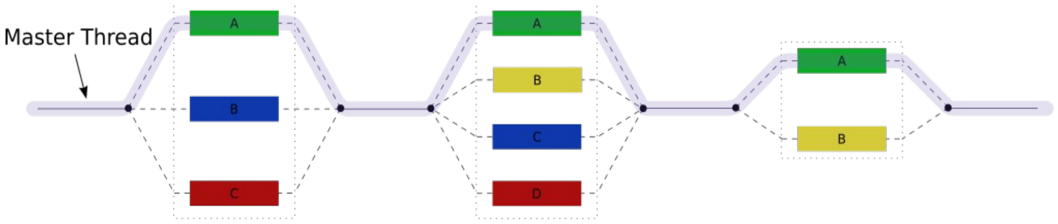
CS 470
Spring 2019

Mike Lam, Professor



OpenMP

OpenMP

- Programming language extension
  - Compiler support required
  - "**Open M**ulti-**P**rocessing"
  - Open standard: latest version is 4.5
  - Managed by a consortium: `openmp.org`

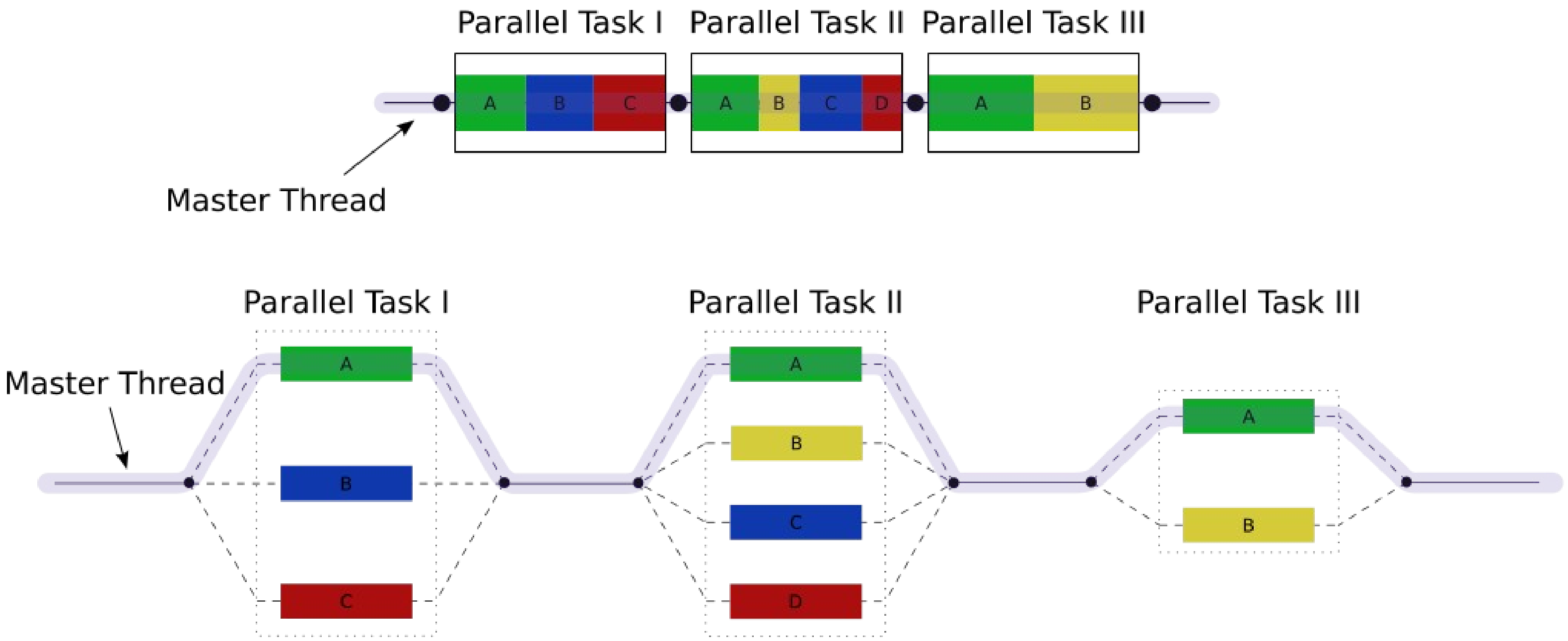
- “Automatic” thread-level parallelism
  - Guided by programmer-supplied directives (pragmas)
  - Does NOT verify correctness of parallel transformations
  - Targets shared-memory systems
  - Used in distributed systems for on-node parallelism

