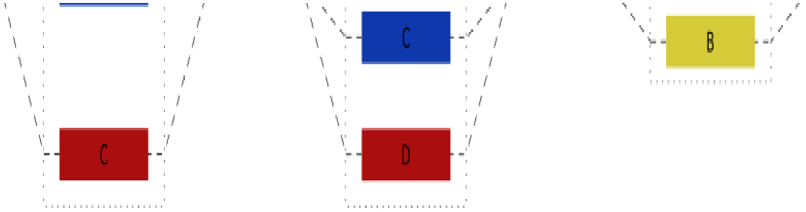
CS 470
Spring 2018

Mike Lam, Professor



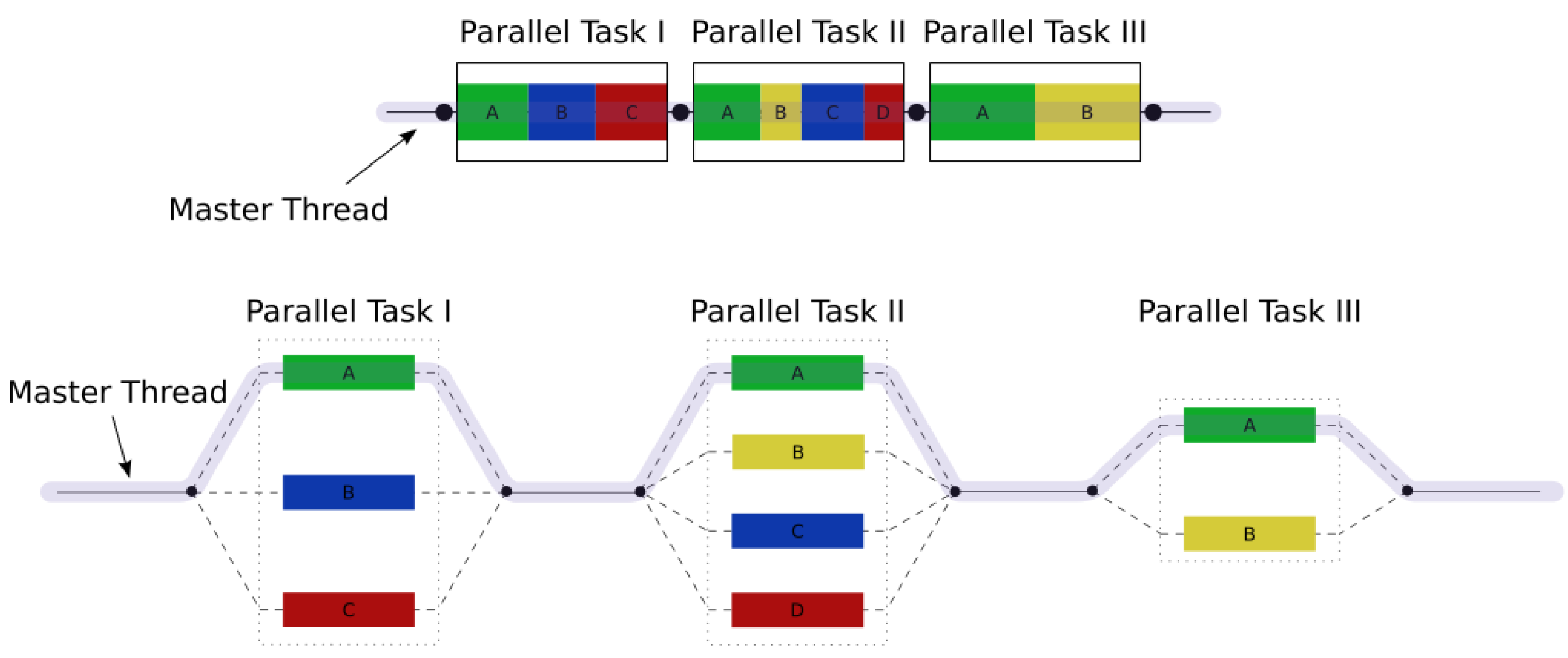
OpenMP

OpenMP

- Programming language extension
  - Compiler support required
  - "**Open M**ulti-**P**rocessing" (open standard; latest version is 4.5)
- "Automatic" thread-level parallelism
  - Guided by programmer-supplied directives
  - Does NOT verify correctness of parallel transformations
  - Targets shared-memory systems
  - Used in distributed systems for on-node parallelism
- Other similar techs: Cilk, OpenACC
  - OpenMP is currently the most popular CPU-based technology

