl's Law: Huh?

What does Amdahl's Law tell us?

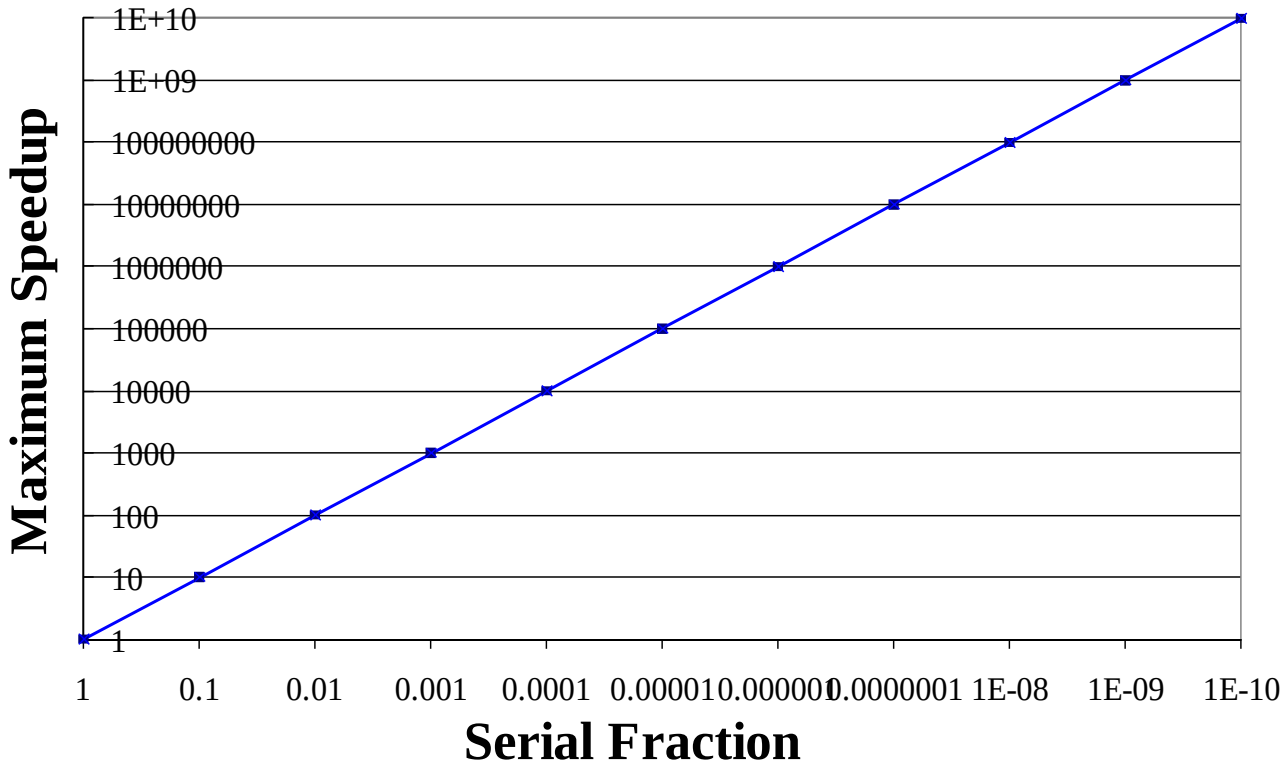
Imagine that you run your code on a zillion processors. The parallel part of the code could speed up by as much as a factor of a zillion.

For sufficiently large values of a zillion, **the parallel part would take zero time**!

But, the **serial** (non-parallel) part would take **the same amount of time as on a single processor**.

So running your code on infinitely many processors would still take at least as much time as it takes to run just the serial part.

Max Speedup by Serial %



Amdahl's Law Example (F90)

```
PROGRAM amdahl_test
  IMPLICIT NONE
  REAL,DIMENSION(a_lot) :: array
  REAL    :: scalar
  INTEGER :: index

  READ *, scalar       !! Serial part
  DO index = 1, a_lot !! Parallel part
    array(index) = scalar * index
  END DO
END PROGRAM amdahl_test
```

If we run this program on infinitely many CPUs, then the total run time will still be at least as much as the time it takes to perform the **READ**.

Amdahl's Law Example (C)

```
int main ()
{
  float array[a_lot];
  float scalar;
  int   index;

  scanf("%f", scalar); /* Serial part */
  /* Parallel part */
  for (index = 0; index < a_lot; index++) {
    array(index) = scalar * index
  }
}
```

If we run this program on infinitely many CPUs, then the total run time will still be at least as much as the time it takes to perform the **`scanf`**.

The Point of Amdahl's Law

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.

Note that this rule will not hold when the ***overhead*** cost of parallelizing exceeds the parallel speedup. More on this presently.

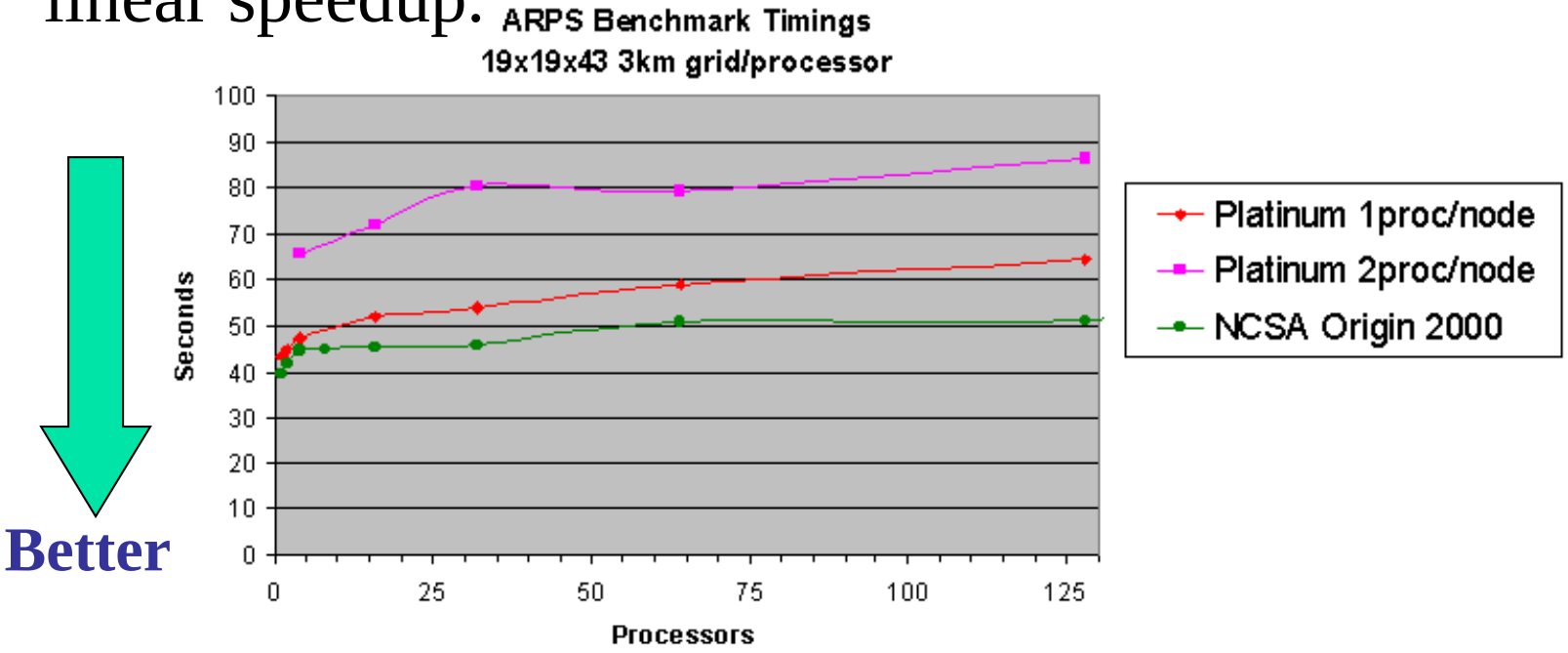
Speedup

The goal in parallelism is ***linear speedup***: getting the speed of the job to increase by a factor equal to the number of processors.

Very few programs actually exhibit linear speedup, but some come close.

Scalability

Scalable means "performs just as well regardless of how big the problem is." A scalable code has near linear speedup.