Technology comparison

- Cilk / Cilk Plus
  - Language extension - new keywords: `spawn`, `sync`, `cilk_for`
  - Purchased by Intel in 2009; losing steam now
- Intel Thread Building Blocks (TBB)
  - Template library (C++ only)
  - Gaining popularity, but fairly complicated to use
- OpenMP
  - Directive-based; supported by most major compilers
  - Currently the most popular CPU-based technology
- OpenACC
  - Directive-based; similar to OpenMP
  - Primarily aimed at GPU parallelism (driven by NVIDIA)

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)
```

WARNING: Only use the "`num_threads`" clause if you wish to hard-code the number of threads (this is not considered best practice for OpenMP!)

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`

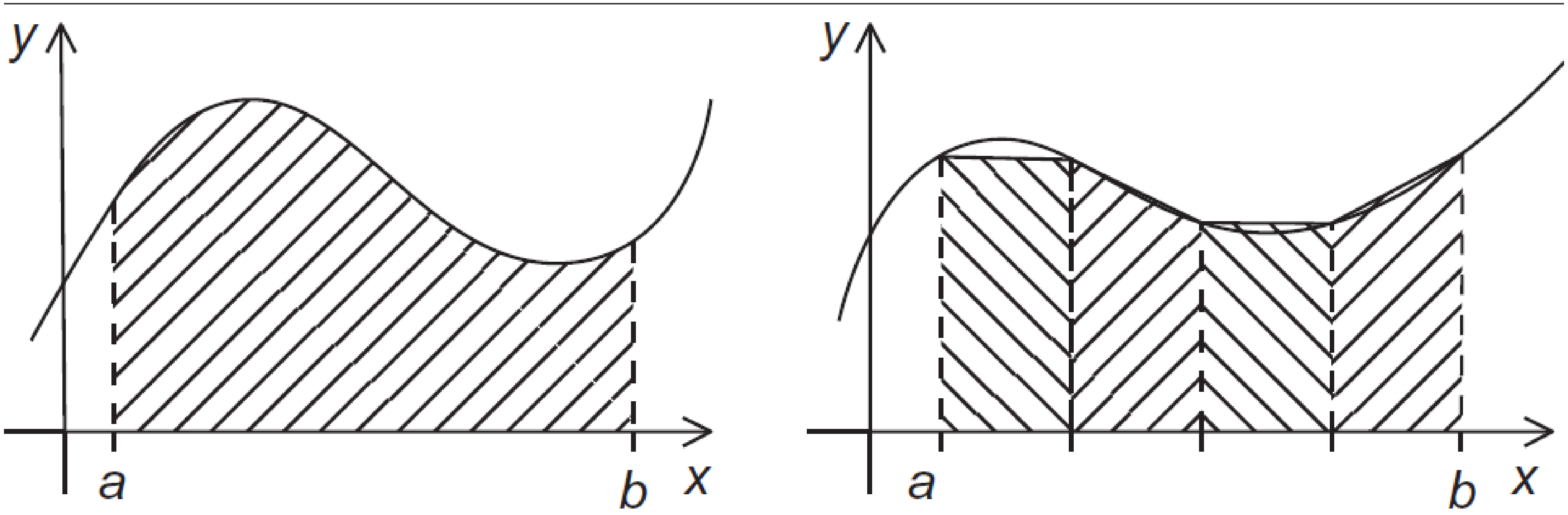
Incremental parallelization

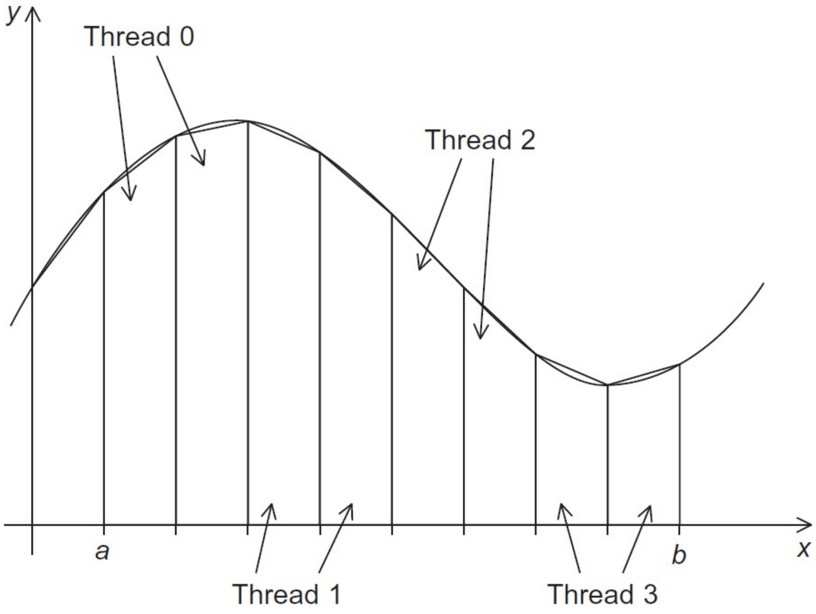
- Pragmas allow incremental parallelization
  - Gradually add parallel constructs
  - OpenMP programs can be correct serial programs when compiled without "`-fopenmp`"
    - Pragma directives are ignored
    - Still need to guard the #include and function calls
  - Use "`_OPENMP`" preprocessor variable 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)





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++;
```

