"Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end"

Herb Sutter, Chair of the ISO C++ Standards Committee, Microsoft, 2008

Lecture B.4:
Shared-Memory Parallel Processing

CS205: Computing Foundations for Computational Science
Dr. David Sondak
Spring Term 2020

HARVARD School of Engineering and Applied Sciences

IACS INSTITUTE FOR APPLIED COMPUTATIONAL SCIENCE AT HARVARD UNIVERSITY

Lectures developed by Dr. Ignacio Llorente

Before We Start

Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

Intro: Large-Scale Computational and Data Science

A. Parallel Processing Fundamentals

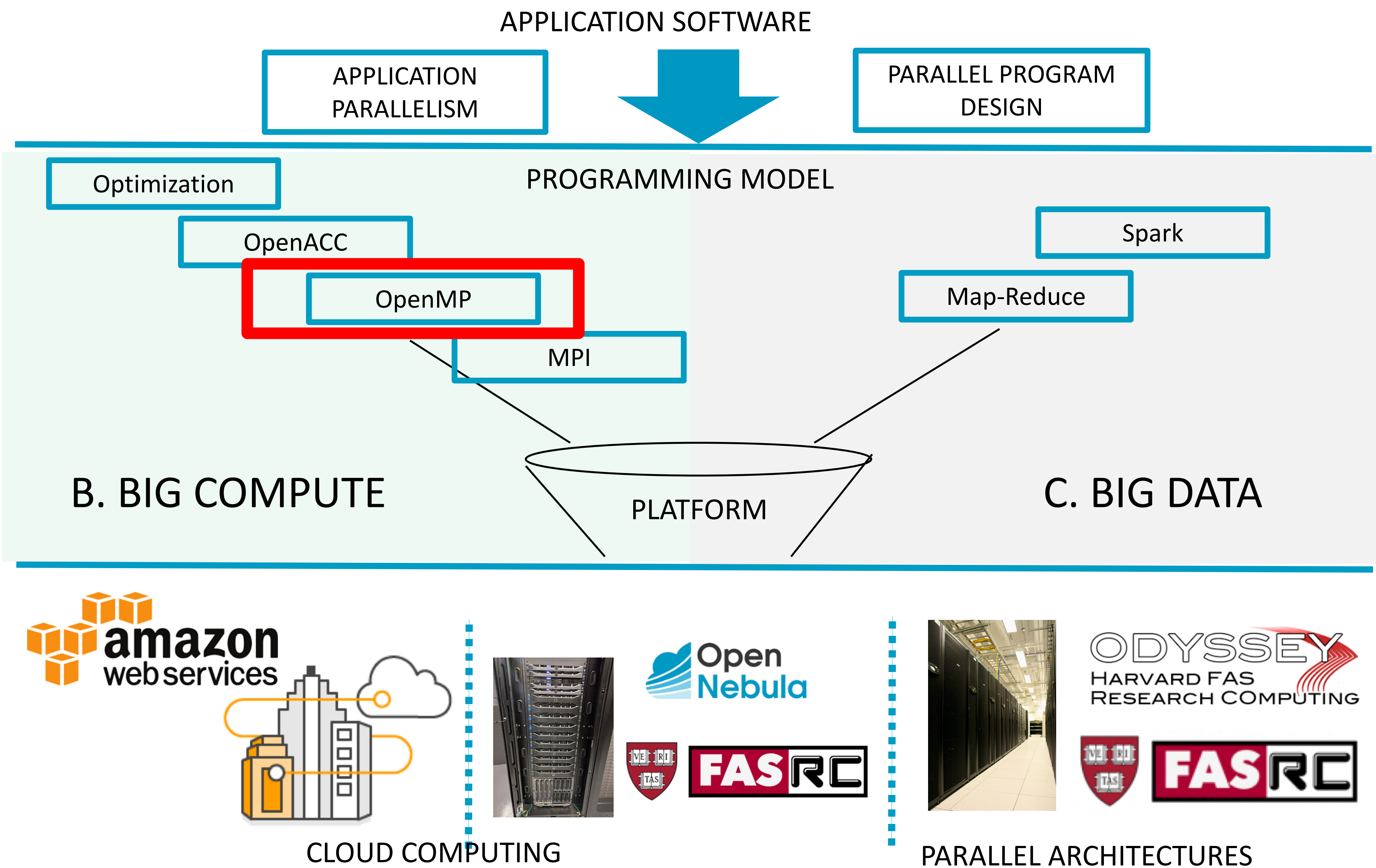
B. Parallel Computing

- B.1. Foundations of Parallel Computing
- B.2. Performance Optimization
- B.3. Accelerated Computing
- B.4. Shared-memory Parallel Processing
- B.5. Distributed-memory Parallel Processing