Fork-join threading

- OpenMP provides directives to control threading
  - General fork-join threading model w/ teams of threads
  - One master thread and multiple worker threads



Source: `https://en.wikipedia.org/wiki/Fork-join_model`

C preprocessor

- Text-based processing phase of compilation
  - Can be run individually with "`cpp`"
- Controlled by directives on lines beginning with "`#`"
  - Must be the first non-whitespace character
  - Alignment is a matter of personal style

```
#include <stdio.h>
#define FOO
#define BAR 5

int main() {
#   ifdef FOO
    printf("Hello!\n");
#   else
    printf("Goodbye!\n");
#   endif
    printf("%d\n", BAR);
    return 0;
}
```

my preference

```
#include <stdio.h>
#define FOO
#define BAR 5

int main() {
    #ifdef FOO
    printf("Hello!\n");
    #else
    printf("Goodbye!\n");
    #endif
    printf("%d\n", BAR);
    return 0;
}
```

Pragmas

- `#pragma` - generic preprocessor directive
  - Provides direction or info to later compiler phases
  - Ignored by compilers that don't support it
  - All OpenMP pragma directives begin with "`omp`"
  - Basic threading directive: "`parallel`"
    - Runs the following code construct in fork/join parallel threads
    - Implicit barrier at end of construct

```
#pragma play(global_thermonuclear_war)
do_something();

#pragma omp parallel
do_something_else();
```

Compiling and running w/ OpenMP

- Must `#include <omp.h>`
- Must compile with "`-fopenmp`" flag

```
gcc -g -std=c99 -Wall -fopenmp -o omp omp.c
./omp
```

- Use `OMP_NUM_THREADS` environment variable to set thread count
  - Default value is core count (w/ hyper-threads)

```
OMP_NUM_THREADS=4 ./omp
```

"Hello World" example

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

int main(int argc, char *argv[])
{

#   pragma omp parallel
    printf("Hello!\n");

    printf("Goodbye!\n");

    return EXIT_SUCCESS;
}
```

Pragma scope

- Most OpenMP pragmas apply to the immediately-following statement or block
  - Not necessarily just the next line!

```
#   pragma omp parallel
    printf("hello!\n");
```

```
#   pragma omp parallel
    total += a * b + c;
```

```
#   pragma omp parallel
    {
        int a = 0;
        ...
        global_var += a;
    }
```

```
#   pragma omp parallel
    for (i = 0; i < n; i++) {
        sum += i;
    }
```

Clauses

- Directives can be modified by clauses
  - Text that follows the directive
  - Some clauses take parameters
  - E.g., "`num_threads`"