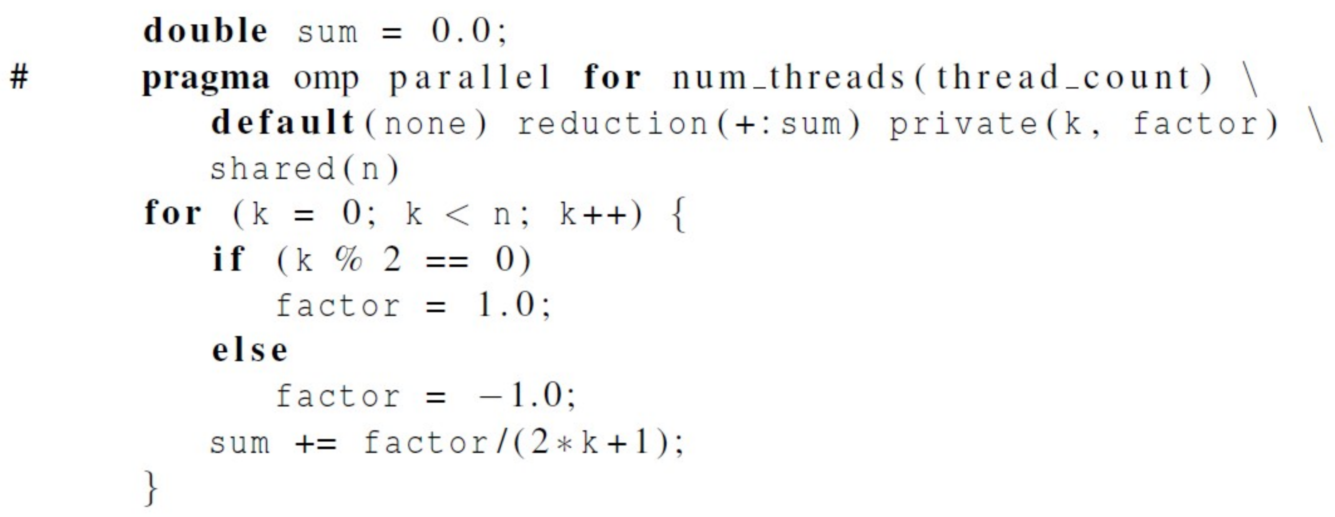
Scope of variables

- 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;                        // shared
#   pragma omp parallel
    {
        double bar = do_calc() * PI;         // private
#       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);
      }
```

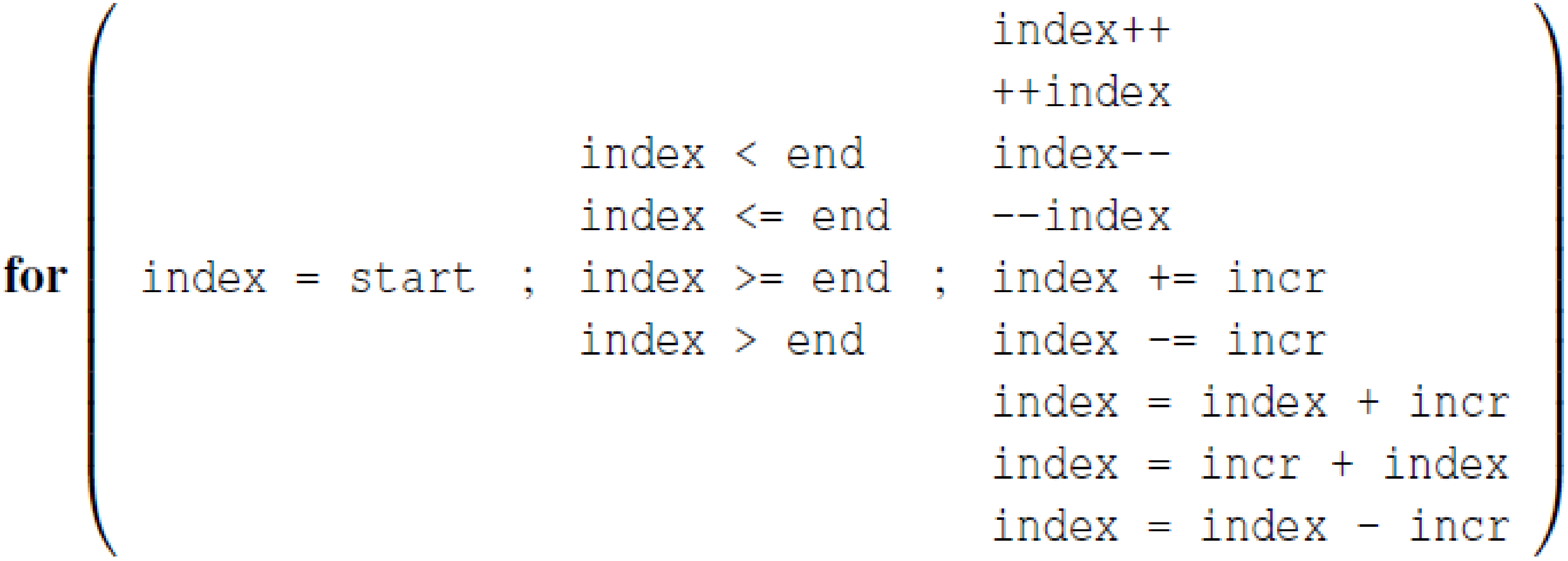
Reductions

- 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



```
                                                   index++
                                                   ++index
                          index <  end             index--
                          index <= end             --index
for ( index = start  ;    index >= end    ;        index += incr                )
                          index >  end             index -= incr
                                                   index = index + incr
                                                   index = incr + index
                                                   index = index - incr
```

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];
}
```

OK!

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

OK!

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

OK!

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

BAD! (iteration i depends on i-1)

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

OK!

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

OK!

Atomics

- OpenMP provides access to highly-efficient hardware synchronization mechanisms
  - Use the `atomic` pragma to annotate a single statement
  - Statement must be a single increment/decrement or in the following form:
    - `x <op>= <expr>;    // <op> can be +, -, *, /, &, |, ^, <<, >>`
  - Many ISAs provide an atomic load/modify/store instruction
    - In x86-64, specified using the LOCK prefix
    - Far more efficient than using a mutex (i.e., `critical`)
      - This requires multiple function calls!

Locks

- OpenMP provides a basic locking system
  - Useful for protecting a data structure rather than a region of code
  - `omp_lock_t`: lock variable
    - Similar to `pthread_mutex_t`
  - `omp_lock_init`: initialize lock
    - Similar to `pthread_mutex_init`
  - `omp_set_lock`: acquire lock
    - Similar to `pthread_mutex_lock`
  - `omp_unset_lock`: release lock
    - Similar to `pthread_mutex_unlock`
  - `omp_lock_destroy`: clean up a lock
    - Similar to `pthread_mutex_destroy`

Thread safety

- Don't **mix** mutual exclusion mechanisms
  - `#pragma omp critical`
  - `#pragma omp atomic`
  - `omp_set_lock()`
- Don't **nest** mutual exclusion mechanisms
  - Nesting unnamed `critical` sections *guarantees* deadlock!
    - The thread cannot enter the second section because it is still in the first section, and unnamed sections “share” a name
  - If you must, use named critical sections or nested locks