C. Parallel Data Processing

Wrap-Up: Advanced Topics

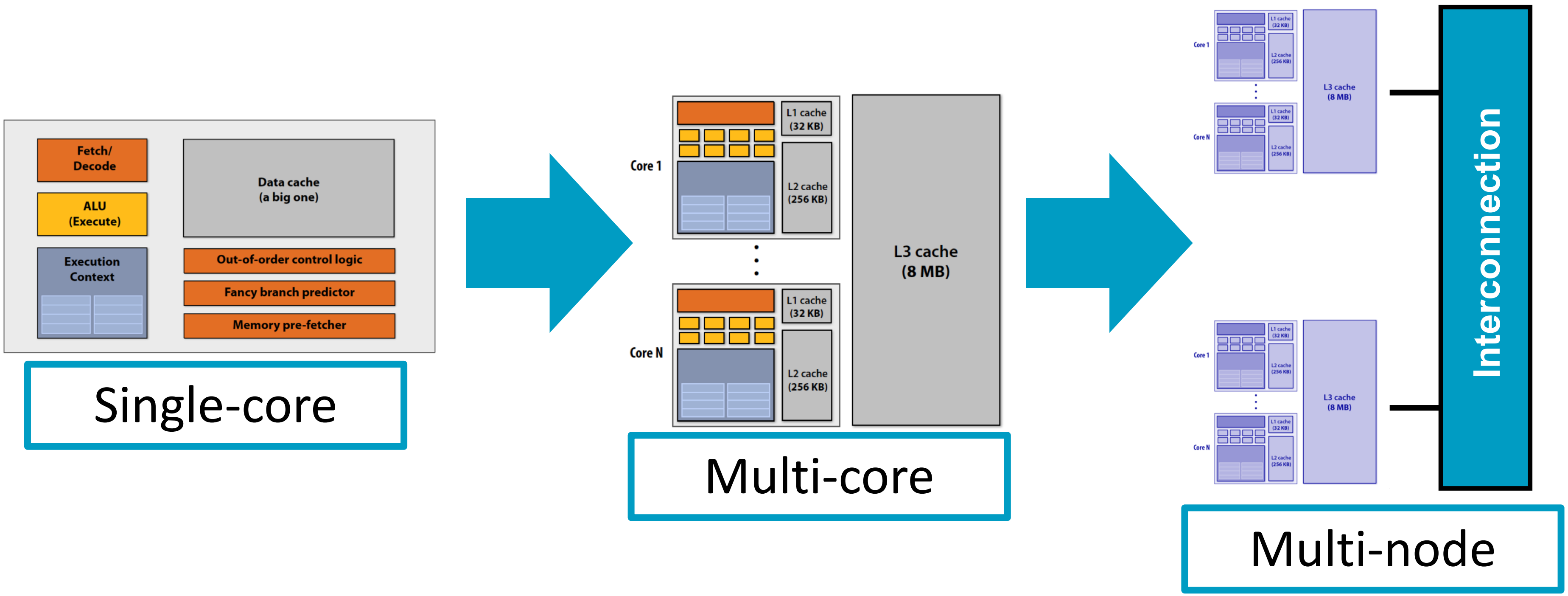
CS205: Contents



Context

Shared-Memory Parallel Processing

How can I make efficient use of multiple cores?



