Supercomputing in Plain English

An Introduction to High Performance Computing

Part V: Shared Memory Multithreading

Henry Neeman, Director
OU Supercomputing Center for Education & Research

Outline

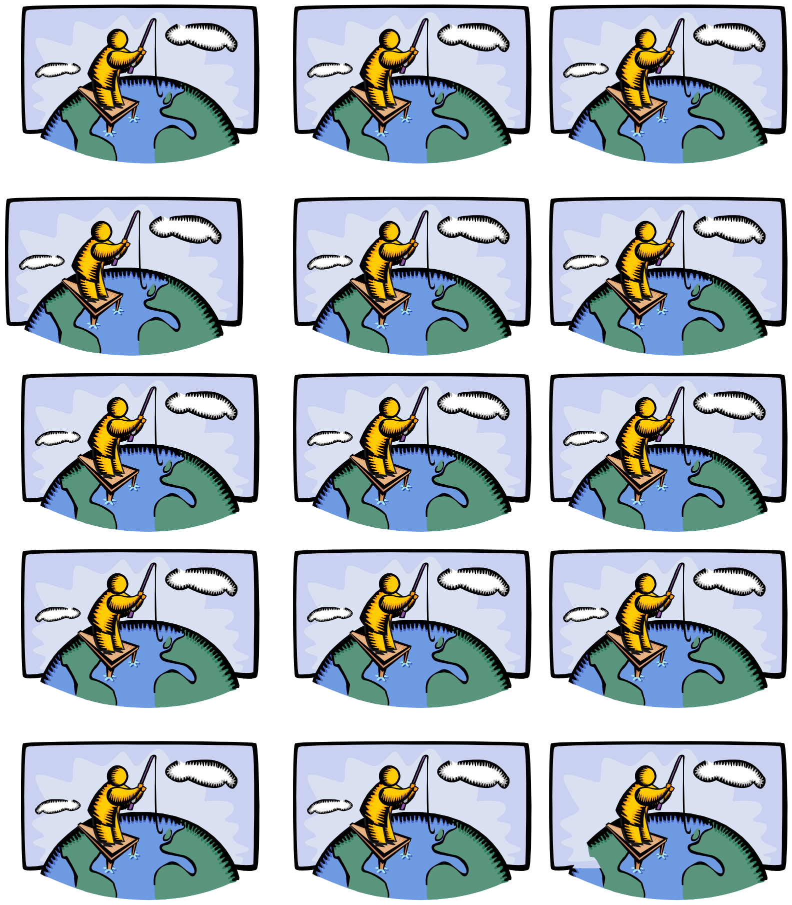
- Parallelism
- Shared Memory Parallelism
- OpenMP

Parallelism

Parallelism

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

Less fish …



More fish!

What Is Parallelism?

Parallelism is the use of multiple processing units – either processors or parts of an individual processor – to solve a problem, and in particular the use of multiple processing units operating concurrently on different parts of a problem.

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

Kinds of Parallelism

- Shared Memory Multithreading (our topic today)
- Distributed Memory Multiprocessing (next time)
- Hybrid Shared/Distributed (your goal)

Why Parallelism Is Good

- The Trees: We like parallelism because, as the number of processing units working on a problem grows, we can solve our 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: execution sequences that share a single memory area ("address space")
- Processes: execution sequences with their own independent, private memory areas

… and thus:

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

As a general rule, 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, these terms are often used interchangeably:

- Parallelism
- Concurrency (not as popular these days)
- 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, and S_p is the speedup achieved in the parallel part.[1]