Nested locks

- Simple vs. nested locks
  - `omp_nest_lock_*` instead of `omp_lock_*`
  - A nested lock may be acquired multiple times
    - Must be in the same thread
    - Must be released the same number of times
    - Allows you to write functions that call each other but need to acquire the same lock

Sections

- OpenMP is most often used for data parallelism (`parallel for`)
- However, it also supports explicit task parallelism
- Pre-OpenMP 3.0 mechanism: `sections` directive
  - Contains multiple `section` blocks; each section runs on separate thread
  - Must list all sections in same location (cannot dynamically add new tasks)
  - Implicit barrier at end (unless `nowait` clause is specified)

```
#   pragma omp parallel sections
    {
#       pragma omp section
        producer();
#       pragma omp section
        consumer();
    }
```

Tasks

- Post-OpenMP 3.0 mechanism: `task` directive
  - Similar to thread pool task model
  - Tasks are assigned to available worker threads by the runtime
    - Tasks may be deferred if no workers available
  - No implicit barrier; use `taskwait` directive if needed
  - Use `single` region if only one thread should begin (e.g., recursion)
    - Use `nowait` clause to allow other threads to run tasks

```
main:
#   pragma omp parallel
#   pragma omp single nowait
    quick_sort(items, n);
```

```
quicksort:
    <select pivot and partition>

    // recursively sort each partition
#   pragma omp task
    quick_sort(items, p+1);
#   pragma omp task
    quick_sort(items+q, n-q);
#   pragma omp taskwait
```

Loop scheduling

- Use the `schedule` clause to control how parallel for-loop iterations are allocated to threads
  - Modified by chunksize parameter
  - static: split into chunks before loop is executed
  - dynamic: split into chunks, dynamically allocated to threads (similar to thread pool or tasks)
  - guided: like dynamic, but chunk sizes decrease
    - The specified chunksize is the minimum
  - auto: allows the compiler or runtime to choose
  - runtime: allows specification using `OMP_SCHEDULE`