Roadmap

Shared-Memory Parallel Processing

Shared-Memory Basics

OpenMP Fundamentals

Data Dependencies

Automatic Parallelization

Parallelization Process

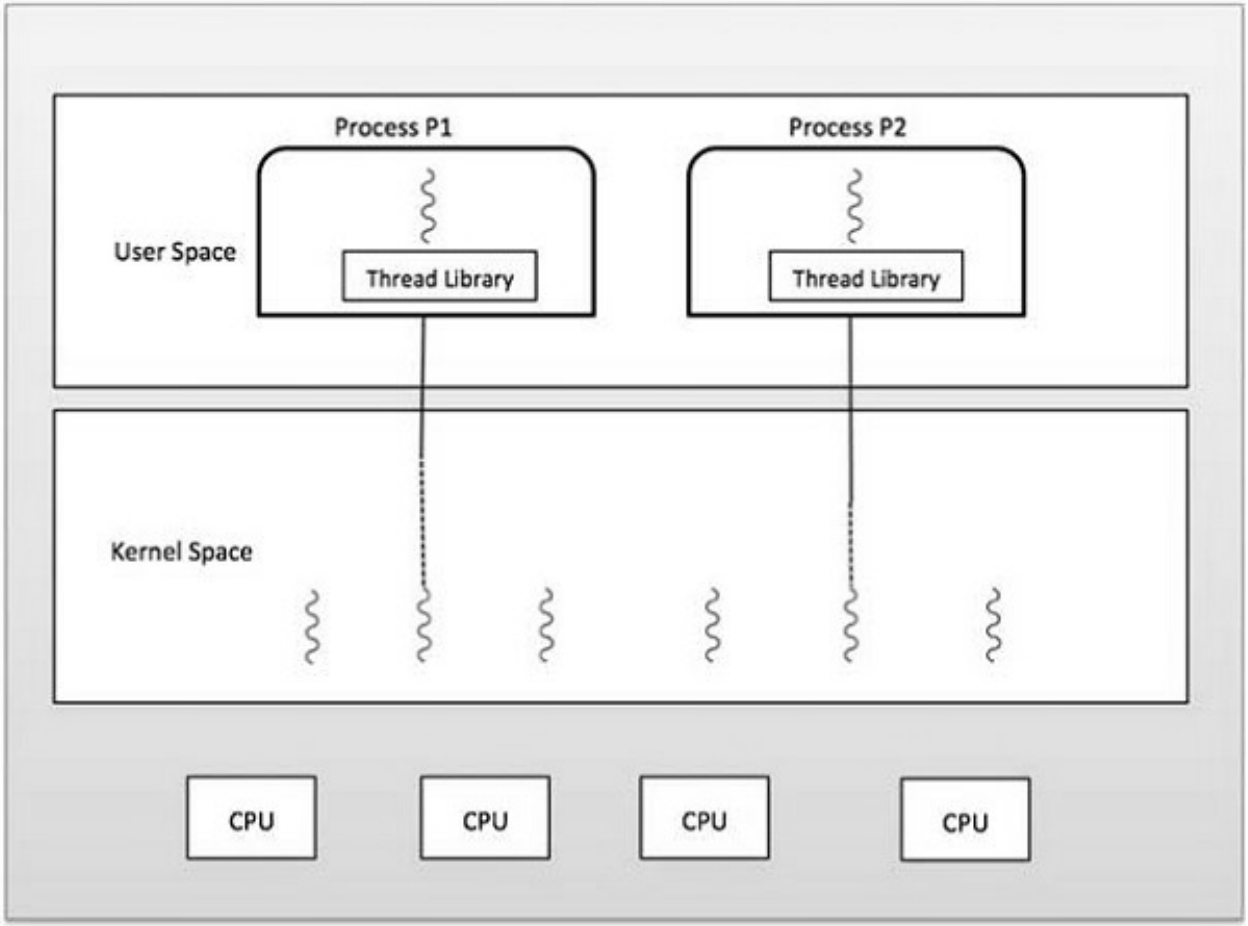
Shared-Memory Basics

Thread Programming

- A process is an instance of a computer program that is being executed

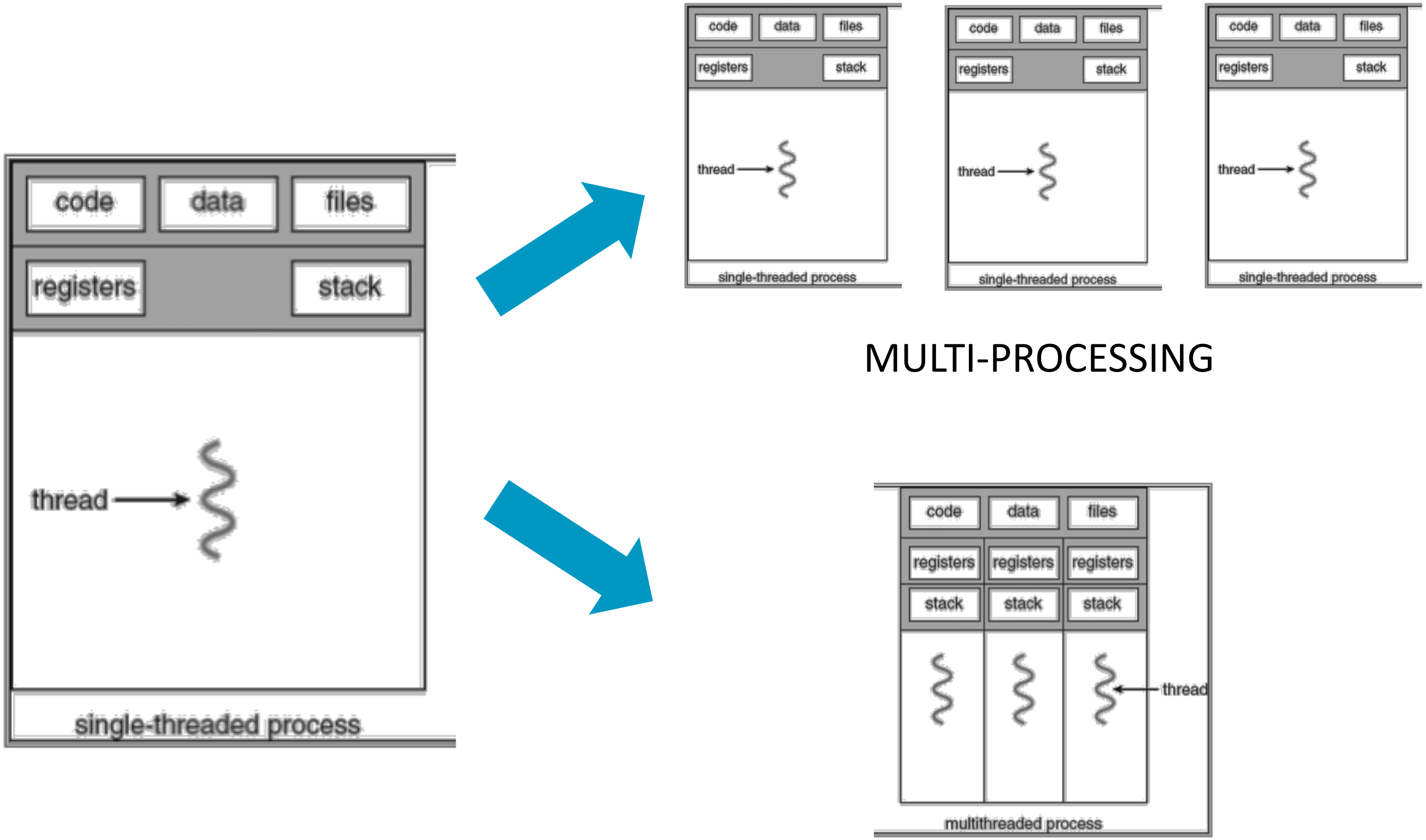
- A process can have 1 or several threads

- The kernel of the OS schedule threads to multiple cores



Multi-processing Basics

Multi-Processing vs Multi-Threading



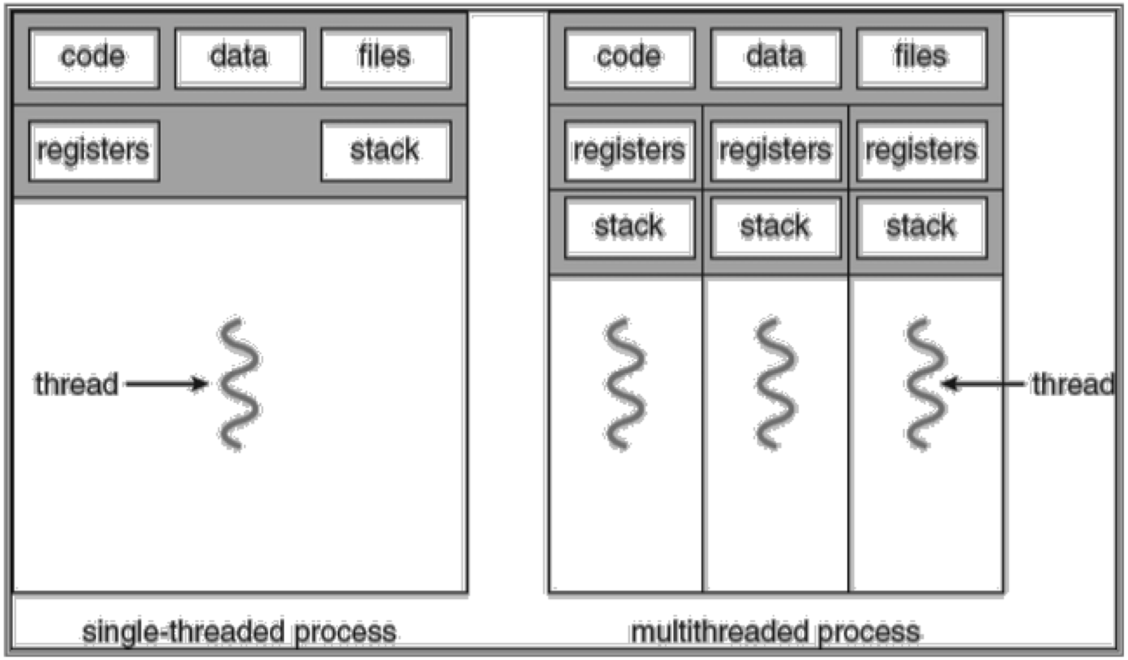
MULTI-PROCESSING

MULTI-THREADING

Shared-Memory Basics

Thread Programming

- Threads of execution: most popular abstraction for concurrency
  - ✓ Created before parallel systems to allow concurrency
  - ✓ Example: Threaded web server for many clients simultaneously
- All threads in one process share same memory, file descriptors, etc.
- Allows one process to use multiple cores and CPUs



Shared-Memory Basics

Elements of Programming

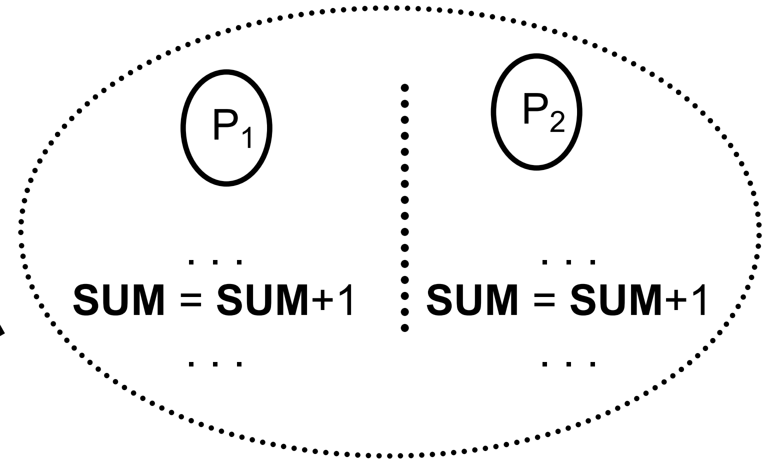
- Shared Memory Collaboration
  - ✓ Threads share memory address space
- Fork/join threads
- Synchronization to ensure no data corruption
  - ✓ Barrier
  - ✓ Mutual exclusive (mutex and lock/unlock)
- Assign/distribute work to threads
  - ✓ Work share
- Run time control
  - ✓ Query/request available resources
  - ✓ Interaction with OS, compiler, etc.