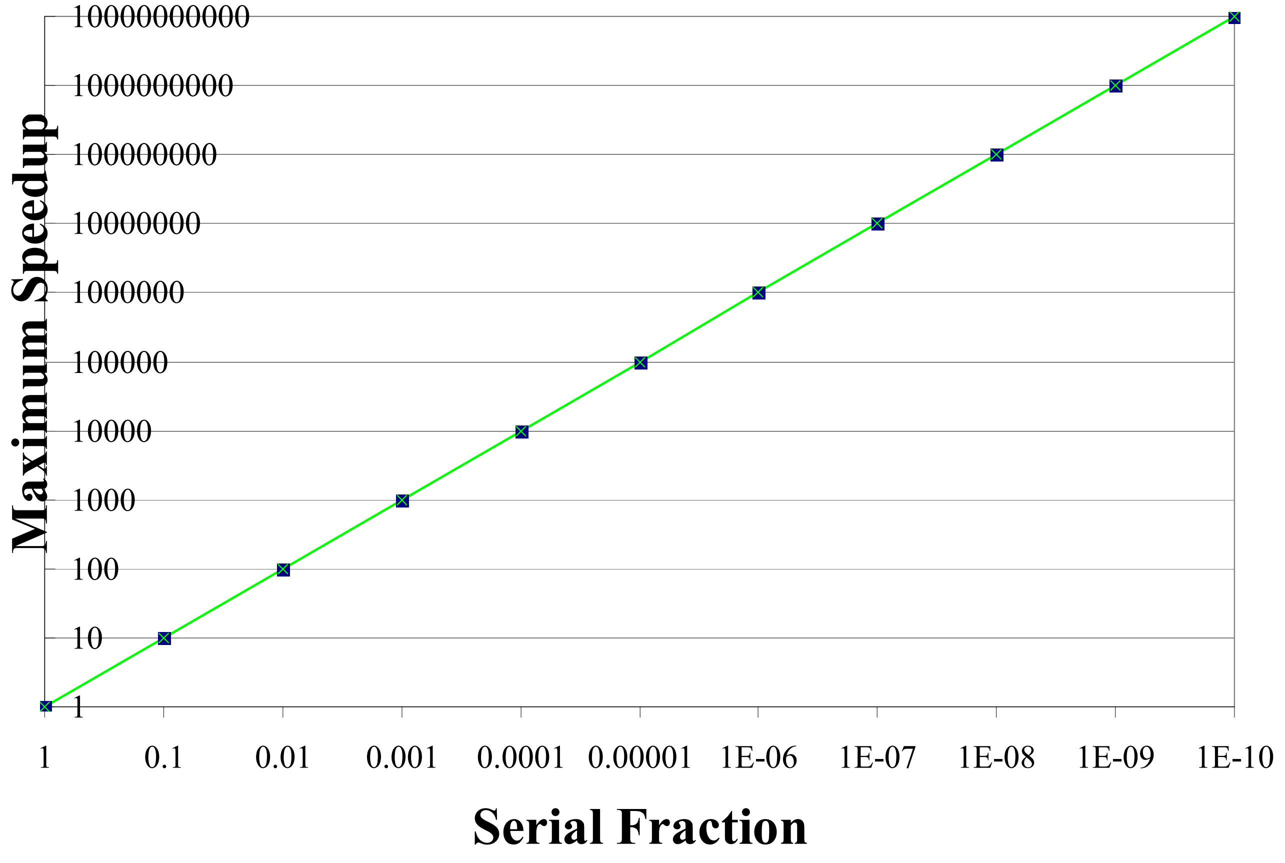
Amdahl’s Law: Huh?