```
# pragma omp parallel num_threads(thread_count)
```

Functions

- Built-in functions:
  - **`omp_get_num_threads()`**
    - Returns the number of threads in the current team
    - Similar to `MPI_Comm_size`
  - **`omp_get_max_threads()`**
    - Returns the maximum number of threads in a team
    - Can be used outside a parallel region
  - **`omp_get_thread_num()`**
    - Returns the caller's thread ID within the current team
    - Similar to `MPI_Comm_rank`
  - **`omp_get_wtime()`**
    - Returns the elapsed wall time in seconds
    - Similar to `MPI_Wtime`

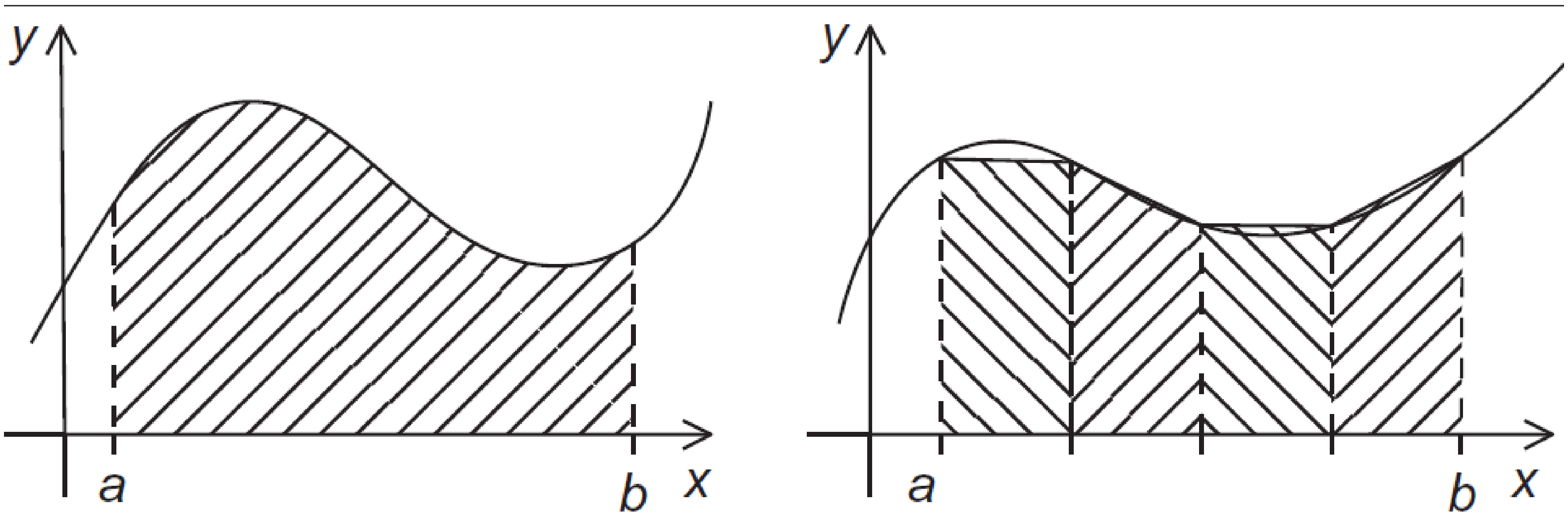
Incremental parallelization

- Pragmas allow incremental parallelization
  - Gradually add parallel constructs
  - OpenMP programs should be correct serial programs when compiled without "`-fopenmp`"