Loop scheduling

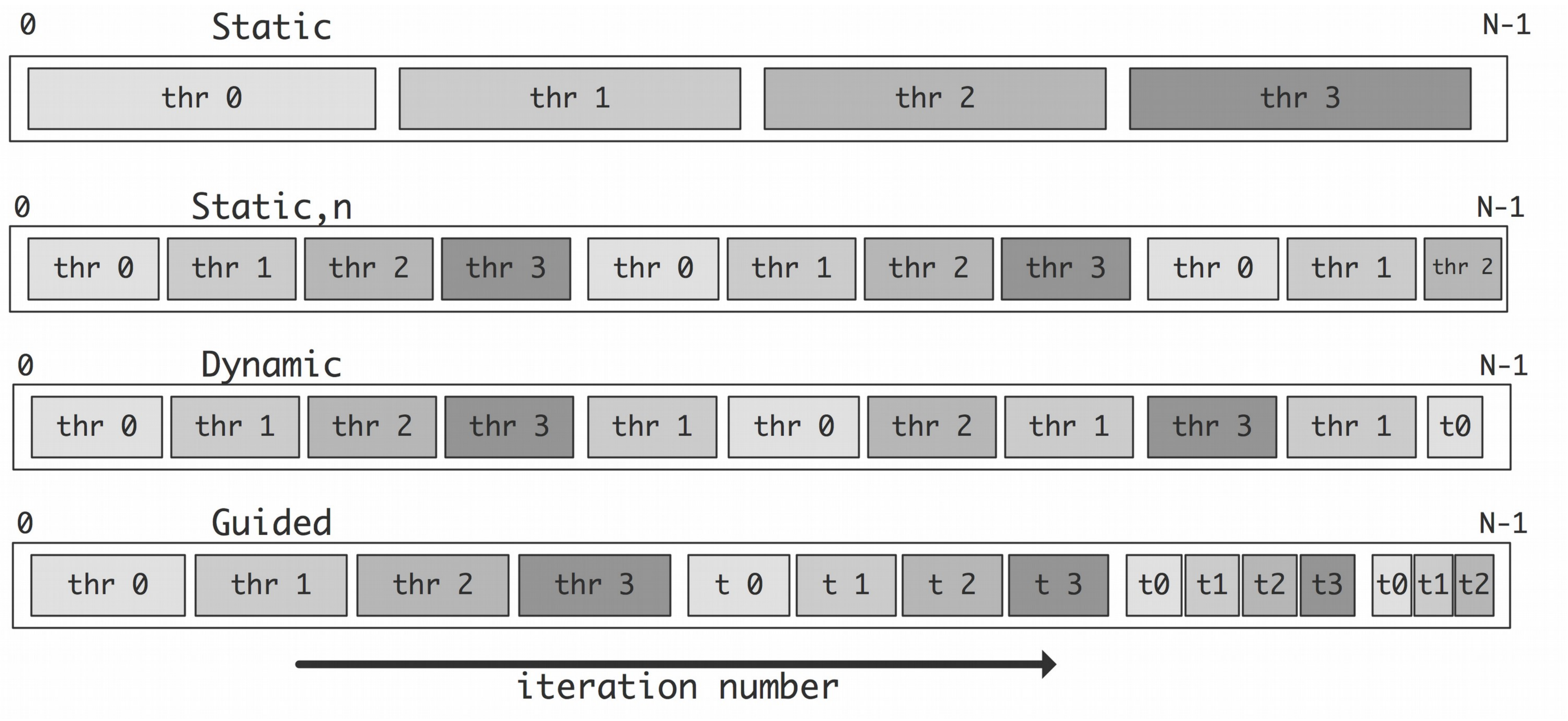


Image from `http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html`

Loop scheduling

```
(static)                    (static, 1)                 (static, 2)                 (dynamic, 2)                (guided)

Iteration 00 on thread 0    Iteration 00 on thread 0    Iteration 00 on thread 0    Iteration 00 on thread 1    Iteration 00 on thread 2
Iteration 01 on thread 0    Iteration 01 on thread 1    Iteration 01 on thread 0    Iteration 01 on thread 1    Iteration 01 on thread 2
Iteration 02 on thread 0    Iteration 02 on thread 2    Iteration 02 on thread 1    Iteration 02 on thread 3    Iteration 02 on thread 2
Iteration 03 on thread 0    Iteration 03 on thread 3    Iteration 03 on thread 1    Iteration 03 on thread 3    Iteration 03 on thread 2
Iteration 04 on thread 0    Iteration 04 on thread 0    Iteration 04 on thread 2    Iteration 04 on thread 2    Iteration 04 on thread 2
Iteration 05 on thread 0    Iteration 05 on thread 1    Iteration 05 on thread 2    Iteration 05 on thread 2    Iteration 05 on thread 2
Iteration 06 on thread 0    Iteration 06 on thread 2    Iteration 06 on thread 3    Iteration 06 on thread 0    Iteration 06 on thread 2
Iteration 07 on thread 0    Iteration 07 on thread 3    Iteration 07 on thread 3    Iteration 07 on thread 0    Iteration 07 on thread 2
Iteration 08 on thread 1    Iteration 08 on thread 0    Iteration 08 on thread 0    Iteration 08 on thread 3    Iteration 08 on thread 0
Iteration 09 on thread 1    Iteration 09 on thread 1    Iteration 09 on thread 0    Iteration 09 on thread 3    Iteration 09 on thread 0
Iteration 10 on thread 1    Iteration 10 on thread 2    Iteration 10 on thread 1    Iteration 10 on thread 3    Iteration 10 on thread 0
Iteration 11 on thread 1    Iteration 11 on thread 3    Iteration 11 on thread 1    Iteration 11 on thread 3    Iteration 11 on thread 0
Iteration 12 on thread 1    Iteration 12 on thread 0    Iteration 12 on thread 2    Iteration 12 on thread 3    Iteration 12 on thread 0
Iteration 13 on thread 1    Iteration 13 on thread 1    Iteration 13 on thread 2    Iteration 13 on thread 3    Iteration 13 on thread 0
Iteration 14 on thread 1    Iteration 14 on thread 2    Iteration 14 on thread 3    Iteration 14 on thread 3    Iteration 14 on thread 1