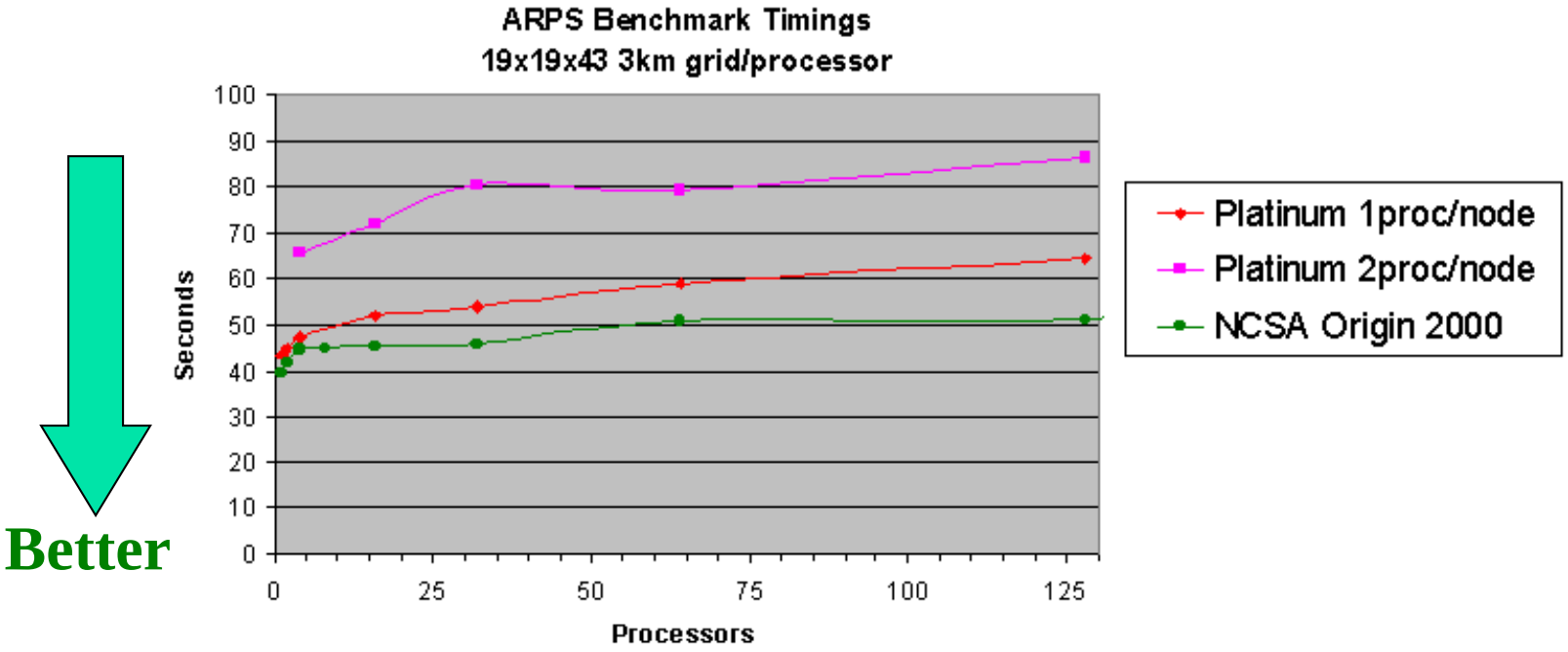
Platinum = NCSA 1024 processor PIII/1GHZ Linux Cluster
Note: NCSA Origin timings are scaled from 19x19x53 domains.

Strong vs Weak Scalability

- ***Strong Scalability***: If you double the number of processors, but you **keep the problem size constant**, then the problem takes **half as long** to complete.
- ***Weak Scalability***: If you double the number of processors, and **double the problem size**, then the problem takes the **same amount of time** to complete.

Scalability

This benchmark shows **weak** scalability.



Platinum = NCSA 1024 processor PIII/1GHZ Linux Cluster
Note: NCSA Origin timings are scaled from 19x19x53 domains.

Granularity

Granularity is the size of the subproblem that each thread or process works on, and in particular the size that it works on between communicating or synchronizing with the others.

Some codes are ***coarse grain*** (a few very large parallel parts) and some are ***fine grain*** (many small parallel parts).

Usually, **coarse grain codes are more scalable** than fine grain codes, because less of the runtime is spent managing the parallelism, so a higher proportion of the runtime is spent getting the work done.

Parallel Overhead

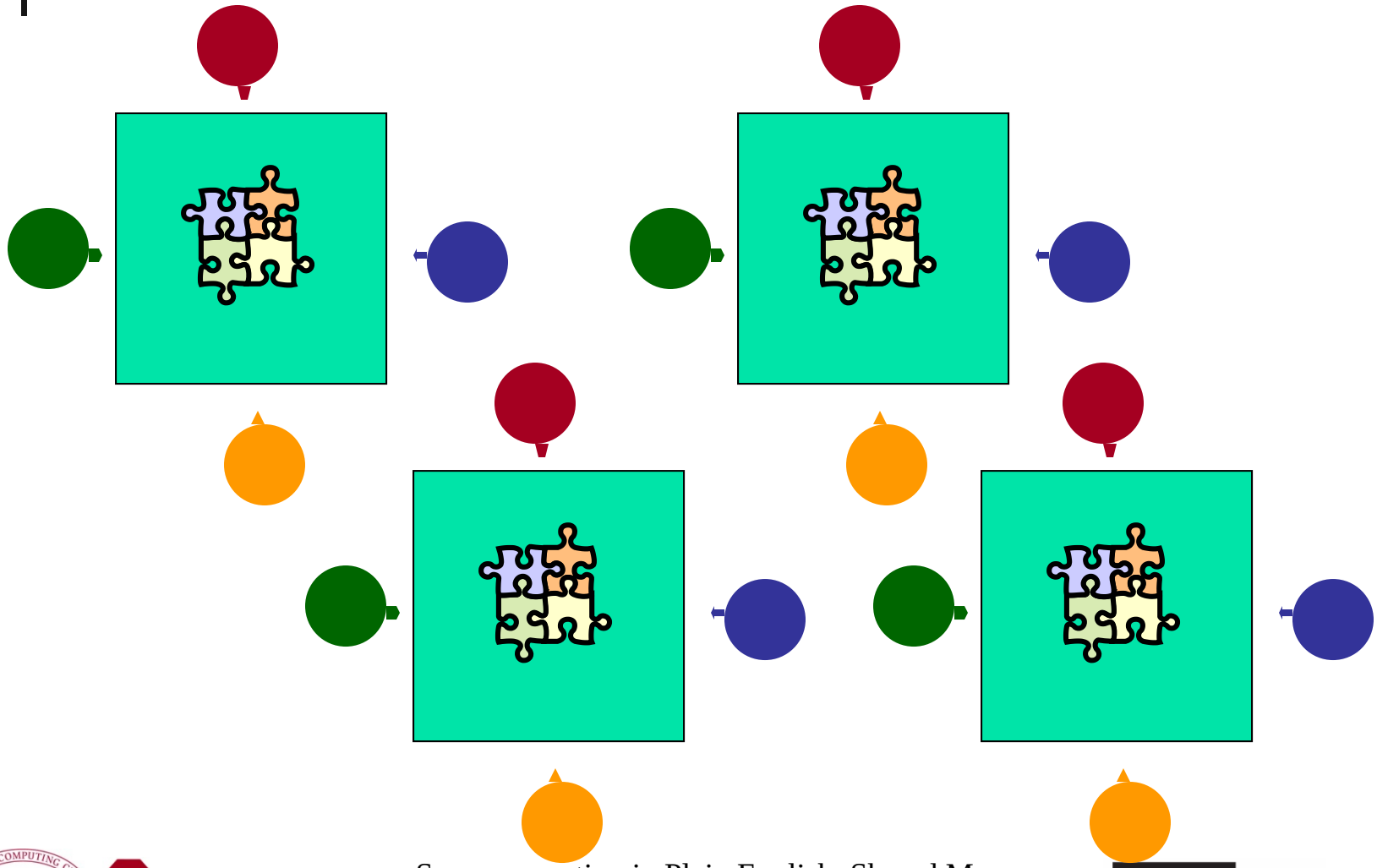
Parallelism isn't free. Behind the scenes, the compiler, the parallel library and the hardware have to do a lot of ***overhead*** work to make parallelism happen.

The overhead typically includes:

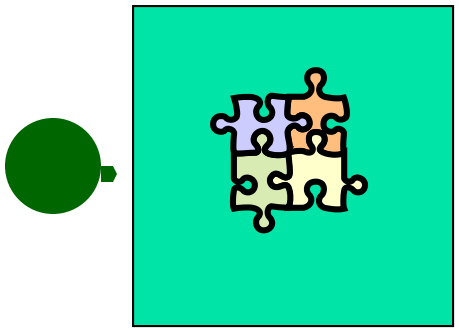
- **Managing** the multiple threads/processes
- **Communication** among threads/processes
- ***Synchronization*** (described later)

Shared Memory Multithreading

The Jigsaw Puzzle Analogy



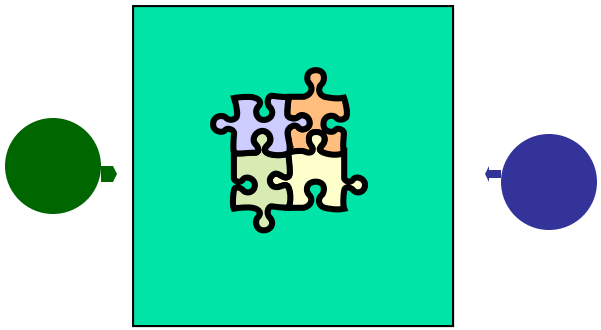
Serial Computing



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

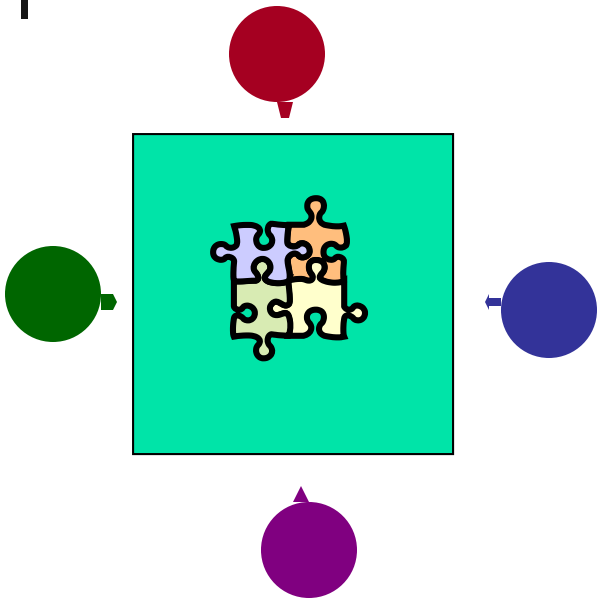
We can imagine that it'll take you a certain amount of time. Let's say that you can put the puzzle together in an hour.