What does Amdahl’s Law tell us? Well, imagine that you run your code on a zillion processors. The parallel part of the code could exhibit up to a factor of a zillion speedup. 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

```
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 !! index = 1, a_lot
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**.

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.

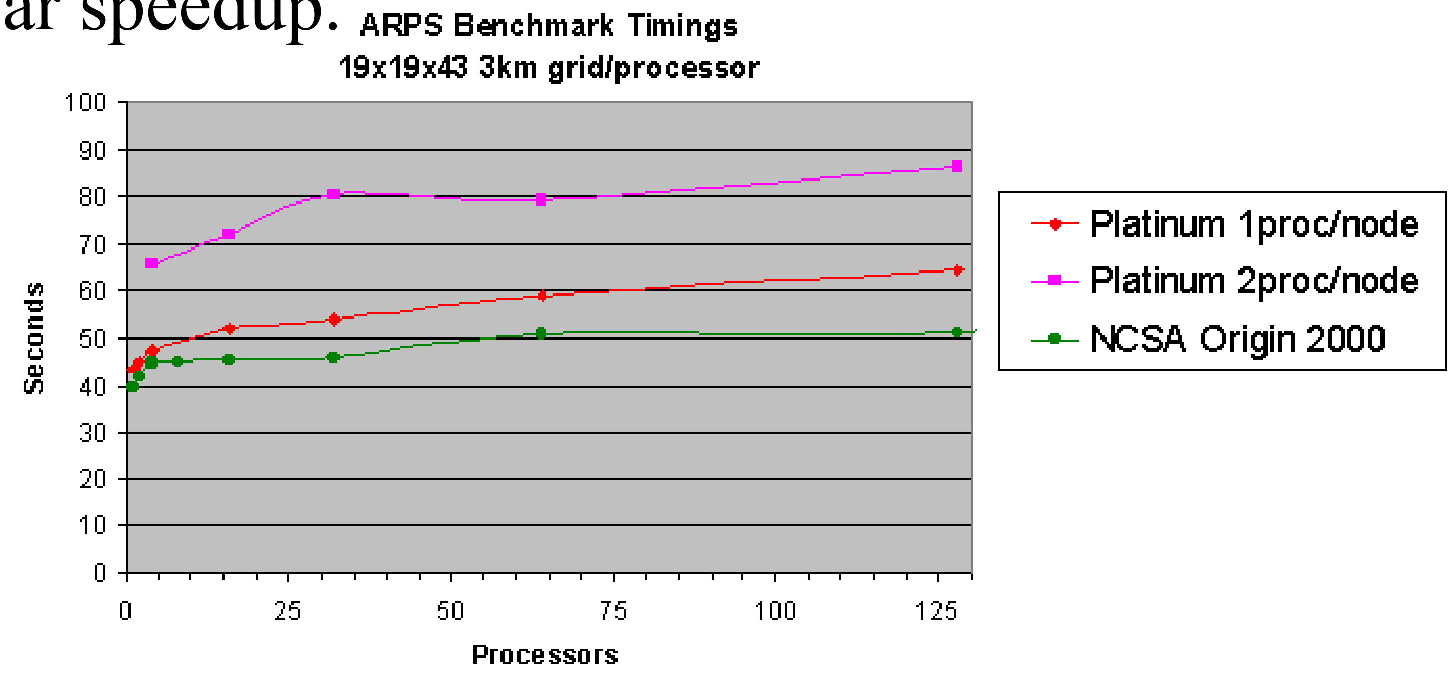
Speedup

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.

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 big parallel parts) and some are fine grain (many little parallel parts).