Iteration 15 on thread 1    Iteration 15 on thread 3    Iteration 15 on thread 3    Iteration 15 on thread 3    Iteration 15 on thread 1
Iteration 16 on thread 2    Iteration 16 on thread 0    Iteration 16 on thread 0    Iteration 16 on thread 2    Iteration 16 on thread 1
Iteration 17 on thread 2    Iteration 17 on thread 1    Iteration 17 on thread 0    Iteration 17 on thread 2    Iteration 17 on thread 1
Iteration 18 on thread 2    Iteration 18 on thread 2    Iteration 18 on thread 1    Iteration 18 on thread 3    Iteration 18 on thread 1
Iteration 19 on thread 2    Iteration 19 on thread 3    Iteration 19 on thread 1    Iteration 19 on thread 3    Iteration 19 on thread 3
Iteration 20 on thread 2    Iteration 20 on thread 0    Iteration 20 on thread 2    Iteration 20 on thread 2    Iteration 20 on thread 3
Iteration 21 on thread 2    Iteration 21 on thread 1    Iteration 21 on thread 2    Iteration 21 on thread 2    Iteration 21 on thread 3
Iteration 22 on thread 2    Iteration 22 on thread 2    Iteration 22 on thread 3    Iteration 22 on thread 1    Iteration 22 on thread 3
Iteration 23 on thread 2    Iteration 23 on thread 3    Iteration 23 on thread 3    Iteration 23 on thread 1    Iteration 23 on thread 2
Iteration 24 on thread 3    Iteration 24 on thread 0    Iteration 24 on thread 0    Iteration 24 on thread 3    Iteration 24 on thread 2
Iteration 25 on thread 3    Iteration 25 on thread 1    Iteration 25 on thread 0    Iteration 25 on thread 3    Iteration 25 on thread 2
Iteration 26 on thread 3    Iteration 26 on thread 2    Iteration 26 on thread 1    Iteration 26 on thread 1    Iteration 26 on thread 2
Iteration 27 on thread 3    Iteration 27 on thread 3    Iteration 27 on thread 1    Iteration 27 on thread 1    Iteration 27 on thread 2
Iteration 28 on thread 3    Iteration 28 on thread 0    Iteration 28 on thread 2    Iteration 28 on thread 1    Iteration 28 on thread 2
Iteration 29 on thread 3    Iteration 29 on thread 1    Iteration 29 on thread 2    Iteration 29 on thread 1    Iteration 29 on thread 1
Iteration 30 on thread 3    Iteration 30 on thread 2    Iteration 30 on thread 3    Iteration 30 on thread 0    Iteration 30 on thread 1
Iteration 31 on thread 3    Iteration 31 on thread 3    Iteration 31 on thread 3    Iteration 31 on thread 0    Iteration 31 on thread 3
```