Shared Memory Parallelism



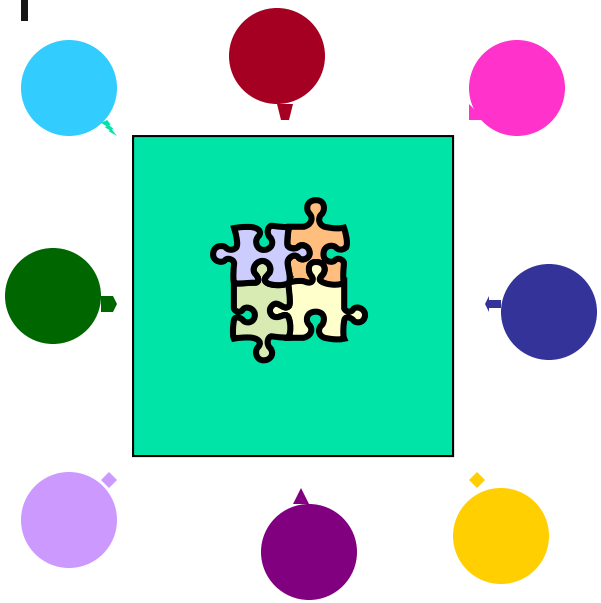
If Scott sits across the table from you, then he can work on his half of the puzzle and you can work on yours. Once in a while, you'll both reach into the pile of pieces at the same time (you'll ***contend*** for the same resource), which will cause a little bit of slowdown. And from time to time you'll have to work together (***communicate***) at the interface between his 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 Paul and Charlie 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



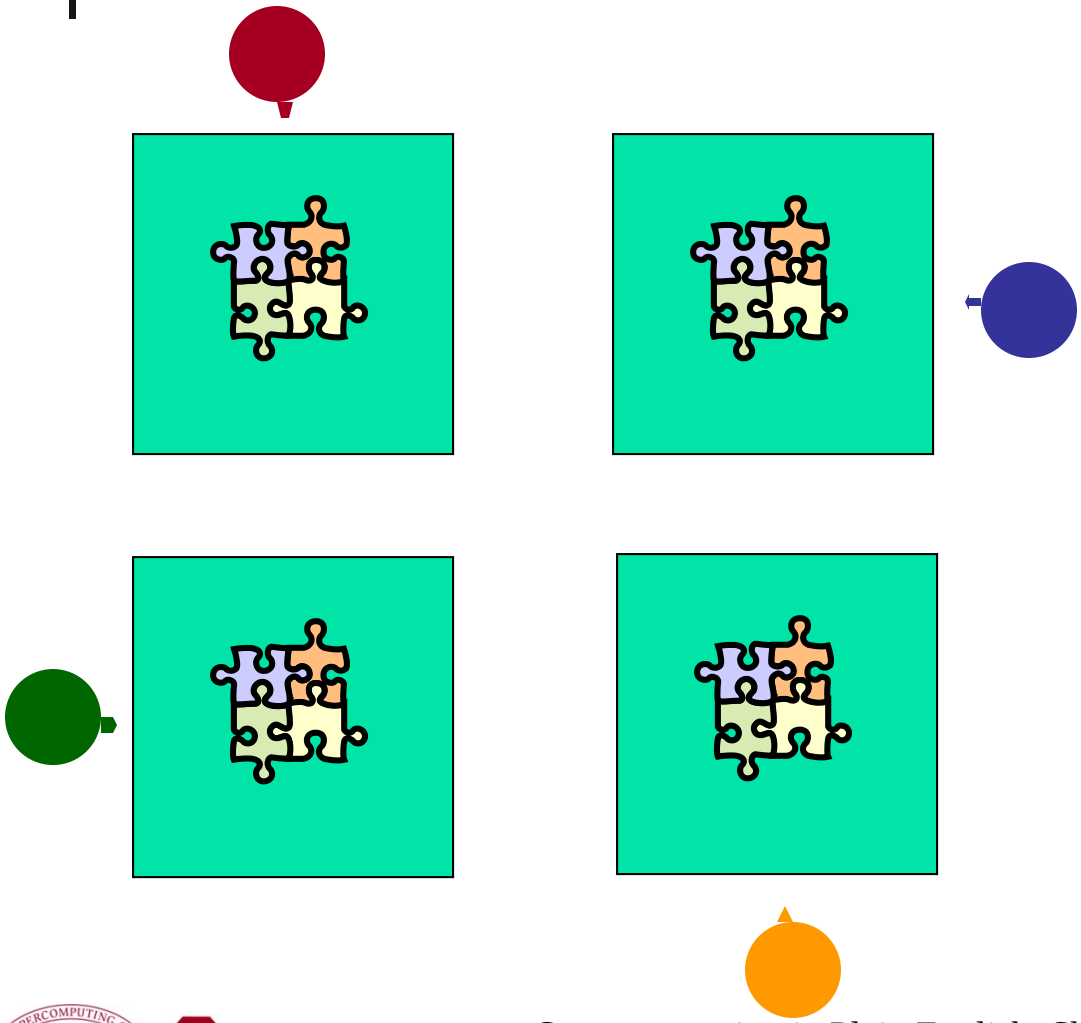
If we now put Dave and Tom and Horst and Brandon on the corners of the table, there's going to be a whole lot of contention for the shared resource, and a lot of communication at the many interfaces. So the speedup y'all get will be much less than we'd like; you'll be lucky to get 5-to-1.

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

Distributed Parallelism

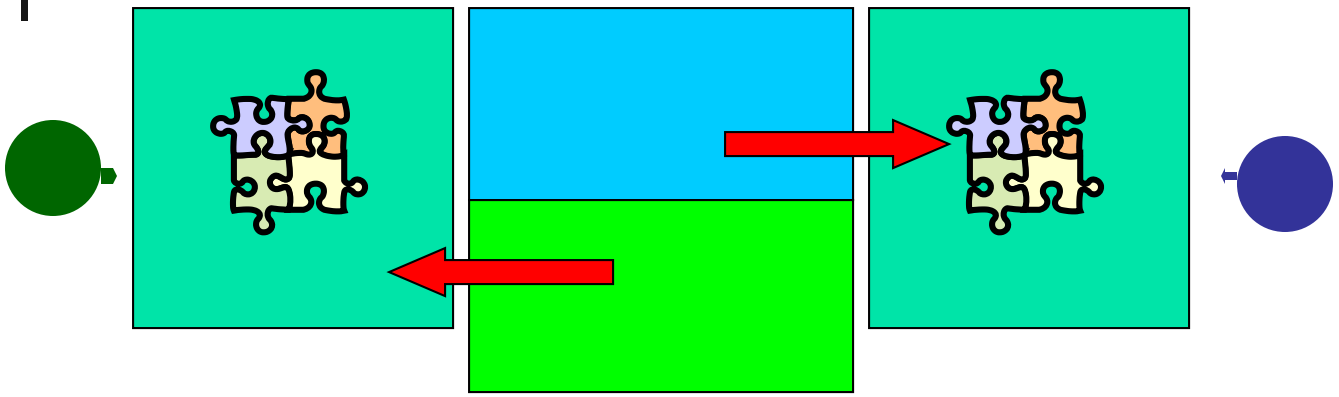
Now let's try something a little different. Let's set up two tables, and let's put you at one of them and Scott at the other. Let's put half of the puzzle pieces on your table and the other half of the pieces on Scott's. Now y'all can work completely independently, without any contention for a shared resource. **BUT**, the cost per communication 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 among the processors. Also, as you add more processors, it may be harder to ***load balance*** the amount of work that each processor gets.