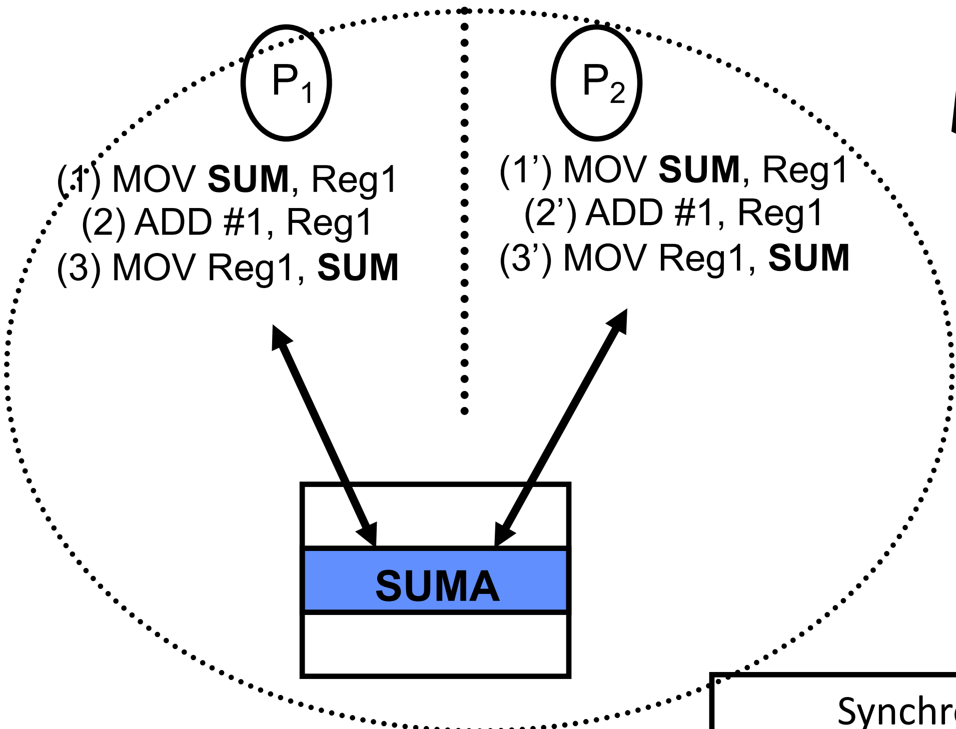
Shared-Memory Basics

Race Conditions

A Race Condition occurs, if

- Two or more processes manipulate a shared resource concurrently, and
- The outcome of the execution depends on the particular order in which the access takes place





(1')(1)(2)(3)(2')(3') => **SUM** = **SUM** +1
(1)(1')(2')(3')(2)(3) => **SUM** = **SUM** +1
(1)(2)(3)(1')(2')(3') => **SUM** = **SUM** +2

Synchronization needed to prevent race conditions
MUTUAL EXCLUSION
Prevents simultaneous access to a shared resource

Shared-Memory Basics

Synchronization

Variable mutex: *S*
Boolean: 0 / 1
General: Integer >= 0

Functions:

Lock(S)
    If *S* == 0 then wait to *S* > 0
    If *S* > 1 then *S* = *S - 1*
Unlock(S):
    S = *S* + *1*

P1: Lock(S)
    Critical Section
    Unlock(S)

P2: Lock(S)
    Critical Section
    Unlock(S)

P_1

. . .
Lock(S)
SUM = SUM+1
Unlock(S)
. . .

P_2

. . .
Lock(S)
SUM = SUM+1
Unlock(S)
. . .

(1)(2)(3)(1')(2')(3') => **SUM** = **SUM** +2

(1')(2')(3')(1)(2)(3) => **SUM** = **SUM** +2

Shared-Memory Basics

Example: Hello World with Posix Threads

```
void *print_message_function( void *ptr );
pthread_mutex_t mutex;
main()
{
     pthread_t thread1, thread2;
     pthread_attr_t pthread_attr_default;
     pthread_mutexattr_t pthread_mutexattr_defa
     struct timespec delay;
     char *message1 = "Hello";
     char *message2 = "World\n";

     delay.tv_sec = 10;
     delay.tv_nsec = 0;

     pthread_attr_init(&pthread_attr_default);
     pthread_mutexattr_init(&pthread_mutexattr_default);

     pthread_mutex_init(&mutex, &pthread_mutexattr_default);
     pthread_mutex_lock(&mutex);

     pthread_create( &thread1, &pthread_attr_default,
                     (void *) print_message_function, (void *) message1);
     pthread_mutex_lock(&mutex);
     pthread_create(&thread2, &pthread_attr_default,
                     (void *) print_message_function, (void *) message2);
     pthread_mutex_lock(&mutex);
     exit(0);
}
```

```
void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s ", message);
     pthread_mutex_unlock(&mutex);
     pthread_exit(0);
}
```

Shared-Memory Basics

Different Libraries and Approaches

OpenMP

High level of abstraction

Posix Threads

OS independent, but still requires thread management and synchronization

OS Threads

OS dependent, use of low level functionality

SIMPLICITY
PORTABILITY

PERFORMANCE
FUNCTIONALITY

OpenMP Fundamentals

What Is it?

OpenMP Fundamentals

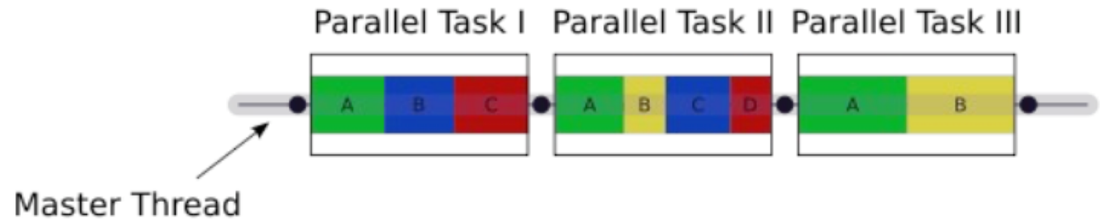
Why OpenMP?