Parallel regions

- Often useful: multiple for-loops inside a `parallel` region
  - Many pragmas bind dynamically to any active `parallel` region
  - Less thread creation/joining overhead
  - Private variables can be re-used across multiple loops

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


    do_something_serial();

#   pragma omp parallel for
    for (int j = 0; j < m; j++) {
        do_something_else_parallel();
    }
```

Original

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

#       pragma omp single
        do_something_serial();

#       pragma omp for
        for (int j = 0; j < m; j++) {
            do_something_else_parallel();
        }
    }
```

Faster

Nested loops

- The parallel for loop only applies to the loop layer that you specify
  - For nested loops, use the `collapse` clause to combine iteration spaces
  - Spaces must be “square”
    - i.e., inner loop iteration count should not depend on outer loop value

```
#pragma omp parallel for collapse(2)
for (i = 0; i < n; i++) {      // row
    for (j = 0; j < n; j++) {  // column
        a[i*n + j] = 1.0;
    }
}
```

Private variables

- Sometimes it is useful to have a variable that is neither completely shared nor completely private
- Use `firstprivate` to initialize with the value before parallel region
  - Useful if all threads need to start with the same value but later diverge
- Use `lastprivate` to save last value after parallel region

```
int i;
#pragma omp parallel
{
#   pragma omp for lastprivate(i)
    for (i = 0; i < n-1; i++)
        a[i] = b[i] + b[i+1];
}
a[i] = b[i];
```

More OpenMP examples

- Posted in `/shared/cs470`
  - For-loop scheduling (`omp-sched`)
  - Critical sections and deadlock (`omp-deadlock`)
  - The ‘atomic’ directive (`omp-atomic`)
  - Tasks (`omp-qsort`)
  - Matrix multiplication (`omp-matmult`)