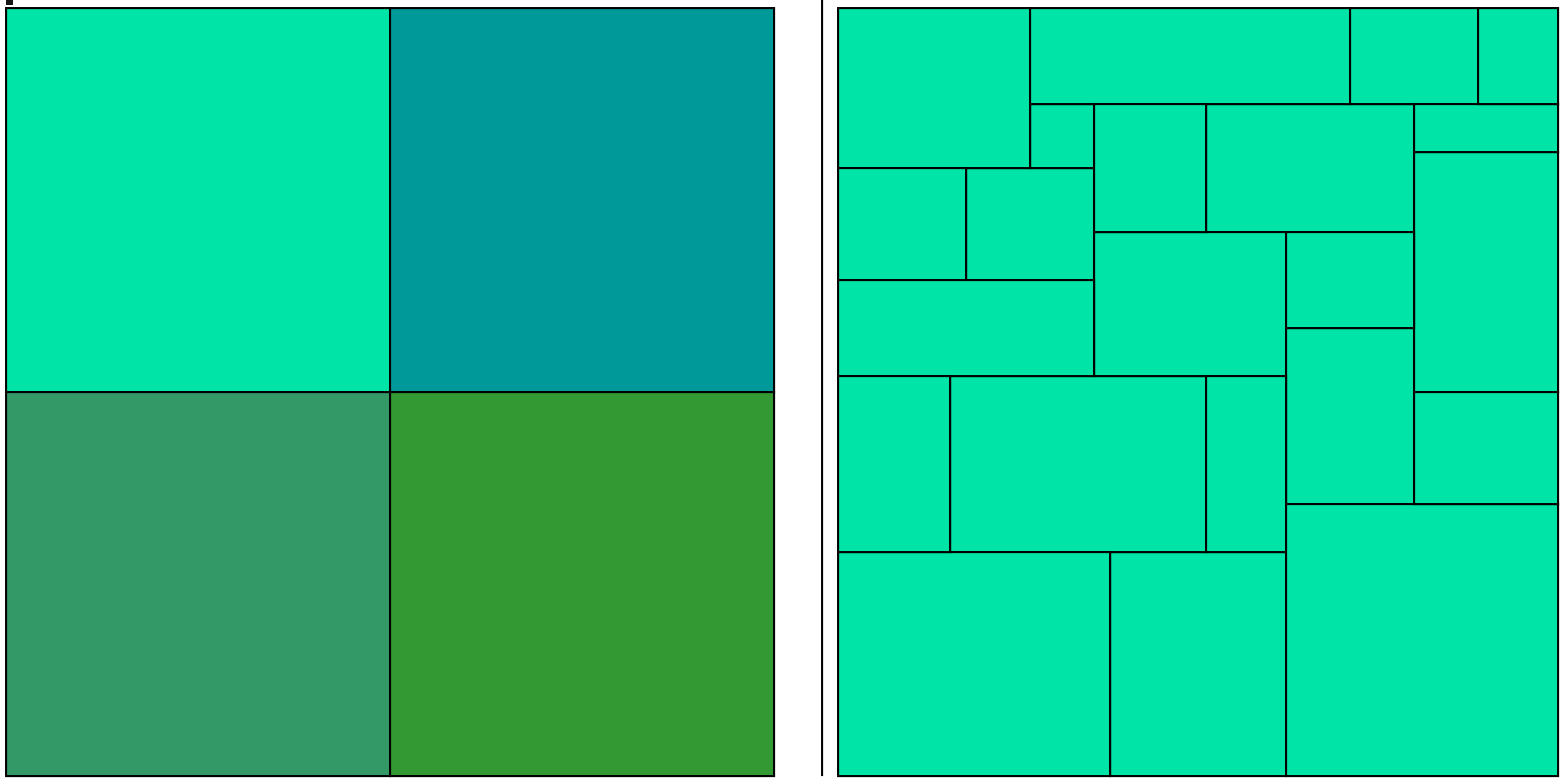
Load Balancing



Load balancing means ensuring that everyone completes their workload at roughly the same time.

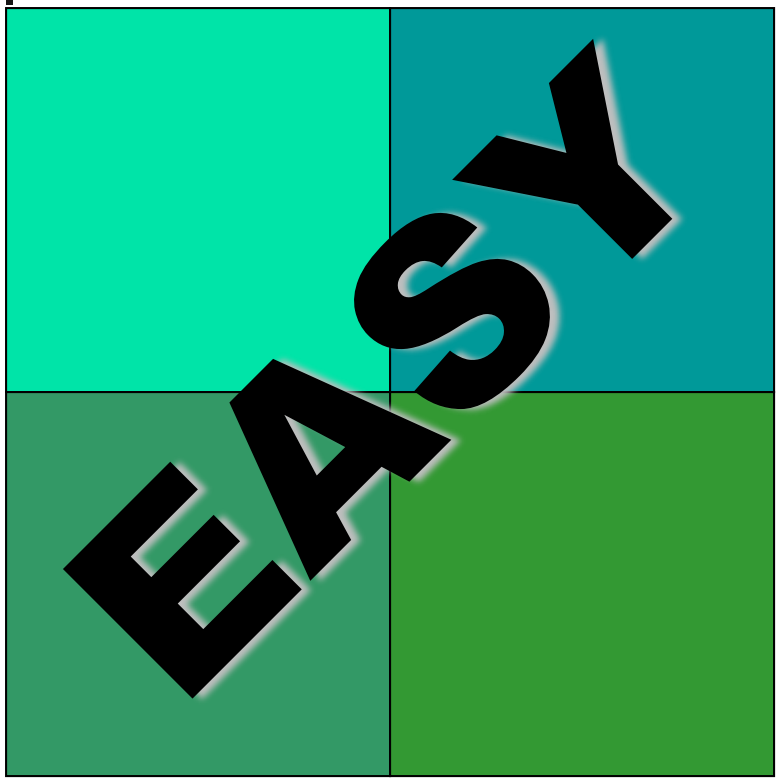
For example, if the jigsaw puzzle is half grass and half sky, then you can do the grass and Scott can do the sky, and then y'all only have to communicate at the horizon – and the amount of work that each of you does on your own is roughly equal. So you'll get pretty good speedup.

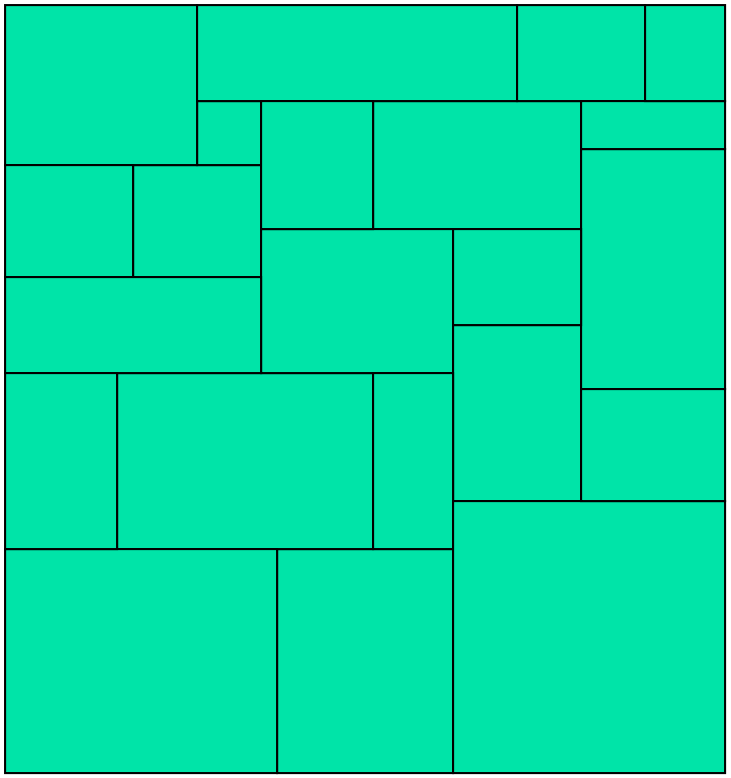
Load Balancing



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.

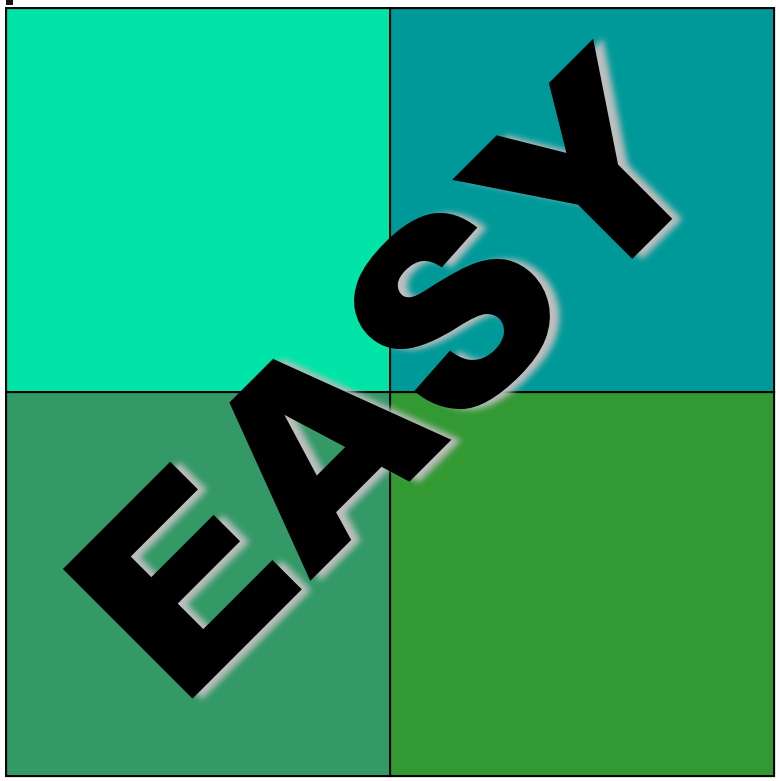
Load Balancing





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.

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.