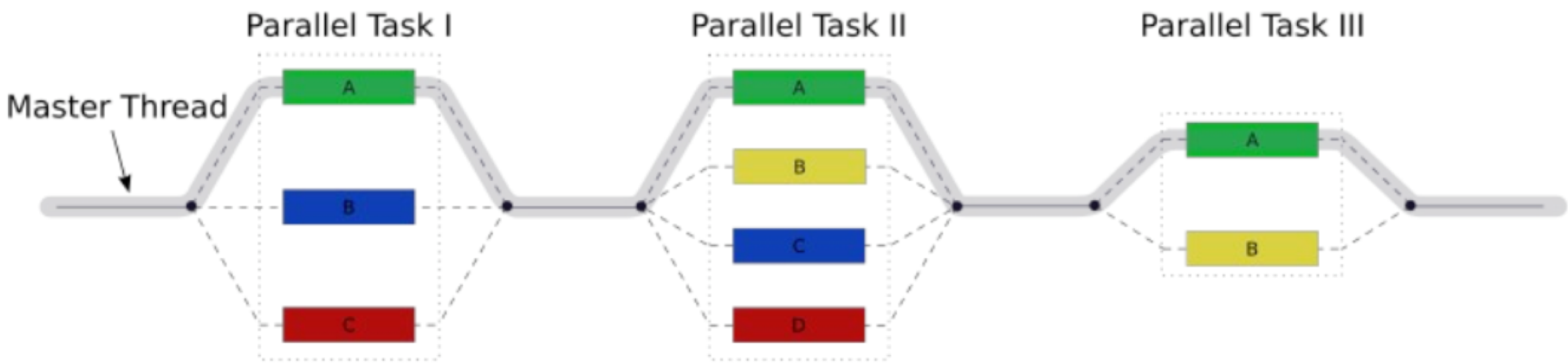
- Simplicity
- It is directly supported by the compiler
- Leave thread management to the compiler
- Widely supported
- **Automatic parallelization as first step**
- Work on the sequential code
- Incremental parallelization possible

OpenMP Fundamentals

Execution Model

- Programs begin as a single process: master thread
- Master executes in serial mode until a parallel region
- Master creates a team of parallel threads (fork) that simultaneously execute statements in the parallel region
- After executing the parallel region, team threads synchronize and terminate (join), but master continues





OpenMP Fundamentals

A Simple Example: Parallel SAXPY

```
const int n = 10000;
float x[n], y[n], a;
int i;
for (i=0; i<n; i++) {
  y[i] = a * x[i] + y[i];
}
```

```
const int n = 10000;
float x[n], y[n], a;
int i;
#pragma omp parallel for
for (i=0; i<n; i++) {
  y[i] = a * x[i] + y[i];
}
```

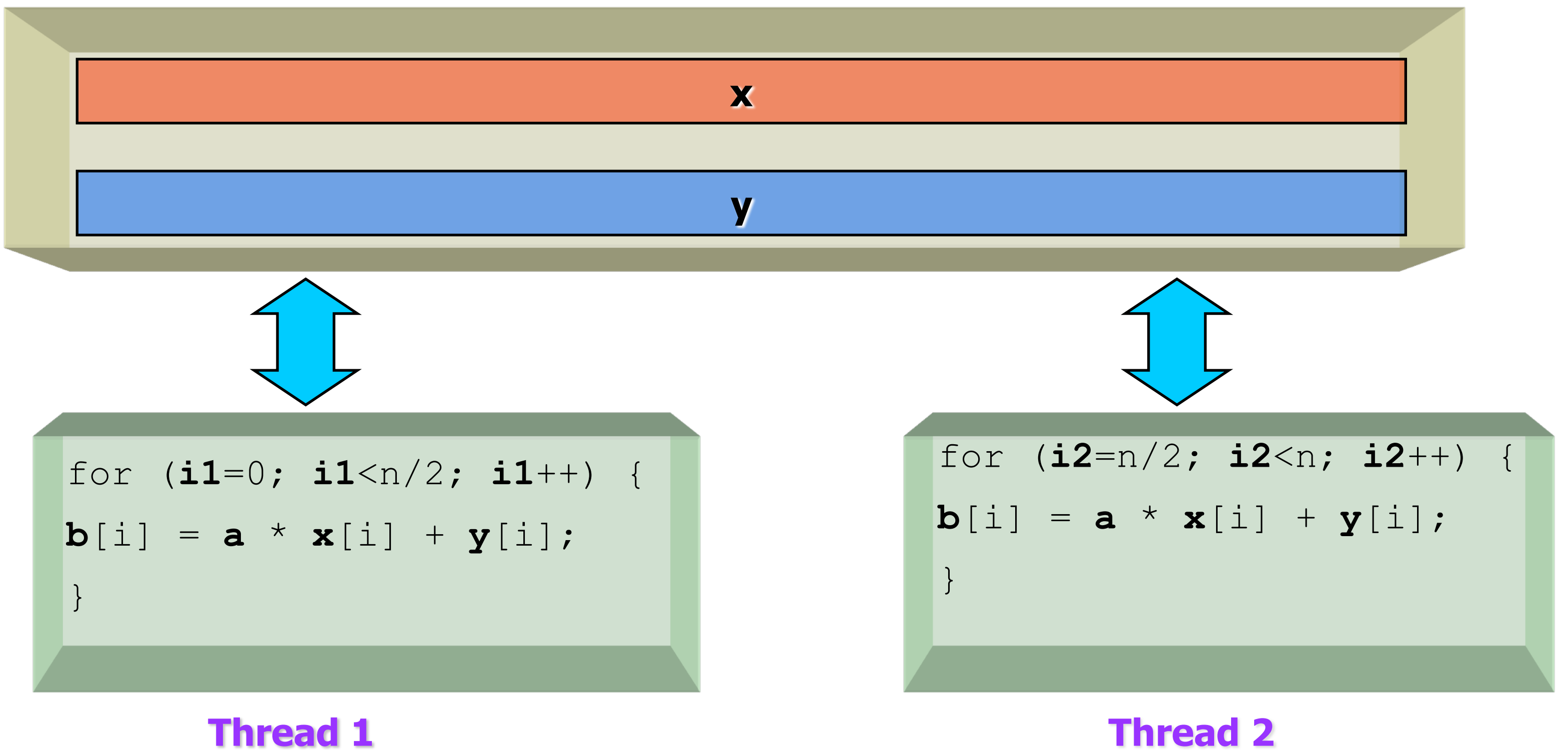
Main programming challenges

- Shared vs. Private variables
- Loop scheduling

OpenMP Fundamentals

A Simple Example: Parallel SAXPY (Scope of Variables)

```
#pragma omp parallel for
for (i=0; i<n; i++) {
  b[i] = a * x[i] + y[i];
}
```



OpenMP Fundamentals

A Simple Example: Parallel SAXPY (Loop Scheduling)

```
#pragma omp parallel for
for (i=0; i<n; i++) {
  b[i] = a * x[i] + y[i];
}
```

static chunk=1

```
for (i=0; i<n; i=i+2) {
b[i] = a * x[i] + y[i];
}
```

```
for (i=1; i<n; i=i+2) {
b[i] = a * x[i] + y[i];
}
```

default

```
for (i=0; i<n/2; i++) {
b[i] = a * x[i] + y[i];
}
```

```
for (i=n/2; i<n; i++) {
b[i] = a * x[i] + y[i];
}
```