Usually, coarse grain codes are more scalable than fine grain codes, because less time is spent managing the parallelism, so more is spent getting the work done.

Parallel Overhead

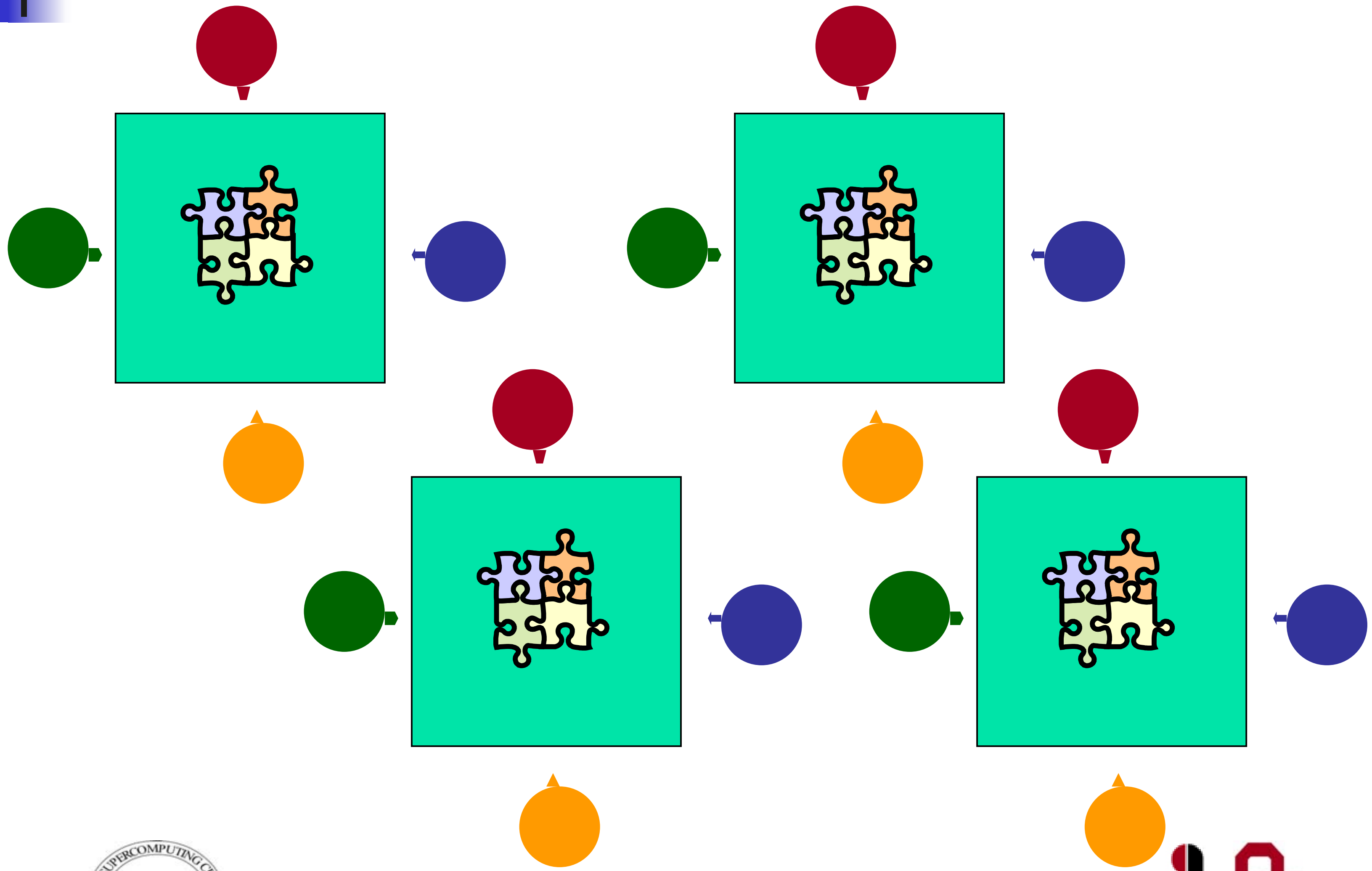
Parallelism isn’t free. Behind the scenes, the compiler and the hardware have to do a lot of <u>overhead</u> work to make parallelism happen.

The overhead typically includes:

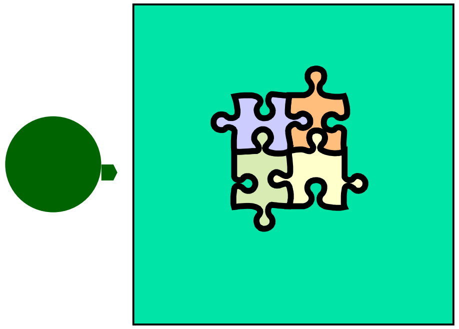
- Managing the multiple threads/processes
- Communication between threads/processes
- Synchronization (described later)

Shared Memory Parallelism

The Jigsaw Puzzle Analogy



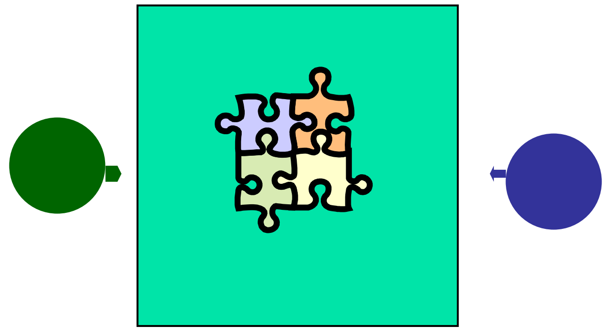
Serial Computing



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

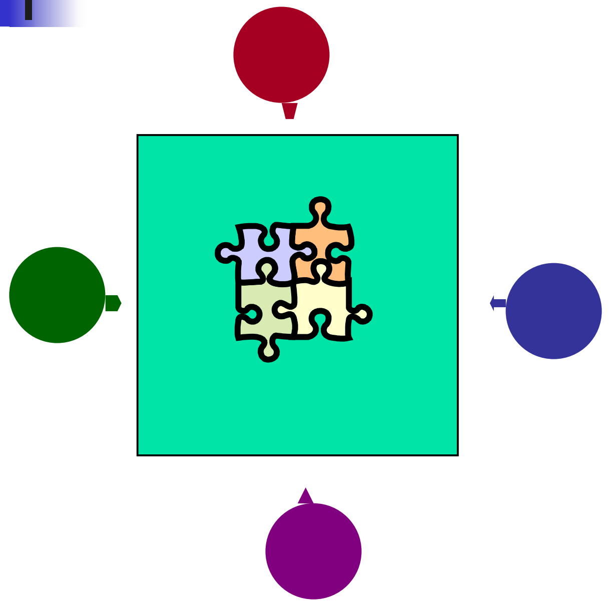