How Shared Memory Parallelism Behaves

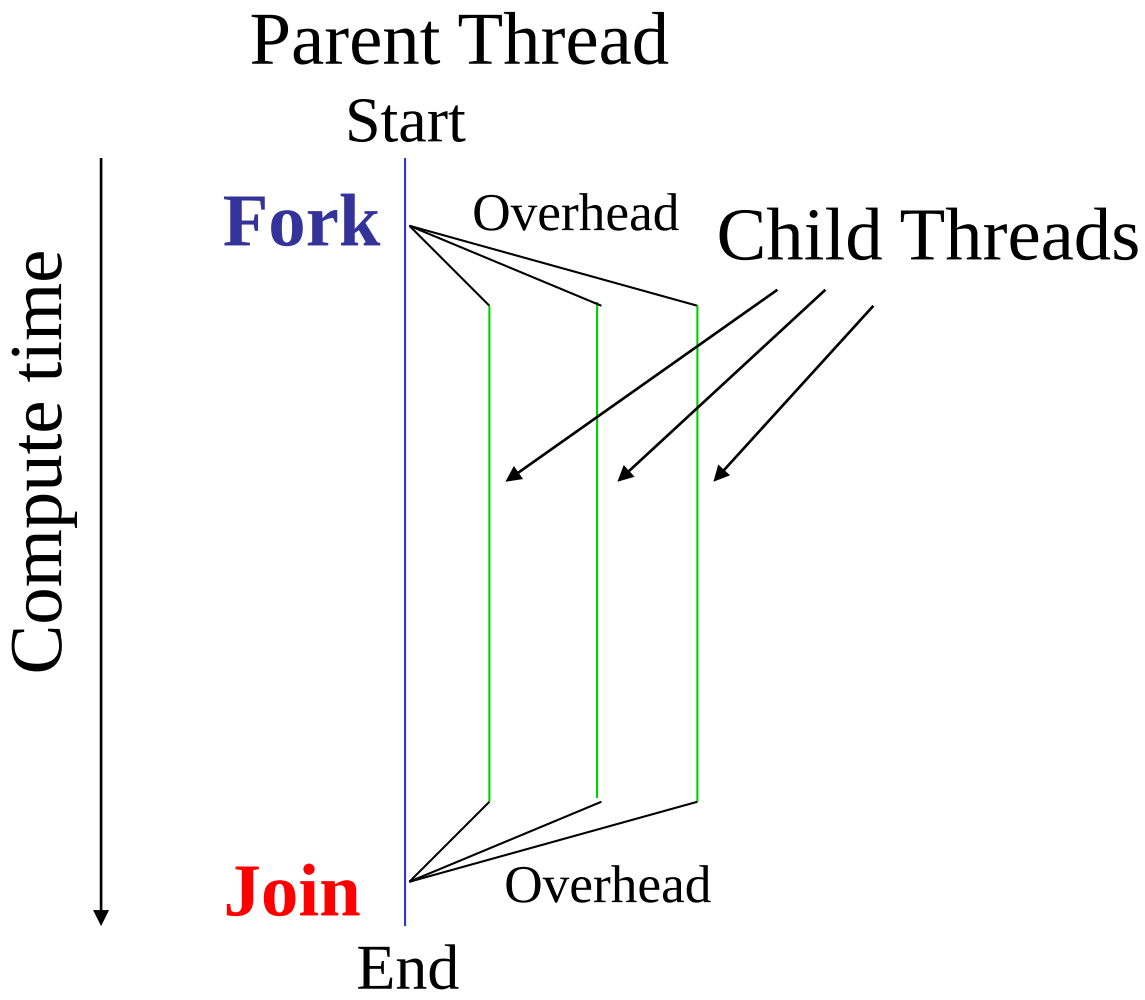
The Fork/Join Model

Many shared memory parallel systems use a programming model called ***Fork***/***Join***. Each program begins executing on just a single thread, called the ***parent***.

Fork: When a parallel region is reached, the **parent** thread ***spawns*** additional ***child*** threads as needed.

Join: When the parallel region ends, **the child threads shut down**, leaving only the parent still running.

The Fork/Join Model (cont'd)



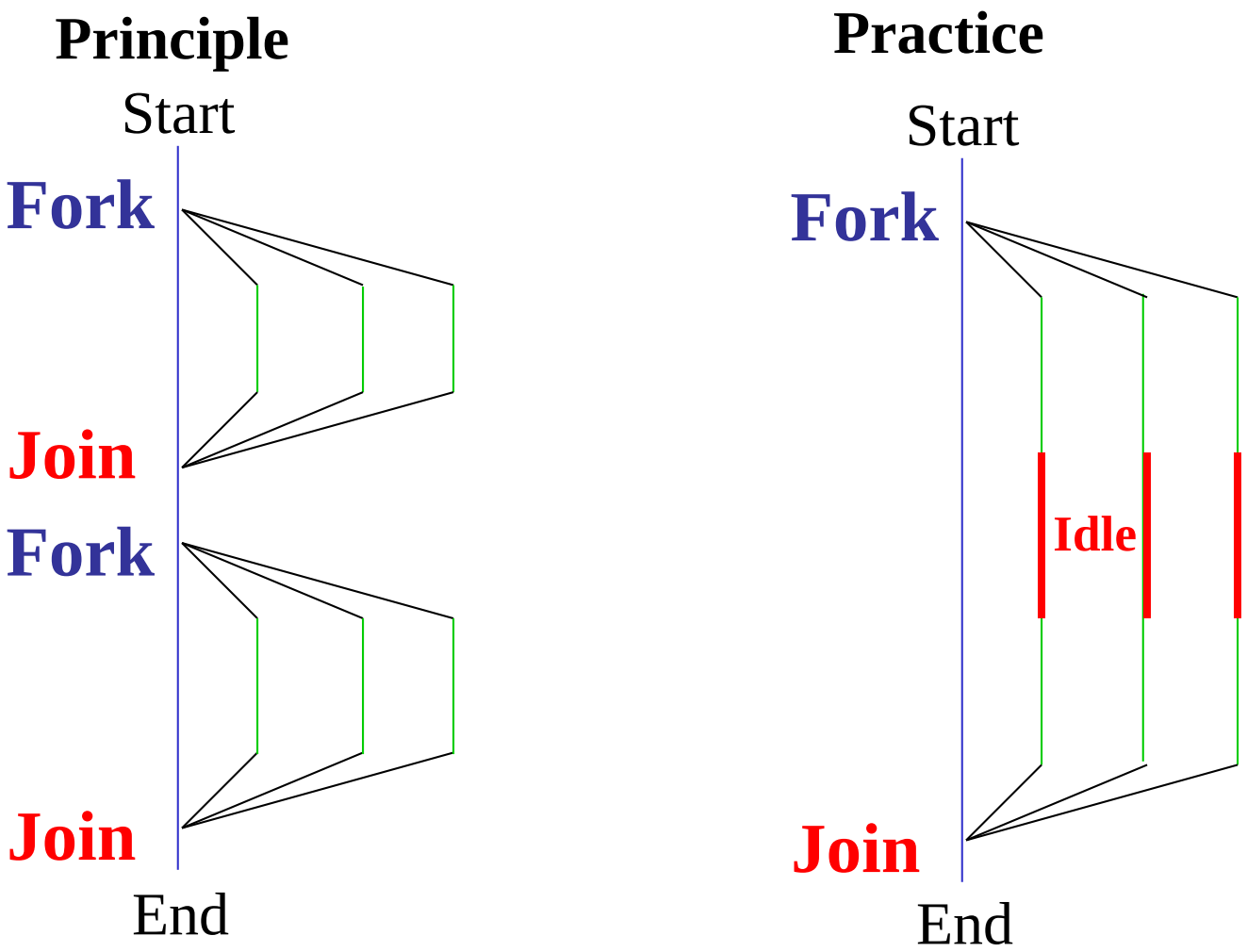
The Fork/Join Model (cont'd)

In principle, as a parallel section completes, the child threads shut down (join the parent), forking off again when the parent reaches another parallel section.

In practice, the child threads often continue to exist but are ***idle***.

Why?

Principle vs. Practice



Why Idle?

- On some shared memory multithreading computers, the **overhead cost** of forking and joining is **high** compared to the cost of computing, so rather than waste time on overhead, the **children sit idle** until the next parallel section.
- On some computers, joining threads releases a program's control over the child processors, so they may not be available for more parallel work later in the run. ***Gang scheduling*** is preferable, because then all of the processors are guaranteed to be available for the whole run.

Why Idle? (cont'd)

These issues are becoming increasingly acute as the number of CPU cores per socket (CPU chip) has gone up:

Current x86 CPUs have:

- Intel Xeon Skylake: up to 28 cores per socket;
- AMD EPYC: up to 32 cores per socket.

So a dual-socket server can have as many as 64 CPU cores.

But the performance of a shared memory parallel code often doesn't scale well to that many cores.

So a single compute node (server PC) might easily have multiple different jobs – potentially from multiple different users – running on it at the same time.

Simultaneous Multithreading

Simultaneous Multithreading #1

- Most contemporary CPU families allow ***Simultaneous Multithreading*** (SMT): a single CPU core can have multiple threads running on it at the same time.
- This is typically accomplished by having multiple, independent ***register files*** (or ***register banks***) on each core, each of which corresponds to a thread of its own.
- But, the multiple register files share the same caches, the same functional units, and the single connection to RAM.
- Threads within a core do ***concurrency*** , but **NOT** parallelism, via ***context switching***, which means ***time slicing***:
  do the work of thread #0 for a little while, then
  switch to thread #1 for a little while, then back to thread #0, etc.