Thread 1 **Thread 2**

OpenMP Fundamentals

A Simple Example: Pi

```
#include <stdio.h>
#include <omp.h>
#define N 2000000000
int main(void) {
  double pi = 0.0f;
  long long i;
#pragma omp parallel for reduction(+:pi) private(i,t), shared(N)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

OpenMP Fundamentals

Programming Model

- Compiler directives specify parallel regions (similar to OpenAcc!)
- Header file: `#include <omp.h>`

```
#pragma omp directive [clause [[,] clause]...]
```

Parallel Regions

```
#pragma omp parallel [clause [[,] clause]...]
```

Work Sharing Constructs

```
#pragma omp for [clause [[,] clause]...]
#pragma omp sections [clause [[,] clause]...]
#pragma omp critical
#pragma omp single
```

OpenMP Fundamentals

Parallel Region

- To fork a team of N threads, numbered 0,1,..,N-1
- Probably the most important construct in OpenMP
- Implicit barrier

```
//sequential code here (master thread)
#pragma omp parallel [clauses] {
  // parallel computing here
  // ...
}
// sequential code here (master thread)
```

clauses

shared	nowait	copyin
if	reduction	private
firstprivate	num_threads	default

OpenMP Fundamentals

Parallel Region

Work Sharing

- We have not yet discussed how work is distributed among threads...
- Without specifying how to share work, all threads will redundantly execute all the work (i.e. no speedup!)
- The choice of work-share method is important for performance
- OpenMP work-sharing constructs
  - ✓ Loop ("for" in C/C++; "do" in Fortran)
  - ✓ Sections
  - ✓ Single
  - ✓ Critical

OpenMP Fundamentals

Loop Construct

```
#pragma omp parallel shared(n,a,b) private(i)
{ #pragma omp for
  for (i=0; i<n; i++)
    a[i]=i;
  #pragma omp for
  for (i=0; i<n; i++)
    b[i] = 2 * a[i];}
```

```
#pragma omp parallel for shared(n,a,b) private(i)
for (i=0; i<n; i++)
    a[i]=i;
```

clauses

shared	nowait	schedule
lastprivate	reduction	**private**
firstprivate	ordered	

OpenMP Fundamentals

Clauses

`Private` Variables => Each thread maintains its own variable

- The values of private data are undefined upon entry to and exit from the specific construct
- To ensure the last value is accessible after the construct, consider using "`lastprivate`"
- To pre-initialize private variables with values available prior to the region, consider using "`firstprivate`"
- Loop iteration variable is private by default

`Shared` Variables => Each thread can read or modify the variable

- Shared among the team of threads executing the region
- Data corruption is possible when multiple threads attempt to update the same memory location
  - ✓ Data race condition
  - ✓ Memory store operation not necessarily atomic
- Code correctness is user's responsibility

OpenMP Fundamentals

Clauses

`nowait` clause

- This is useful inside a big parallel region
- Allows threads that finish earlier to proceed without waiting
- Less synchronization – may improve performance

```
#pragma omp for nowait
// for loop here
#pragma omp for nowait
...
```

`if (integer expression)` clause

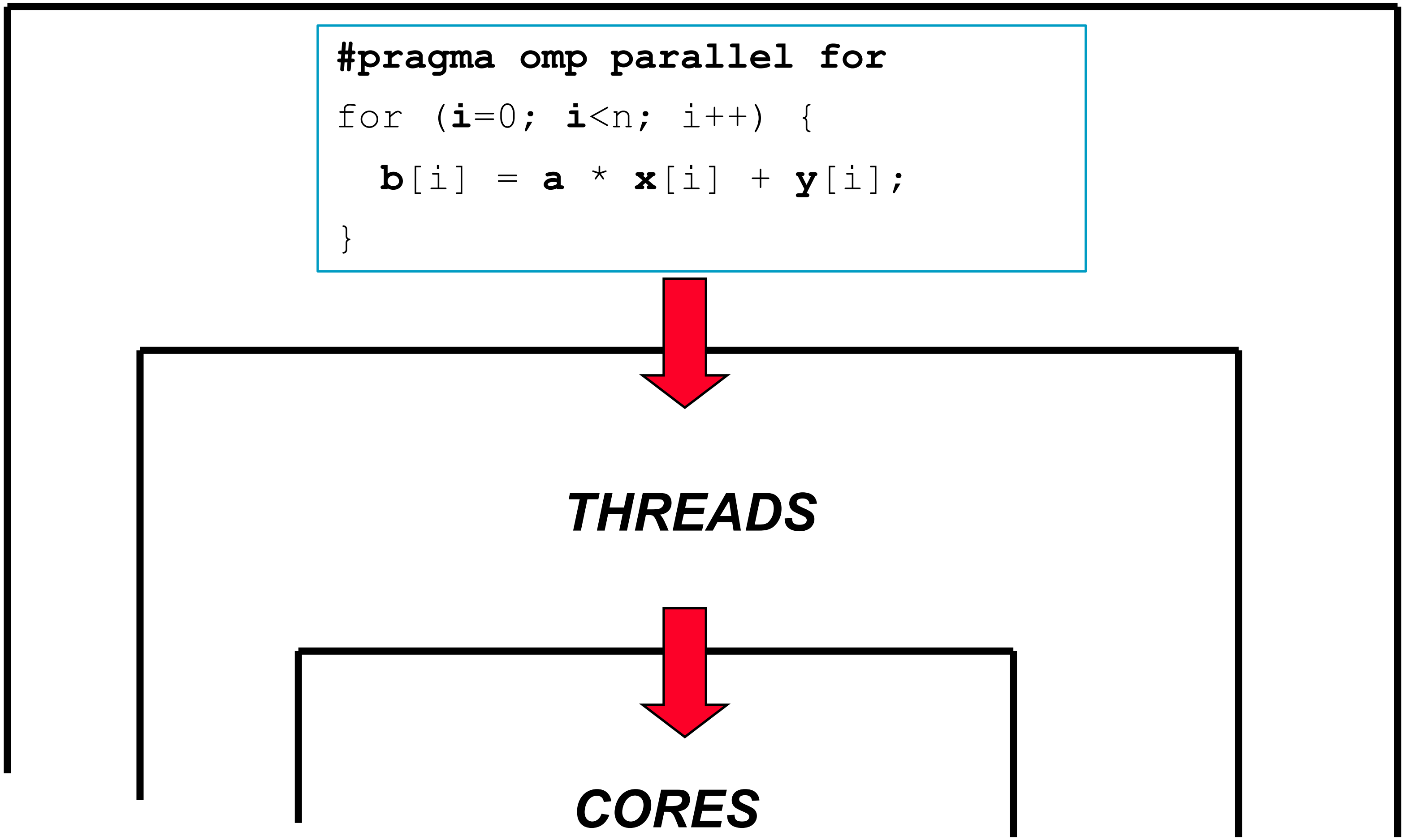
- Determine if the region should run in parallel
- Useful option when data is too small (or too large)