We can imagine that it'll take you a certain amount of time. Let's say that you can put the puzzle together in an hour.

Shared Memory Parallelism



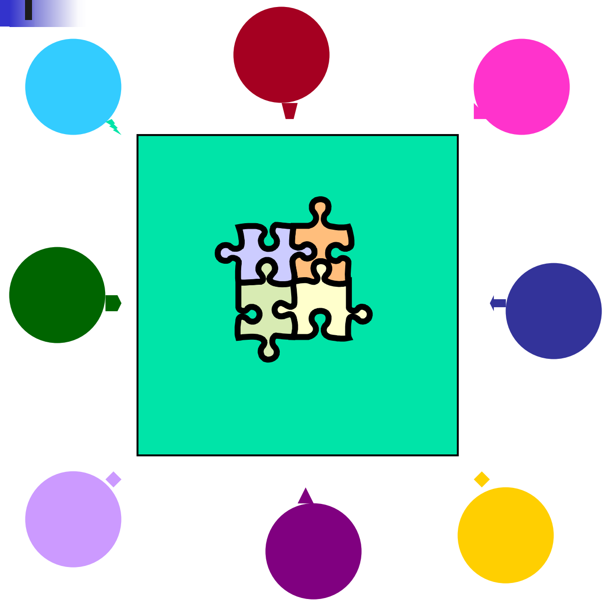
If Julie sits across the table from you, then she can work on her half of the puzzle and you can work on yours. Once in a while, you'll both reach into the pile of pieces at the same time (you'll contend for the same resource), which will cause a little bit of slowdown. And from time to time you'll have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

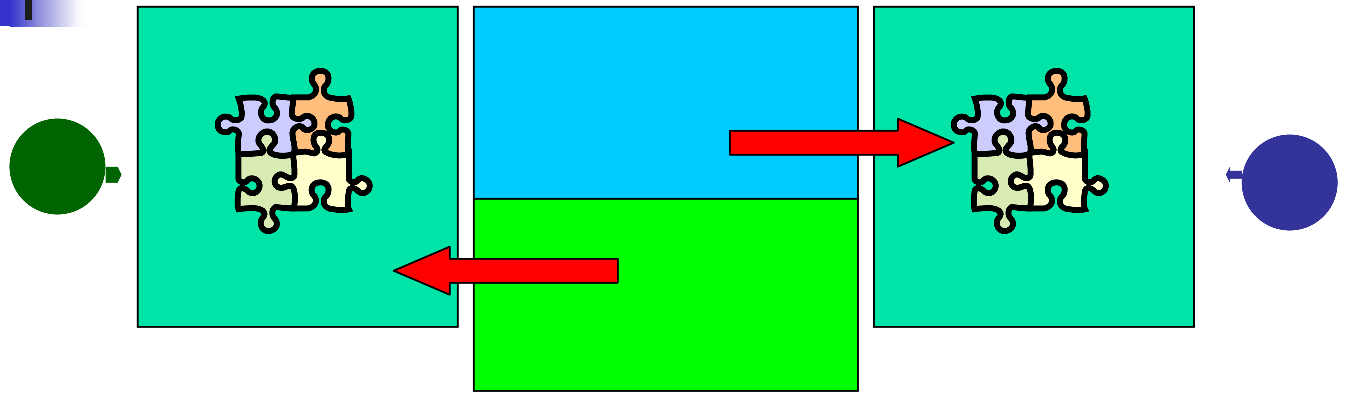
Diminishing Returns



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.

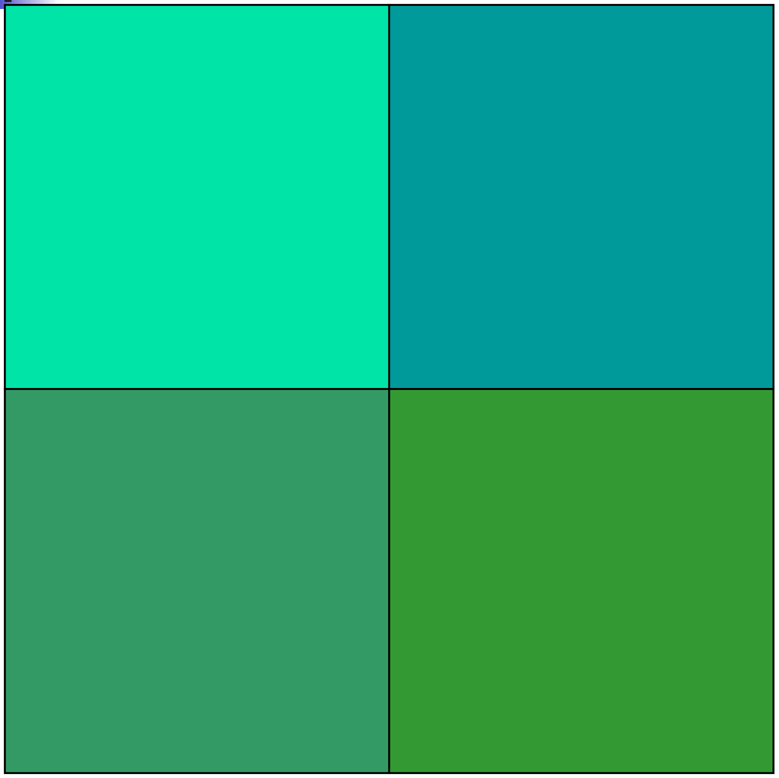