    - Pragma directives are ignored
  - Use "`_OPENMP`" preprocessor variable to test
    - If defined, it is safe to call OpenMP functions

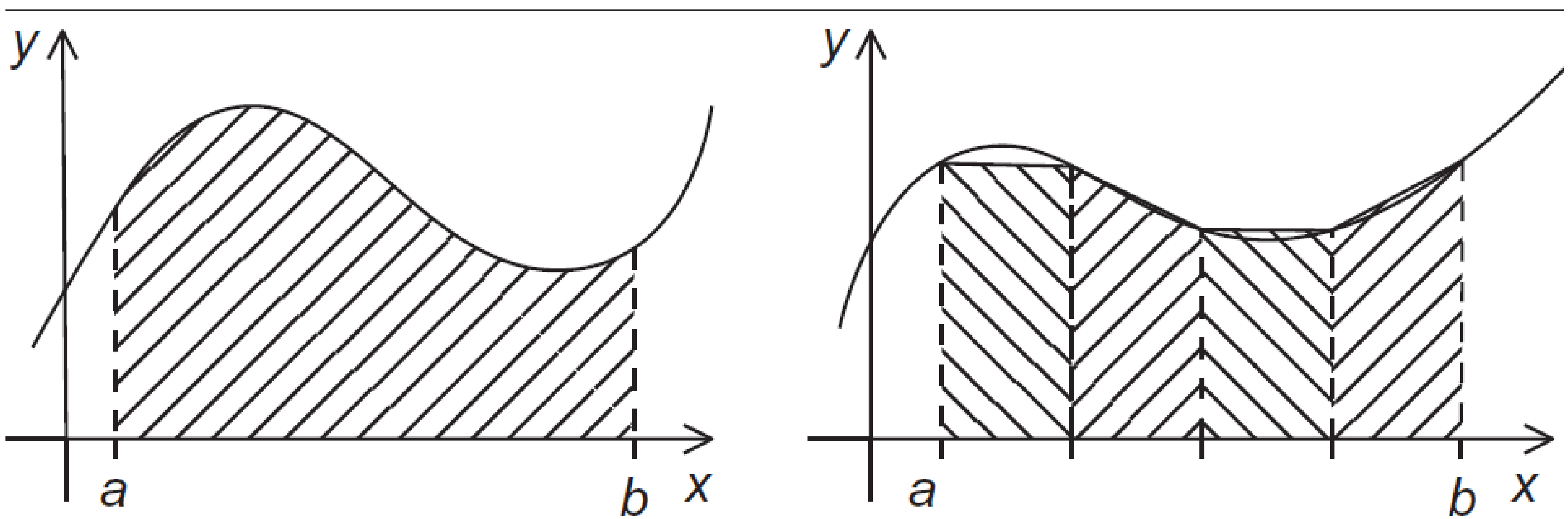
```
#ifdef _OPENMP
#include <omp.h>
#endif
```

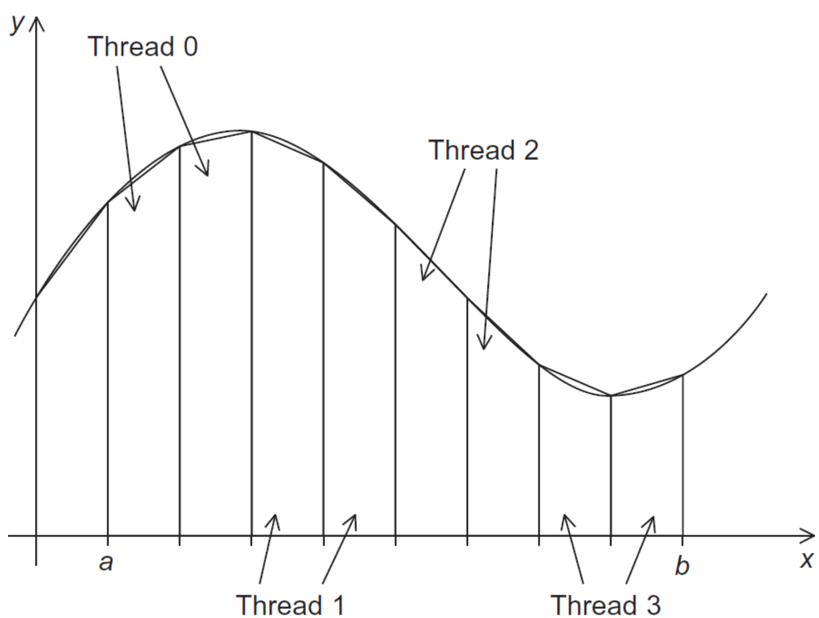
```
#   ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
#   else
    int my_rank = 0;
    int thread_count = 1;
#   endif
```

Trapezoid example (from textbook)



Trapezoid example (from textbook)





Is this task or data parallelism?

What problem(s) might we run into?

Mutual exclusion

- Use "`critical`" directive to enforce mutual exclusion
  - Only one thread at a time can execute the following construct
  - A critical section can optionally be named
    - Sections that share a name share exclusivity
    - *CAUTION: all unnamed sections “share” a name!*