```
#pragma omp parallel if (n>100)
{
//...some stuff
}
```

OpenMP Fundamentals

Loop Scheduling

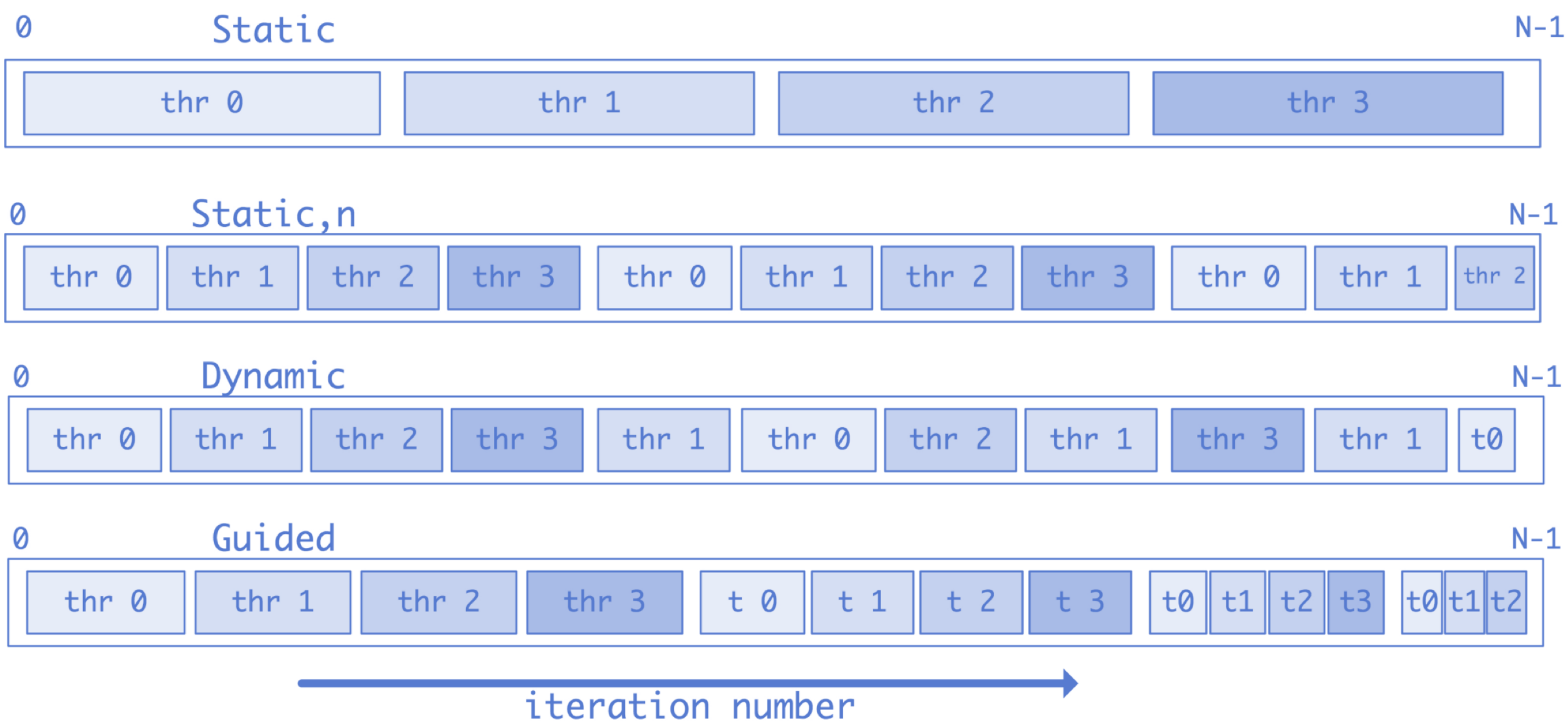
```
#pragma omp parallel for
for (i=0; i<n; i++) {
  b[i] = a * x[i] + y[i];
}
```



OpenMP Fundamentals

Loop Scheduling

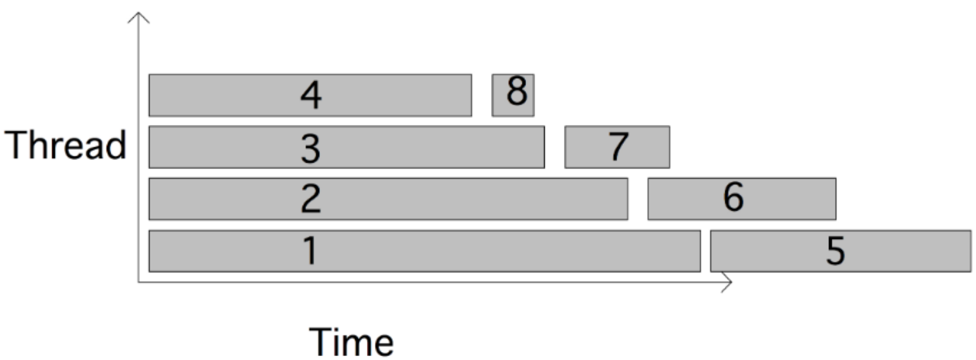
Data Clauses	Comment
`static`	Each thread is assigned a fixed-size chunk (default)
`dynamic`	Work is assigned as a thread requests it
`guided`	Big chunks first and smaller and smaller chunks later
`runtime`	Use environment variable to control scheduling



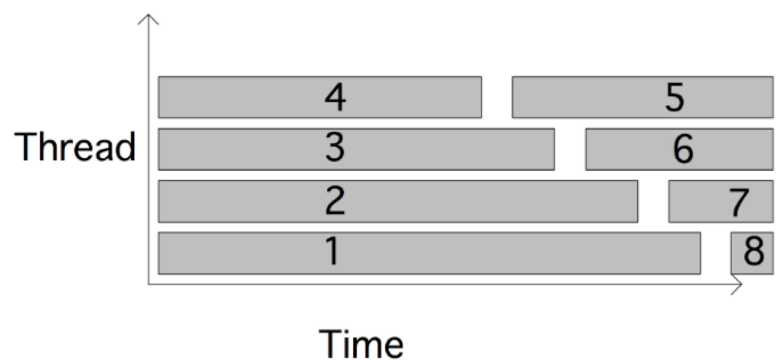
OpenMP Fundamentals

Loop Scheduling

Data Clauses	Comment
`static`	Each thread is assigned a fixed-size chunk (default)
`dynamic`	Work is assigned as a thread requests it
`guided`	Big chunks first and smaller and smaller chunks later
`runtime`	Use environment variable to control scheduling



Static



Dynamic

From TACC (https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html)

OpenMP Fundamentals

Sections

- One thread executes one section
  - ✓ If "too many", some threads execute more than one (round-robin)
  - ✓ If "too few" sections, some threads are idle
  - ✓ We don't know in advance which thread will execute which section