Load Balancing

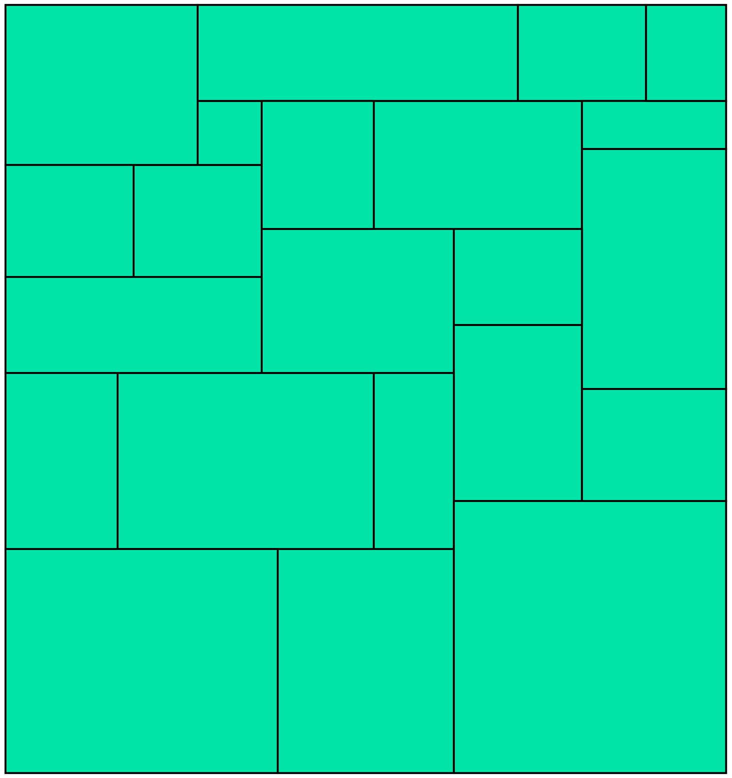


Load balancing means giving everyone roughly the same amount of work to do.

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.

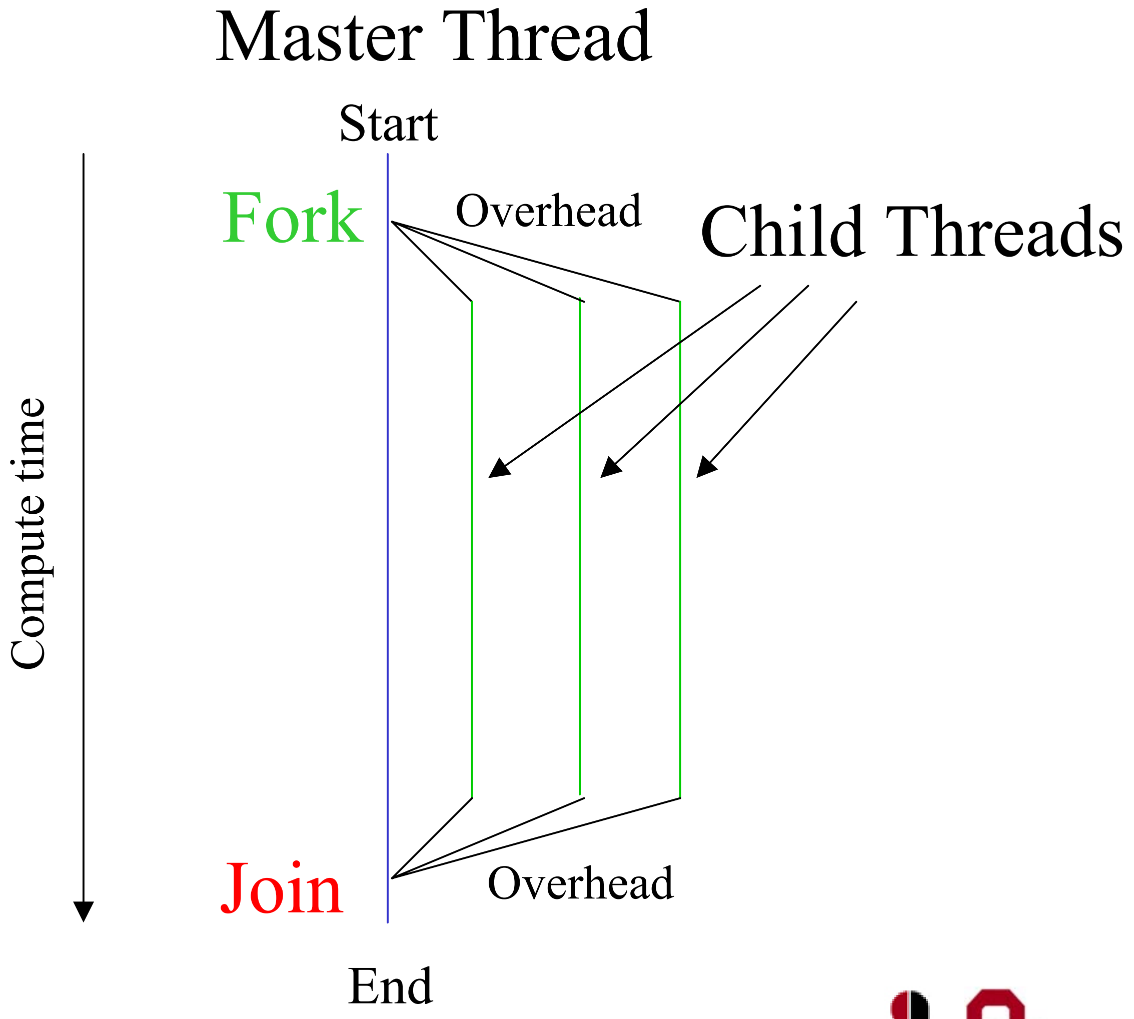
The Fork/Join Model

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

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

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

The Fork/Join Model (cont'd)



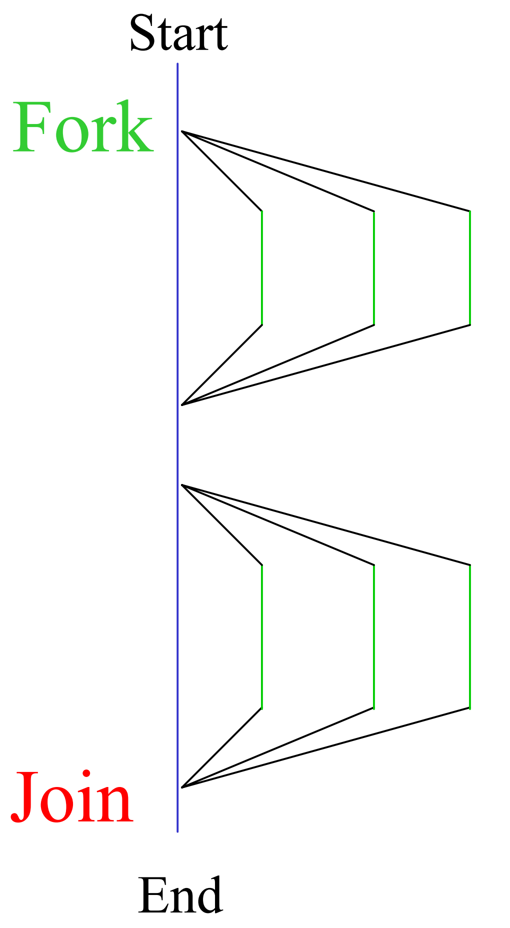
The Fork/Join Model (cont'd)

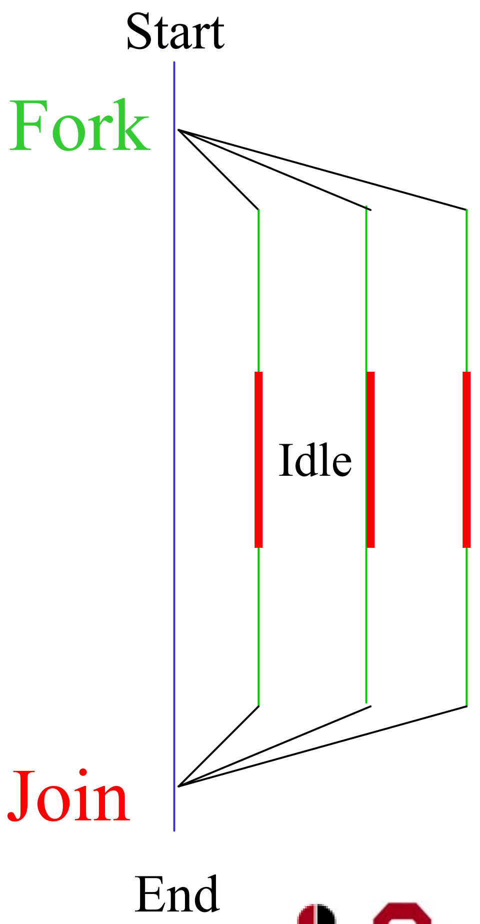
In principle, as a parallel section completes, the child threads shut down (join the master), forking off again when the master reaches another parallel section.

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 simply 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. <u>Gang scheduling</u> is preferable, because then all of the processors are guaranteed to be available for the whole run.

OpenMP

Most of this discussion is from [2], with a little bit from [3].

What Is OpenMP?

OpenMP is a standardized way of expressing shared memory parallelism.

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

When you compile a program that has OpenMP in it, 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++.

Compiler Directives

A <u>compiler directive</u> 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’
```