Simultaneous Multithreading #2

1. SMT can be a big win if any given thread spends a lot of its time ***stalled*** while waiting for data from RAM,
   because thread #1 can be doing its calculations
   while thread #0 is waiting for its data from RAM, and vice versa.
   - But this works best when the stalls don't overlap in time.
2. SMT can be a big loss if any given thread gets a lot of value from data in cache, because the data in cache for thread #0 will constantly get clobbered by the data needed for thread #1, and vice versa, causing ***cache thrashing*** (common in HPC codes).
3. SMT can also be a big loss when the multiple threads stall at the same time, which is common in memory-intensive HPC codes.

So SMT is often **off on HPC systems**, because 2 and 3 often apply.

Simultaneous Multithreading #3

- Intel x86: 2 SMT threads per core on some CPU types (some don't have SMT) – known as Hyper-Threading.
- AMD x86: 2 SMT threads per core on some CPU type (some don't have SMT).
- POWER9: 4 or 8 SMT threads per core.
- SPARC: up to 8 SMT threads per core.
- Intel Itanium: 2 SMT threads per core.
- ARM: Currently doesn't support SMT (as far as I can find).

(This information is from Wikipedia.)

Standards and Nonstandards

Standards and Nonstandards

In computing, there are standards and nonstandards.

Standards are established by independent organizations and made public, so that anyone can produce a standard-compliant implementation.

Example standards organizations include:

- International Organization for Standardization (ISO)
  - "'ISO' [is] derived from the Greek *isos*, meaning 'equal'." [2]
- American National Standards Institute (ANSI)
- Ecma International

Nonstandards are produced by a single organization or consortium, with no requirement for external input and no recognized standard.

Standards and Nonstandards

In practice, there are:

- **standard standards**, which both are common and have been accepted as official standards – for example: C, TCP/IP, HTML;
- **nonstandard standards**, which aren't common but have been accepted as official standards – for example: Myrinet;
- **standard nonstandards**, which are common but haven't been accepted as official standard – for example: PDF, Windows;
- **nonstandard nonstandards**, which aren't common and haven't been accepted as official standards – for example: WordStar.

OpenMP

Most of this discussion is from [3], with a little bit from [4].

What Is OpenMP?

OpenMP is a **standard** for expressing shared memory parallelism.

OpenMP consists of ***compiler directives***, **functions** and **environment variables**.

When you compile a program that has OpenMP in it, then:

- if your compiler knows OpenMP, then you get an executable that can run in parallel;
- otherwise, the compiler ignores the OpenMP stuff and you get a purely serial executable.

OpenMP can be used in Fortran, C and C++, but only if your preferred compiler explicitly supports it.

For Java, there have been several attempts, but it doesn't seem like any are being actively developed.

For Python, there's PyMP, but it's not part of OpenMP. https://github.com/classner/pymp

Compiler Directives

A ***compiler directive*** is a line of source code that gives the compiler special information about the statement or block of code that immediately follows.

C++ and C programmers already know about compiler directives:

```
#include "MyClass.h"
```

Many Fortran programmers already have seen at least one compiler directive:

```
INCLUDE 'mycommon.inc'
```

OR

```
INCLUDE "mycommon.inc"
```

OpenMP Compiler Directives

OpenMP compiler directives in Fortran look like this:

`!$OMP` …*stuff*…

In C++ and C, OpenMP directives look like:

`#pragma omp` …*stuff*…

Both directive forms mean "the rest of this line contains OpenMP information."

Aside: "***pragma***" (πράγμα) is the Greek word for "thing." Go figure.

[translate.google.com]

Example OpenMP Directives

Fortran	C++/C
`!$OMP PARALLEL DO`	`#pragma omp parallel for`
`!$OMP CRITICAL`	`#pragma omp critical`
`!$OMP MASTER`	`#pragma omp master`
`!$OMP BARRIER`	`#pragma omp barrier`
`!$OMP SINGLE`	`#pragma omp single`
`!$OMP ATOMIC`	`#pragma omp atomic`
`!$OMP SECTION`	`#pragma omp section`
`!$OMP FLUSH`	`#pragma omp flush`
`!$OMP ORDERED`	`#pragma omp ordered`

Note that we won't cover all of these.

A First OpenMP Program (F90)

```
PROGRAM hello_world
  IMPLICIT NONE
  INTEGER :: number_of_threads, this_thread, iteration
  INTEGER,EXTERNAL :: omp_get_max_threads,  &
 &                    omp_get_thread_num

  number_of_threads = omp_get_max_threads()
  WRITE (0,"(I2,A)") number_of_threads, " threads"
!$OMP PARALLEL DO
  DO iteration = 0, number_of_threads - 1
    this_thread = omp_get_thread_num()
    WRITE (0,"(A,I2,A,I2,A) ")"Iteration ", &
 &    iteration, ", thread ", this_thread, &
 &    ": Hello, world!"
  END DO
END PROGRAM hello_world
```

A First OpenMP Program (C)

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main ()
{
  int number_of_threads, this_thread, iteration;

  number_of_threads = omp_get_max_threads();
  fprintf(stderr, "%2d threads\n", number_of_threads);
# pragma omp parallel for
  for (iteration = 0;
       iteration < number_of_threads; iteration++) {
    this_thread = omp_get_thread_num();
    fprintf(stderr, "Iteration %2d, thread %2d: Hello, world!\n",
         iteration, this_thread);
  }
}
```

Running hello_world

```
% setenv  OMP_NUM_THREADS  4
% hello_world
 4 threads
Iteration  0, thread  0: Hello, world!
Iteration  1, thread  1: Hello, world!
Iteration  3, thread  3: Hello, world!
Iteration  2, thread  2: Hello, world!
% hello_world
 4 threads
Iteration  2, thread  2: Hello, world!
Iteration  1, thread  1: Hello, world!
Iteration  0, thread  0: Hello, world!
Iteration  3, thread  3: Hello, world!
% hello_world
 4 threads
Iteration  1, thread  1: Hello, world!
Iteration  2, thread  2: Hello, world!
Iteration  0, thread  0: Hello, world!
Iteration  3, thread  3: Hello, world!
```

OpenMP Issues Observed

From the **`hello_world`** program, we learn that:

- At some point before running an OpenMP program, you must set an environment variable

  `OMP_NUM_THREADS`

  that represents the number of threads to use.
- The order in which the threads execute is **nondeterministic**.

The PARALLEL DO Directive (F90)

The **PARALLEL DO** directive tells the compiler that the **DO** loop immediately after the directive should be executed in parallel; for example:

```
!$OMP PARALLEL DO
  DO index = 1, length
    array(index) = index * index
  END DO
```

The iterations of the DO loop will be computed in parallel (note that they are independent of one another).

The `parallel for` Directive (C)

The **`parallel for`** directive tells the compiler that the **`for`** loop immediately after the directive should be executed in parallel; for example:

```
# pragma omp parallel for
  for (index = 0; index < length; index++) {
    array[index] = index * index;
  }
```

The iterations of the for loop will be computed in parallel (note that they are independent of one another).

A Change to `hello_world`

Suppose we do 3 loop iterations per thread:

```
DO iteration = 0, number_of_threads * 3 - 1
```

```
% hello_world
 4 threads
Iteration  9, thread  3: Hello, world!
Iteration  0, thread  0: Hello, world!
Iteration 10, thread  3: Hello, world!
Iteration 11, thread  3: Hello, world!
Iteration  1, thread  0: Hello, world!
Iteration  2, thread  0: Hello, world!
Iteration  3, thread  1: Hello, world!
Iteration  6, thread  2: Hello, world!
Iteration  7, thread  2: Hello, world!
Iteration  8, thread  2: Hello, world!
Iteration  4, thread  1: Hello, world!
Iteration  5, thread  1: Hello, world!
```

Notice that the iterations are split into contiguous ***chunks***, and each thread gets one chunk of iterations.

Chunks

By default, OpenMP splits the iterations of a loop into chunks of equal (or roughly equal) size, assigns each chunk to a thread, and lets each thread loop through its subset of the iterations.

So, for example, if you have 4 threads and 12 iterations, then each thread gets three iterations:

- Thread 0: iterations 0, 1, 2
- Thread 1: iterations 3, 4, 5
- Thread 2: iterations 6, 7, 8
- Thread 3: iterations 9, 10, 11

Notice that each thread performs its own chunk in deterministic order, but that the overall order is nondeterministic.

Private and Shared Data

Private data are data that are owned by, and only visible to, a single individual thread.

Shared data are data that are owned by and visible to all threads.

(Note: In distributed parallelism, all data are private, as we'll see next time.)

Should All Data Be Shared?

In our example program, we saw this:

```
!$OMP PARALLEL DO PRIVATE(iteration, this_thread)  &
!$OMP             SHARED(number_of_threads)
```

or this:

```
#pragma parallel for private(iteration, this_thread) \
                     shared(number_of_threads)
```

What do **PRIVATE** and **SHARED** mean?

We said that OpenMP uses shared memory parallelism.
So **PRIVATE** and **SHARED** refer to memory.

Would it make sense for all data within a parallel loop to be shared?

A Private Variable (F90)

Consider this loop:

```
!$OMP PARALLEL DO …
  DO iteration = 0, number_of_threads - 1
    this_thread = omp_get_thread_num()
    WRITE (0,"(A,I2,A,I2,A) ") "Iteration ", iteration, &
 &    ", thread ", this_thread, ": Hello, world!"
  END DO
```

Notice that, if the iterations of the loop are executed in parallel, then the loop index variable named **iteration** will be wrong for all but one of the threads.

Each thread should get its own copy of the variable named **iteration**.

A Private Variable (C)

Consider this loop:

