CIS 431/531
Intro to Parallel Computing

OpenMP II

Quiz

1. (10 minutes)

1a. Draw a DAG for the given instruction stream

1b. Use register coloring/renaming to eliminate as many dependencies from the DAG as possible

i1. R1 <- 34

i2. R2 <- 56

i3. R1 <- R1 + R5

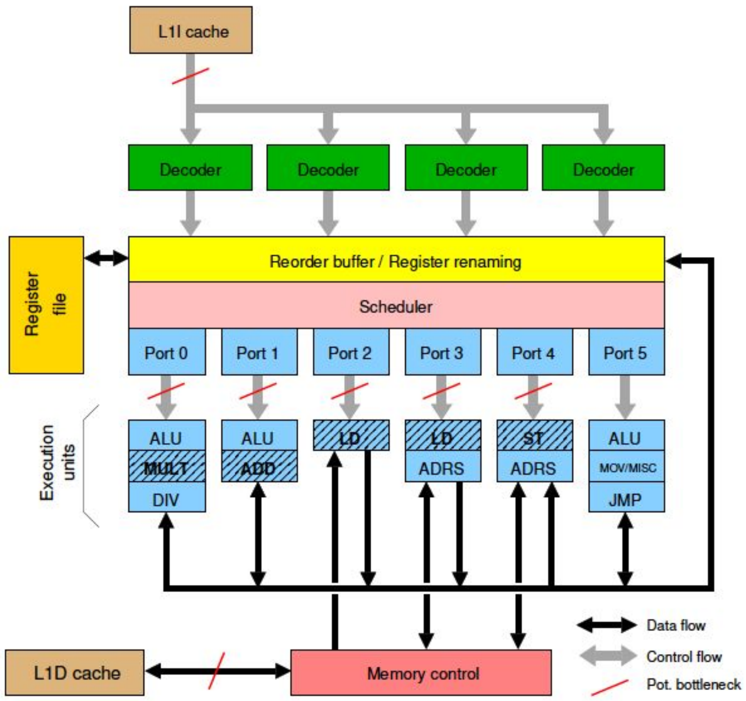
i4. R5 <- R1 - R2

i5. R5 <- R1 + R2

2. (5 minutes)

Given the following architecture, what are the peak and sustained throughputs (i.e., instructions per cycle)?

If they are different, explain why



Questions?

Previous Lecture

Fork-Join Model

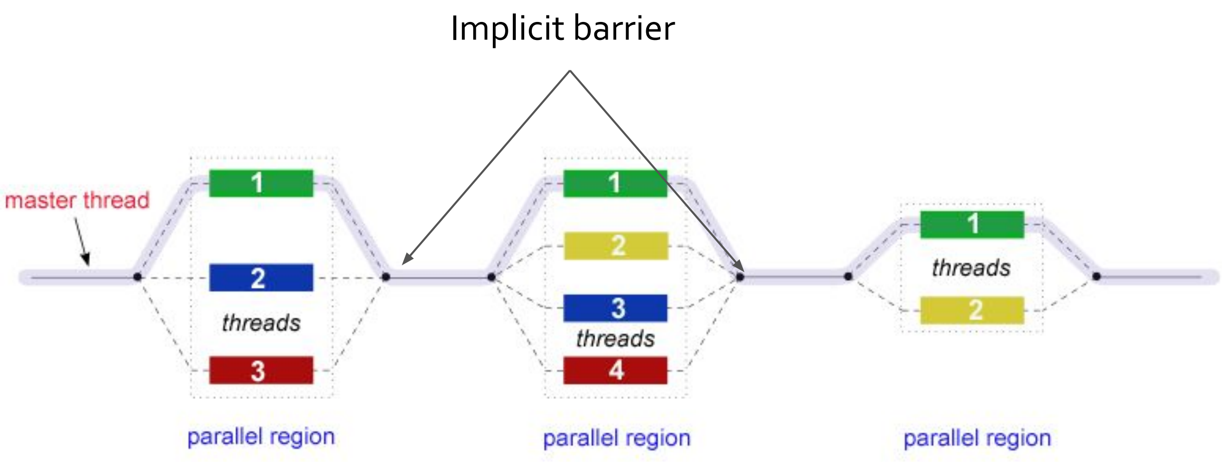
- OS fork()
- pthreads
- OpenMP
  - Work-sharing constructs *do/for*, *sections*, and *single*

This Lecture

OpenMP

- Tasks
- Synchronization
- Data scope
- Directives

OpenMP Model



OpenMP Syntax

Most OpenMP constructs are compiler directives

#pragma omp <directive> [clause ...]

```
#pragma omp parallel default(shared) private(a, b)
```