```
#pragma omp sections
{
  #pragma omp section
    { foo(); }
  #pragma omp section
    { bar(); }
  #pragma omp section
    { beer(); }
} // end of sections
```

OpenMP Fundamentals

Single

- A "single" block is executed by one thread
  - ✓ Useful for initializing shared variables
  - ✓ We don't know exactly which thread will execute the block
  - ✓ Only one thread executes the "single" region; others bypass it

```
#pragma omp single
{
  a = 10;
}
#pragma omp for
{ for (i=0; i<N; i++)
  b[i] = a;
}
```

OpenMP Fundamentals

Critical

- One thread at a time
  - ✓ Note the difference between “single” and “critical”
  - ✓ ALL threads will execute the region eventually
  - ✓ Mutual exclusive

```
#pragma omp critical
{
//...some stuff
}
```

OpenMP Fundamentals

Reduction Operations

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(sum_local)
{
  sum_local = 0; #pragma omp for
  for (i=0; i<n; i++)
    sum_local += a[i];
  #pragma omp critical {
  // form per-thread local sum
    sum += sum_local; // form global sum }
}
```

A reduction variable accumulates a value that depends on all the iterations together, but is independent of the iteration order.

```
sum = 0;
#pragma omp parallel for shared(...) private(...) \
reduction(+:sum)
{
  for (i=0; i<n; i++)
    sum += a[i];
}
```

Reduction operations of +,*,-,& |, ^, &&, || are supported

OpenMP Fundamentals

Functions and Environment Variables

Resource Query Functions

- Max number of threads: `omp_get_max_threads()`
- Number of processors: `omp_get_num_procs()`
- Number of threads (inside a parallel region): `omp_get_num_threads()`
- Get thread ID: `omp_get_thread_num()`

Control the Number of Threads

- Parallel region: `#pragma omp parallel num_threads(integer)`
- Run-time function: `omp_set_num_threads()`
- Environment variable: `export OMP_NUM_THREADS=n`

PRIORITY

Environment Variables

- Loop scheduling policy: `OMP_SCHEDULE`
- Number of threads: `OMP_NUM_THREADS`

Data Dependencies

Relationship Between Iterations of a Loop

- Not all loops can be parallelized.
- Parallelization of code must not affect the correctness of a program!
- Before adding OpenMP directives need to check for any dependencies:
  - ✓ Flow dependencies occur when an iteration depends on the result of a previous iteration.

```
# pragma omp parallel for num_threads(thread_count)
for (i = 2; i < n; i++)
  fibo[i] = fibo[i-1] + fibo[i-2];
```

  - ✓ Anti-dependencies occur when an iteration requires a value that is later updated.

```
# pragma omp parallel for num_threads(thread_count)
for (i = 1; i < n; i++)
  fibo[i] = fibo[i+1] + fibo[i+2];
```

Can be solved!

Data Dependencies

Relationship Between Iterations of a Loop

Bold= private

NO DEPENDENCY

```
        DO 10 I = 1,N
10    A(I) = X + B(I)*C(I)
```

PARALLEL

DATA DEPENDENCY

```
        DO 20 I = 2,N
20    A(I) = B(I) - A(I-1)
```

SEQUENTIAL

NO DEPENDENCY

```
        DO 20 I = 2,N,2
20     A(I) = B(I) - A(I-1)
```

PARALLEL

VARIABLE LOCAL

```
DO I = 1, N
   X = A(I)*A(I) + B(I)
   B(I) = X + B(I)*X
END DO
```

PARALLEL

FUNCTION CALL

```
 DO 10 I = 1, N
       X = SQRT(A(I))
       B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

FUNCTION DEPENDENT

NO DEPENDENCY

```
INDX = 0
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) + C(INDX)
END DO
```

RESTRUCTURE

Automatic Parallelization

A Parallel Version in Seconds!

- Vision: Take a sequential program and automatically convert it into a parallel version
  - ✓ Lots of research in the early 1990s, then tapered off. (it's hard!)
  - ✓ Renewed interest now since multicores are so common. (it's still hard!)

- Some languages are easier than others (FORTRAN!). C can be easy to parallelize, given the right code (avoid dynamic data), plus compiler hints

- "The right code" = Arrays with no loop-carried dependencies.

- Under the hood, most parallelization frameworks use OpenMP

Automatic Parallelization

Conditions for Automatic Parallelization

A Loop must

- have a recognized loop style, e.g., for loops with bounds that don't vary per-iteration
- have no dependencies between data accessed in loop bodies for each iteration
- not conditionally change scalar variables read after the loop terminates, or change any scalar variable across iterations
- have enough work in the loop body to make parallelization profitable

Automatic Parallelization

Automatic Parallelization in gcc

`gcc` (since 4.3) can also auto-parallelize loops, with several limitations:

1. It does not tell which loops it parallelizes
2. It only operates with a fixed number of threads
3. The profitability metrics are quite simple
4. Only operates in simple cases

Relevant flags

`-ftree-parallelize-loops=N` to parallelize where `N` is the number of threads

`-fdump-tree-parloops-details` shows the automatic parallelization (quite unreadable)

Automatic Parallelization

Some Examples

Loops That gcc's Automatic Parallelization Can Handle

Single Loop

```
for (i=0; i<1000; i++)
  x[i]=i+3;
```

Nested loops with simple dependency

```
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    X[i][j] = X[i][j] +Y[i-1][j];
```

Single loop with not-very-simple dependency

```
for (i=0; i<10; i++)
  X[2*i+1] =X[2*i];
```

Loops That gcc's Automatic Parallelization Can't Handle

Single loop with if statement

```
for (j = 0; j <= 10; j++)
  if (j>5)X[i]=i+3;
```

Triangle loop

```
for (i=0; i<100; i++)
  for (j = i; j < 100; j++)
    X[i][j] = 5
```

Parallelization Process

Continuous Process

1. Use Optimized Sequential Version (baseline execution time and results for validation)

2. Apply Automatic Parallelization

3. Evaluate execution time and speedup for a growing number of processors with a fixed and a growing problem size

4. Explicit Parallelization Using Directives (use info from automatic parallelization)

   Start with the loops with high CPU usage (profiling tools)

   Verify results for different number of processors (race conditions), and evaluate execution time and speedup for a growing number of processors with a fixed and a growing problem size

   Consider the sched type

   Repeat until results are good enough in terms of time and/or speedup

5. Explicit Parallelization Adapting Code

   ☺ Restructure loops to enhance parallelism and eliminate data dependencies
   😐 Change the numerical algorithm

5. Explicit Parallelization adopting a coarser-grain domain decomposition approach

Next Steps

- Get ready for tomorrow's **lab sessions**
  I6 - OpenMP on AWS

- Get ready for second **hands-on:**
  H2. OpenMP Programming (on Cannon)
  <u>Check your Cannon account</u>