```
#pragma omp parallel for …
  for (iteration = 0;
       iteration < number_of_threads; iteration++) {
    this_thread = omp_get_thread_num();
    printf("Iteration %d, thread %d: Hello, world!\n",
        iteration, this_thread);
}
```

Notice that, if the iterations of the loop are executed in parallel, then the loop index variable named **`iteration`** will be wrong for all but one of the threads.

Each thread should get its own copy of the variable named **`iteration`**.

Another Private Variable (F90)

```
!$OMP PARALLEL DO …
  DO iteration = 0, number_of_threads - 1
    this_thread = omp_get_thread_num()
    WRITE (0,"(A,I2,A,I2,A)") "Iteration ", iteration, &
 &    ", thread ", this_thread, ": Hello, world!"
  END DO
```

Notice that, if the iterations of the loop are executed in parallel, then **this_thread** will be wrong for all but one of the threads.

Each thread should get its own copy of the variable named **this_thread**.

Another Private Variable (C)

```
#pragma omp parallel for …
  for (iteration = 0;
       iteration < number_of_threads; iteration++) {
    this_thread = omp_get_thread_num();
    printf("Iteration %d, thread %d: Hello, world!\n",
        iteration, this_thread);
}
```

Notice that, if the iterations of the loop are executed in parallel, then **`this_thread`** will be wrong for all but one of the threads.

Each thread should get its own copy of the variable named **`this_thread`**.

A Shared Variable (F90)

```
!$OMP PARALLEL DO …
  DO iteration = 0, number_of_threads - 1
    this_thread = omp_get_thread_num()
    WRITE (0,"(A,I2,A,I2,A)"“) "Iteration ", iteration, &
 &    ", thread ", this_thread, ": Hello, world!"
  END DO
```

Notice that, regardless of whether the iterations of the loop are executed serially or in parallel, **number_of_threads** will be correct for all of the threads.

All threads should share a single instance of **number_of_threads**.

A Shared Variable (C)

```
#pragma omp parallel for …
  for (iteration = 0;
       iteration < number_of_threads; iteration++) {
    this_thread = omp_get_thread_num();
    printf("Iteration %d, thread %d: Hello, world!\n",
        iteration, thread);
}
```

Notice that, regardless of whether the iterations of the loop are executed serially or in parallel, **`number_of_threads`** will be correct for all of the threads.

All threads should share a single instance of **`number_of_threads`**.

SHARED & PRIVATE Clauses

The **PARALLEL DO** directive allows extra ***clauses*** to be appended that tell the compiler which variables are shared and which are private:

```
!$OMP PARALLEL DO PRIVATE(iteration,this_thread) &
!$OMP             SHARED (number_of_threads)
```

or:

```
#pragma parallel for private(iteration, this_thread) \
                     shared(number_of_threads)
```

This tells that compiler that **iteration** and **this_thread** are private but that **number_of_threads** is shared.

(Note the syntax for continuing a directive in Fortran90 and C.)

DEFAULT Clause

If your loop has lots of variables, it may be cumbersome to put all of them into **SHARED** and **PRIVATE** clauses.

So, OpenMP allows you to declare one kind of data to be the default, and then you only need to explicitly declare variables of the other kind:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) &
!$OMP             SHARED(number_of_threads)
```

The default **DEFAULT** (so to speak) is **SHARED**, except for the loop index variable, which by default is **PRIVATE**.

Different Workloads (F90)

What happens if the threads have different amounts of work to do?

```
!$OMP PARALLEL DO
  DO index = 1, length
    x(index) = index / 3.0
    IF (x(index) < 0) THEN
      y(index) = LOG(x(index))
    ELSE
      y(index) = 1.0 - x(index)
    END IF
  END DO
```

The threads that finish early have to wait.

Different Workloads (C)

What happens if the threads have different amounts of work to do?

```
# pragma parallel for
  for (index = 0; index < length; index++) {
    x[index] = index / 3.0;
    if (x[index] < 0) {
      y[index] = log(x[index]);
    }
    else {
      y[index] = 1.0 – x[index];
    }
  }
```

The threads that finish early have to wait.

Recap: Chunks

By default, OpenMP splits the iterations of a loop into chunks of equal (or roughly equal) size, assigns each chunk to a thread, and lets each thread loop through its subset of the iterations.

So, for example, if you have 4 threads and 12 iterations, then each thread gets three iterations:

- Thread 0: iterations 0, 1, 2
- Thread 1: iterations 3, 4, 5
- Thread 2: iterations 6, 7, 8
- Thread 3: iterations 9, 10, 11

Notice that each thread performs its own chunk in deterministic order, but that the overall order is nondeterministic.

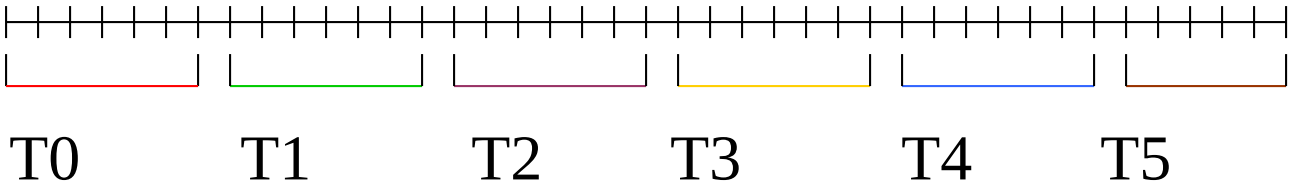
Scheduling Strategies

OpenMP supports three scheduling strategies:

- ***Static***: The default, as described in the previous slides – good for iterations that are inherently load balanced.
- ***Dynamic***: Each thread gets a chunk of a few iterations, and when it finishes that chunk it goes back for more, and so on until all of the iterations are done – good when iterations aren't load balanced at all.
- ***Guided***: Initially, each thread gets a pretty big chunk, but over time, each thread gets smaller and smaller chunks – a compromise between static and dynamic.

Static Scheduling

For N_i iterations and N_t threads, each thread gets one chunk of N_i/N_t loop iterations:



- Thread #0: iterations 0 through N_i/N_t-1
- Thread #1: iterations N_i/N_t through $2N_i/N_t$-1
- Thread #2: iterations $2N_i/N_t$ through $3N_i/N_t$-1

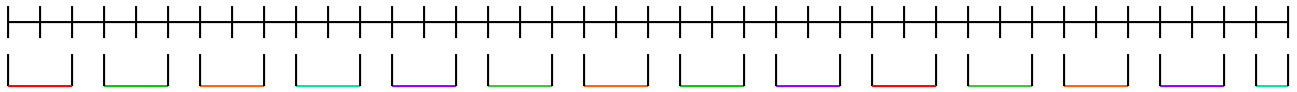
…

- Thread #N_t-1: iterations $(N_t\text{-}1)N_i/N_t$ through N_i-1

So the relationship between iterations and threads is deterministic – no decisions are made while the loop is running.

Dynamic Scheduling

For N_i iterations and N_t threads, each thread gets a fixed-size chunk of k loop iterations, where $k << N_i$:



T0 T1 T2 T3 T4 T5 T2 T3 T4 T0 T1 T5 T3 T2

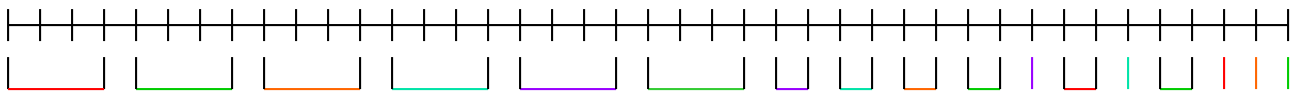
When a particular thread finishes its chunk of iterations, it gets assigned a new chunk.

So, the relationship between loop iterations and threads is nondeterministic.

- Advantage: very flexible.
- Disadvantage: high overhead – lots of decision making about which thread gets each chunk (the number of decisions made while the loop is running is proportional to N_i).

Guided Scheduling

For N_i iterations and N_t threads, initially each thread gets a fixed-size chunk of $k < N_i/N_t$ loop iterations:



T0 T1 T2 T3 T4 T5 2 3 4 1 0 2 5 4 2 3 1

After each thread finishes its chunk of k iterations, it gets a chunk of $k/2$ iterations, then $k/4$, etc. Chunks are assigned dynamically, as threads finish their previous chunks.

- Advantage over static: can handle imbalanced load.
- Advantage over dynamic: fewer decisions, so less overhead – the number of decisions made while the loop is running is proportional to $\log_2 (N_i)$.

How to Know Which Schedule?

Test all three using a typical case as a ***benchmark***.

Whichever wins is probably the one you want to use most of the time on that particular platform.

This may vary depending on:

- problem size,
- compiler versions,
- what's running on the machine,
- what day of the week it is, etc.

So you may want to re-benchmark the three schedules from time to time.

SCHEDULE Clause

The **PARALLEL DO** directive allows a **SCHEDULE** clause to be appended that tell the compiler which variables are shared and which are private:

```
!$OMP PARALLEL DO … SCHEDULE(STATIC)
```

This tells that compiler that the schedule will be static.

Likewise, the schedule could be **GUIDED** or **DYNAMIC**.

However, the very best schedule to put in the **SCHEDULE** clause is **RUNTIME**.