Library Functions

Thread queries (number of threads, thread ID, etc.)

```
int omp_get_num_threads(void)
```

Environment Variables

Setting number of threads, affinity, etc.

```
export OMP_NUM_THREADS=8
```

Work-Sharing Constructs

Divides the work in the code region **between** the threads (vs. all threads executing the entirety of the code region)

Types of work-sharing constructs

- Do/For
- Sections
- Single

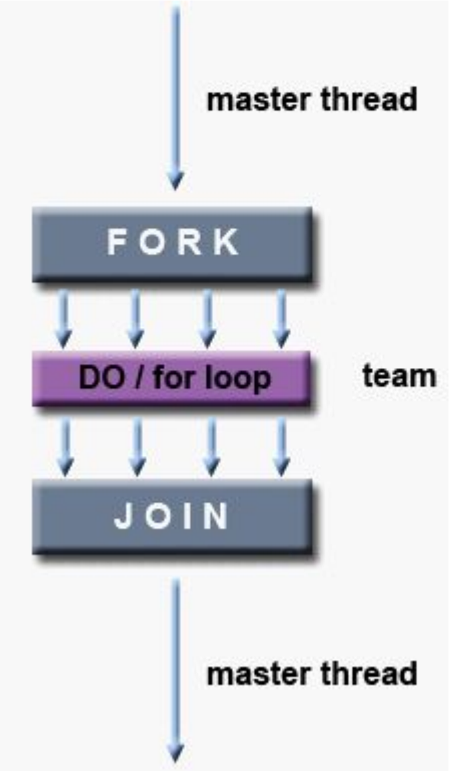
Work-Sharing - Do/For

Share iterations of the loop across the threads (i.e., data parallelism)

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < ARR_SIZE; i++) {
        x[i] = 1;
    }
}
```

OR

```
#pragma omp parallel for
for(int i = 0; i < ARR_SIZE; i++) {
        x[i]++;
}
```



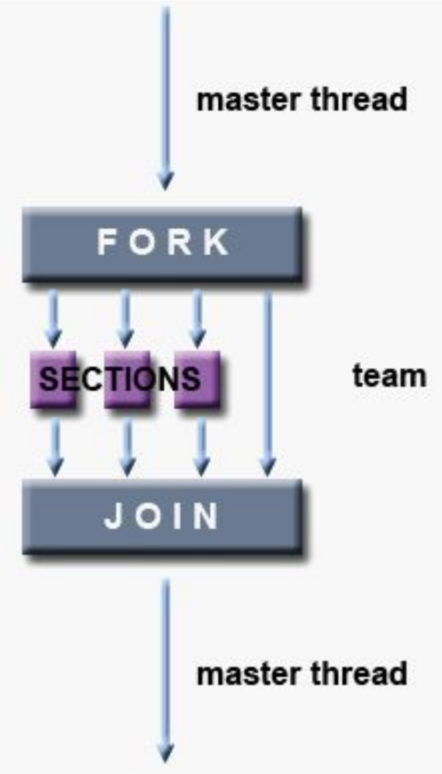
Also an implicit barrier at the end of the loop

Work-Sharing - Sections

Each section can do different parts of the code section (assuming they can be done independently) or completely different work altogether

```
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                for(int i = 0; i < ARR_SIZE/4; i++) {
                    x[i] = 1;
                }
            }
            …
            #pragma omp section
            {
                for(int i = (ARR_SIZE/4)*3; i <
ARR_SIZE; i++) {
                    x[i] = 1;
                }
            }
        }
    }
```

This code has a **similar** effect as using 4 threads with **parallel for**



Work-Sharing - Single

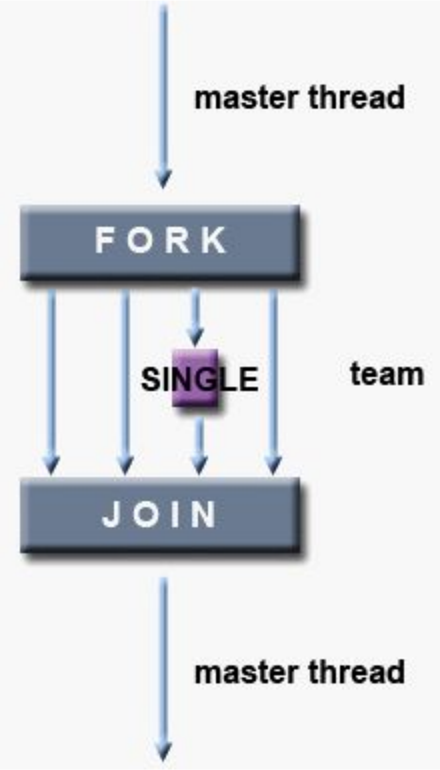
Only 1 thread in the team executes the code section

Why??

- Might be useful when executing code sections that are not thread safe (e.g., IO)