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" is the Greek word for "thing." Go figure.

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

```
PROGRAM hello_world
  IMPLICIT NONE
  INTEGER :: number_of_threads, this_thread, iteration

  number_of_threads = omp_get_max_threads()
  WRITE (0,"(I2,A)") number_of_threads, " threads"
!$OMP PARALLEL DO DEFAULT(PRIVATE) &
!$OMP                  SHARED(number_of_threads)
  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 !! iteration = 0, number_of_threads - 1
END PROGRAM hello_world
```

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

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 !! index = 1, length
```

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

A Change to `hello_world`

Suppose we do 3 loops 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 computing, 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 DEFAULT(PRIVATE) SHARED(number_of_threads)
```

What do **`DEFAULT(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

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 !! iteration = 0, number_of_threads - 1
```

Notice that, if the iterations of the loop are executed concurrently, 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

```
!$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 !! iteration = 0, number_of_threads - 1
```

Notice that, if the iterations of the loop are executed concurrently, 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

```
!$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 !! iteration = 0, number_of_threads - 1
```

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)
```

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.)

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

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 ((index / 1000) < 1) THEN
      y(index) = LOG(x(index))
    ELSE
      y(index) = x(index) + 2
    END IF
  END DO !! index = 1, length
```

The threads that finish early have to wait.

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.

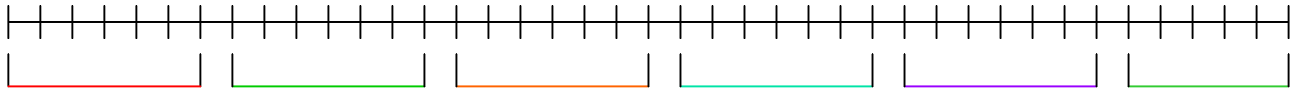
Scheduling Strategies

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: each thread gets smaller and smaller chunks over time – a compromise

Static Scheduling

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



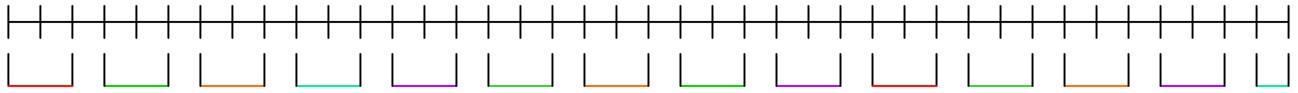
- Thread #0: iterations 0 through N_i/N_t-1
- Thread #1: iterations N_i/N_t through $2N_i/N_t$-1
- Thread #2: iterations $2N_i/N_t$ through $3N_i/N_t$-1

…

- Thread #N_t-1: iterations $(N_t-1)N_i/N_t$ through N_i-1

Dynamic Scheduling

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

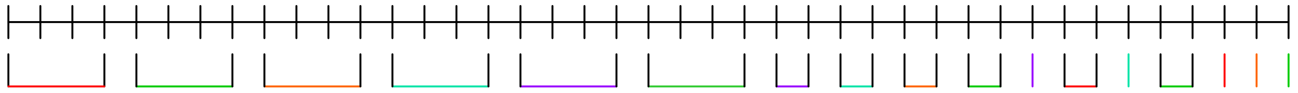


When a particular thread finishes its chunk of iterations, it gets assigned a new chunk. So, the relationship between iterations and threads is nondeterministic.

- Advantage: very flexible
- Disadvantage: high overhead – lots of decision making about which thread gets each chunk

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:



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

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, new versions of the compiler, who’s on the machine, what day of the week it is, etc, so you may want to 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 the parallel section is called <u>synchronization</u>.

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?

Synchronizing is necessary when the code that follows 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 !! index = 1, length
! Need to synchronize here!
  DO index = 1, length
    z(index) = y(index) + y(length - index + 1)
  END DO !! index = 1, length
```

Barriers

A barrier is a place where synchronization is forced to occur; that is, where faster threads have to wait for slower ones.

The **PARALLEL DO** directive automatically puts a 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 !! index = 1, length
```

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 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 <u>race condition</u>: the result depends on who wins the race.

Reductions

A <u>reduction</u> converts an array to a scalar: 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 = 1, length
```

This is equivalent to:

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

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
```

Next Time

Part VI:

Distributed Multiprocessing

References

[1] Amdahl, G.M. “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] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald and R. Menon, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

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