```
#   pragma omp critical(gres)
    global_result += my_result ;
```

Barriers

- Explicit barrier: "`barrier`" directive
  - All threads must sync

```
#   pragma omp barrier
```

Single-thread regions

- Implicit barrier: "`single`" directive
  - Only one thread executes the following construct
    - Could be any thread; don't assume it's the master
    - For master-thread-only, use "`master`" directive
  - All threads must sync at end of directive
    - Use "`nowait`" clause to prevent this implicit barrier

```
#   pragma omp single
    global_result /= 2;

#   pragma omp single nowait
    global_iter_count++;
```

Scope of variables

- In OpenMP, each variable has a thread "scope"
  - Shared scope: accessible by all threads in team
    - Default for variables declared **before** a parallel block
  - Private scope: accessible by only a single thread
    - Default for variables declared **inside** a parallel block

```
    double foo = 0.0;
#   pragma omp parallel
    {
        double bar = do_calc() * PI;
#       pragma omp critical
        foo = foo + bar/2.0;
    }
```

Default scoping

- The "`default`" clause changes the default scope for variables declared outside the parallel block
  - **`default(none)`** mandates explicit scope declaration
    - Use "`shared`" and "`private`" clauses
    - Compiler will check that you declared all variables
    - This is good programming practice!

```
   double sum = 0.0;
#  pragma omp parallel for num_threads(thread_count) \
      default(none) reduction(+:sum) private(k, factor) \
      shared(n)
   for (k = 0; k < n; k++) {
      if (k % 2 == 0)
         factor = 1.0;
      else
         factor = -1.0;
      sum += factor/(2*k+1);
   }
```

Reductions

- The `reduction(op:var)` clause applies an operator to a sequence of operands to get a single result
  - Similar to `MPI_Reduce`, but not distributed
  - In OpenMP, uses a shared-memory reduction variable (`var`)
  - All intermediate/final values are stored in the reduction variable
  - OpenMP handles synchronization (implicit mutex)
  - Supported operations (`op`): `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `min`, `max`

```
    double foo = 0.0;
#   pragma omp parallel reduction(+:foo)
    foo += (do_calc() * PI)/2.0;
```

Parallel for loops

- The "`parallel for`" directive parallelizes a loop
  - Probably the most powerful and most-used directive
  - Divides loop iterations among a team of threads
  - **CAVEAT**: the for-loop must have a very particular form

$$\textbf{for}\left(\begin{array}{lclcl}
 & & & & \texttt{index++} \\
 & & & & \texttt{++index} \\
 & & \texttt{index < end} & & \texttt{index--} \\
 & & \texttt{index <= end} & & \texttt{--index} \\
\texttt{index = start} & ; & \texttt{index >= end} & ; & \texttt{index += incr} \\
 & & \texttt{index > end} & & \texttt{index -= incr} \\
 & & & & \texttt{index = index + incr} \\
 & & & & \texttt{index = incr + index} \\
 & & & & \texttt{index = index - incr}
\end{array}\right)$$

Parallel for loops

- The compiler must be able to determine the number of iterations *prior to the execution of the loop*
- Implications/restrictions:
  - The number of iterations must be finite (no "**`for (;;)`**")
  - The **`break`** statement cannot be used (although **`exit()`** is ok)
  - The **`index`** variable must have an integer or pointer type
  - The **`index`** variable must only be modified by the "increment" part of the loop declaration
  - The **`index`**, **`start`**, **`end`**, and **`incr`** expressions/variables must all have compatible types
  - The **`start`**, **`end`**, and **`incr`** expressions must not change during execution of the loop

Issue: correctness

```
fib[0] = fib[1] = 1;
for (i = 2; i < n; i++)
     fib[i] = fib[i-1] + fib[i-2];
```

2 threads

```
     fib[0] = fib[1] = 1;
#    pragma omp parallel for num_threads(2)
     for (i = 2; i < n; i++)
          fib[i] = fib[i-1] + fib[i-2];
```

`1 1 2 3 5 8 13 21 34 55`

this is correct

but sometimes we get this

`1 1 2 3 5 8 0 0 0 0`

Loop dependencies

- A loop has a data dependence if one iteration depends on another iteration
  - Explicitly (as in Fibonacci example) or implicitly
  - Includes side effects!
  - Sometimes called loop-carried dependence
- A loop with dependencies cannot (usually) be parallelized correctly by OpenMP
  - Identifying dependencies is very important!
  - OpenMP does not check for them

Loop dependencies

- Examples:

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

```
for (i = 1; i < n; i++) {
    a[i] += a[i-1]
}
```

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

```
for (i = 1; i < n; i += 2) {
    a[i] += a[i-1]
}
```

```
for (i = 0; i < n; i++) {
    a[i] += a[i]
}
```

```
for (i = 1; i < n; i++) {
    a[i] += b[i-1]
}
```

Loop dependencies

- Examples:

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

OK!

```
for (i = 1; i < n; i++) {
    a[i] += a[i-1]
}
```

BAD! (iteration i depends on i-1)

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

OK!

```
for (i = 1; i < n; i += 2) {
    a[i] += a[i-1]
}
```

OK!

```
for (i = 0; i < n; i++) {
    a[i] += a[i]
}
```

OK!

```
for (i = 1; i < n; i++) {
    a[i] += b[i-1]
}
```

OK!