You can then set the environment variable **OMP_SCHEDULE** to **STATIC** or **GUIDED** or **DYNAMIC** at runtime – great for benchmarking!

Synchronization

Jargon: Waiting for other threads to finish a parallel loop (or other parallel section) before going on to the work after that parallel section is called ***synchronization***.

Synchronization is **BAD**, because when a thread is waiting for the others to finish, it isn't getting any work done, so it isn't contributing to speedup.

So why would anyone ever synchronize?

Why Synchronize? (F90)

Synchronizing is necessary when the code that's after a parallel section needs all threads to have their final answers.

```
!$OMP PARALLEL DO
  DO index = 1, length
    x(index) = index / 1024.0
    IF ((index / 1000) < 1) THEN
      y(index) = LOG(x(index))
    ELSE
      y(index) = x(index) + 2
    END IF
  END DO
! Need to synchronize here!
  DO index = 1, length
    z(index) = y(index) + y(length – index + 1)
  END DO
```

Why Synchronize? (C)

Synchronizing is necessary when the code that's after a parallel section needs all threads to have their final answers.

```
#pragma omp parallel for
  for (index = 0; index < length; index++) {
    x[index] = index / 1024.0;
    if ((index / 1000) < 1) {
      y[index] = log(x[index]);
    }
    else {
      y[index] = x[index] + 2;
    }
  }
/* Need to synchronize here! */
  for (index = 0; index < length; index++) {
    z[index] = y[index] + y[length – index + 1];
  }
```

Barriers

A ***barrier*** is a place where synchronization is forced to occur; that is, where threads that finish sooner have to wait for threads that finish later.

The **PARALLEL DO** directive automatically puts an invisible, implied barrier at the end of its **DO** loop:

```
!$OMP PARALLEL DO
  DO index = 1, length
     … parallel stuff …
  END DO
! Implied barrier
  … serial stuff …
```

OpenMP also has an explicit **BARRIER** directive, but most people don't need it.

Critical Sections

A ***critical section*** is a piece of code that any thread can execute, but that only one thread can execute at a time.

```
!$OMP PARALLEL DO
  DO index = 1, length
     … parallel stuff …
!$OMP CRITICAL(summing)
     sum = sum + x(index) * y(index)
!$OMP END CRITICAL(summing)
     … more parallel stuff …
  END DO
```

What's the point?

Why Have Critical Sections?

If only one thread at a time can execute a critical section, that slows the code down, because the other threads may be waiting to enter the critical section.

But, for certain statements, if you don't ensure ***mutual exclusion***, then you can get nondeterministic results.

If No Critical Section

```
!$OMP CRITICAL(summing)
     sum = sum + x(index) * y(index)
!$OMP END CRITICAL(summing)
```

Suppose that, for thread #0, **`index`** is 27, and for thread #1, **`index`** is 92.

If the two threads execute the above statement at the same time, **`sum`** could be

- the value after adding **`x(27) * y(27)`**, or
- the value after adding **`x(92) * y(92)`**, or
- garbage!

This is called a ***race condition***: the result depends on which thread wins the race.

Pen Game #1: Take the Pen

We need two volunteers for this game.

1. I'll hold a pen in my hand.
2. You win by taking the pen from my hand.
3. One, two, three, go!

Can we predict the outcome? Therefore, can we guarantee that we get the correct outcome?

Pen Game #2: Look at the Pen

We need two volunteers for this game.

1. I'll hold a pen in my hand.
2. You win by looking at the pen.
3. One, two, three, go!

Can we predict the outcome? Therefore, can we guarantee that we get the correct outcome?

Race Conditions

A ***race condition*** is a situation in which multiple processes can change the value of a variable at the same time.

As in Pen Game #1 (Take the Pen), a race condition can lead to unpredictable results.

So, race conditions are **BAD**.

Reductions

A ***reduction*** converts an array to a scalar (or, more generally, many data into fewer data):
sum, product, minimum value, maximum value, location of minimum value, location of maximum value, Boolean AND, Boolean OR, number of occurrences, etc.

Reductions are so common, and so important, that OpenMP has a specific construct to handle them:
the `REDUCTION` clause in a `PARALLEL DO` directive.

Reduction Clause

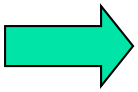
```
  total_mass = 0
!$OMP PARALLEL DO REDUCTION(+:total_mass)
  DO index = 1, length
    total_mass = total_mass + mass(index)
  END DO !! index
```

This is equivalent to:

```
  DO thread = 0, number_of_threads - 1
    thread_mass(thread) = 0
  END DO !! thread
$OMP PARALLEL DO
  DO index = 1, length
    thread = omp_get_thread_num()
    thread_mass(thread) = thread_mass(thread) + mass(index)
  END DO !! index
  total_mass = 0
  DO thread = 0, number_of_threads - 1
    total_mass = total_mass + thread_mass(thread)
  END DO !! thread
```

Parallelizing a Serial Code #1

```
PROGRAM big_science
  … declarations …

    DO …
   … parallelizable work …
    END DO
  … serial work …

    DO …
   … more parallelizable work …
    END DO
  … serial work …
  … etc …
END PROGRAM big_science
```

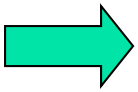


```
PROGRAM big_science
  … declarations …
!$OMP PARALLEL DO …
    DO …
   … parallelizable work …
    END DO
  … serial work …
!$OMP PARALLEL DO …
    DO …
   … more parallelizable work …
    END DO
  … serial work …
  … etc …
END PROGRAM big_science
```

This way may have lots of **synchronization** overhead.

Parallelizing a Serial Code #2

```
PROGRAM big_science
  … declarations …

   DO task = 1, numtasks
     CALL science_task(…)
   END DO
END PROGRAM big_science


SUBROUTINE science_task (…)
 … parallelizable work …

  … serial work …

  … more parallelizable work …

  … serial work …

  … etc …
END PROGRAM big_science
```



```
PROGRAM big_science
  … declarations …
!$OMP PARALLEL DO …
   DO task = 1, numtasks
     CALL science_task(…)
   END DO
END PROGRAM big_science


SUBROUTINE science_task (…)
 … parallelizable work …
!$OMP MASTER
  … serial work …
!$OMP END MASTER
  … more parallelizable work …
!$OMP MASTER
  … serial work …
!$OMP END MASTER
  … etc …
END PROGRAM big_science
```

TENTATIVE Schedule

Tue Jan 23: Storage: What the Heck is Supercomputing?
Tue Jan 30: The Tyranny of the Storage Hierarchy Part I
Tue Feb 6: The Tyranny of the Storage Hierarchy Part II
Tue Feb 13: Instruction Level Parallelism
Tue Feb 20: Stupid Compiler Tricks
Tue Feb 27: Shared Memory Multithreading
Tue March 6: Distributed Multiprocessing
Tue March 13: **NO SESSION** (Henry business travel)
Tue March 20: **NO SESSION** (OU's Spring Break)
Tue March 27: Applications and Types of Parallelism
Tue Apr 3: Multicore Madness
Tue Apr 10: High Throughput Computing
Tue Apr 17: **NO SESSION** (Henry business travel)
Tue Apr 24: GPGPU: Number Crunching in Your Graphics Card
Tue May 1: Grab Bag: Scientific Libraries, I/O Libraries, Visualization

Thanks for helping!

- OU IT
  - OSCER operations staff (Dave Akin, Patrick Calhoun, Kali McLennan, Jason Speckman, Brett Zimmerman)
  - OSCER Research Computing Facilitators (Jim Ferguson, Horst Severini)
  - Debi Gentis, OSCER Coordinator
  - Kyle Dudgeon, OSCER Manager of Operations
  - Ashish Pai, Managing Director for Research IT Services
  - The OU IT network team
  - OU CIO Eddie Huebsch
- OneNet: Skyler Donahue
- Oklahoma State U: Dana Brunson

This is an experiment!

It's the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the phone bridge to fall back on.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Coming in 2018!

- Coalition for Advancing Digital Research & Education (CADRE) Conference: Apr 17-18 2018 @ Oklahoma State U, Stillwater OK USA
  https://hpcc.okstate.edu/cadre-conference
- Linux Clusters Institute workshops
  http://www.linuxclustersinstitute.org/workshops/
  - Introductory HPC Cluster System Administration: May 14-18 2018 @ U Nebraska, Lincoln NE USA
  - Intermediate HPC Cluster System Administration: Aug 13-17 2018 @ Yale U, New Haven CT USA
- Great Plains Network Annual Meeting: details coming soon
- Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency Aug 5-10 2018, U Oklahoma, Norman OK USA
- PEARC 2018, July 22-27, Pittsburgh PA USA
  https://www.pearc18.pearc.org/
- IEEE Cluster 2018, Sep 10-13, Belfast UK
  https://cluster2018.github.io
- **OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2018, Sep 25-26 2018 @ OU**
- SC18 supercomputing conference, Nov 11-16 2018, Dallas TX USA
  http://sc18.supercomputing.org/

Thanks for your attention!

Questions?

www.oscer.ou.edu

References

[1] G. M. Amdahl, 1967: "Validity of the single-processor approach to achieving large scale computing capabilities." In *AFIPS Conference Proceedings* vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston VA, 1967, pp. 483-485. Cited in http://www.scl.ameslab.gov/Publications/AmdahlsLaw/Amdahls.html

[2] http://www.iso.org/iso/about/discover-iso_isos-name.htm

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald and R. Menon, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

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