```
#pragma omp parallel
{
    #pragma omp single
    {
        some code…
    }
}
```

Only construct that does **not allow "parallel single"**



Questions?

OpenMP Tasks

So far we've seen **data-parallel** computation

SIMD

OpenMP do-for construct

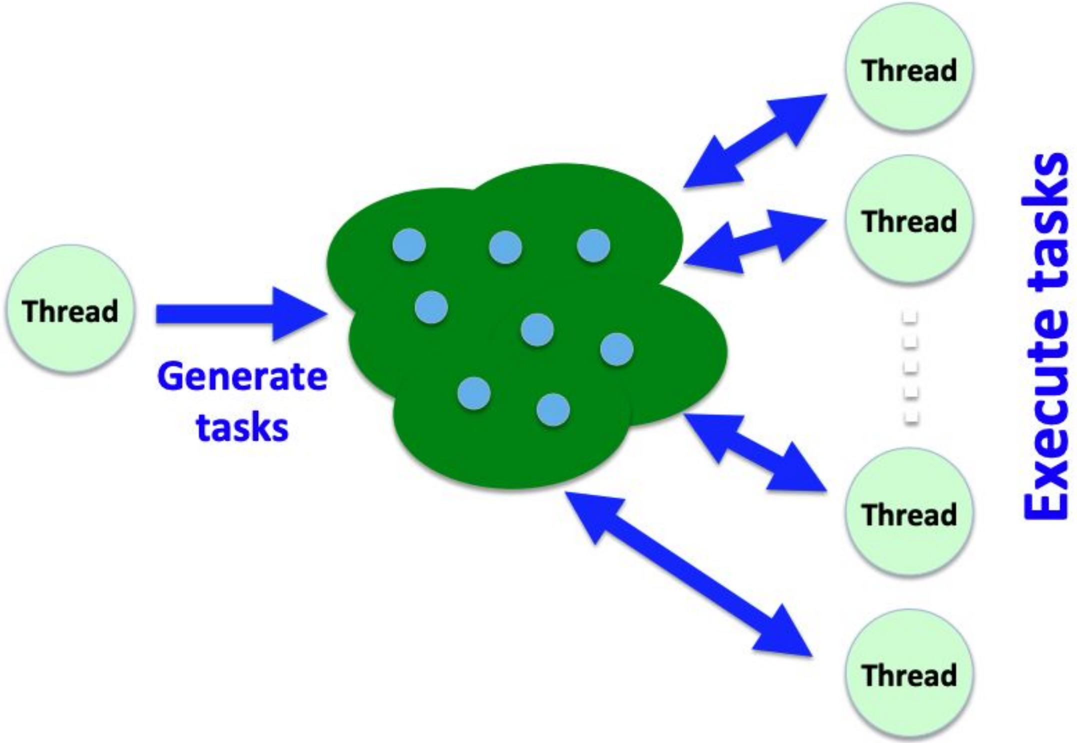
etc.

What about DAGs with independent **tasks**, where each task is more than a simple instruction (i.e., **task-based parallelism**)?

OpenMP can also creates a set of tasks -

- When a thread encounters a task construct, a new task is generated
- The moment of execution of the task is up to the runtime system
    - Execution can either be immediate or delayed
- Completion of a task can be enforced through task synchronization

OpenMP Tasks



OpenMP Tasks - Example

You want some code that prints *either* “A race car” or “A car race” - how would you do this using OpenMP?

```
int main()
{
    printf("A ");
    printf("race ");
    printf("car ");
    printf("\n");
    return(0);
}
```

OpenMP Tasks - Example

You want some code that prints *either* “A race car” or “A car race” - how would you do this using OpenMP?

```
int main()
{
    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
    }
    printf("\n");
    return(0);
}
```

OpenMP Tasks - Example

```
int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            printf("race ");
            printf("car ");
        }
    }
    printf("\n");
    return(0);
}
```

OpenMP Tasks - Example

```
int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            { printf("race ");  }
            #pragma omp task
            { printf("car ");  }
        }
    }
    printf("\n");
    return(0);
}
```

OpenMP Tasks - Example

You want some code that prints *either* “A race car is fun to watch” or “A car race is fun to watch” - how would you do this using OpenMP?

```
int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            { printf("race ");   }
            #pragma omp task
            { printf("car ");   }
            printf(“is fun to watch”);
        }
    }
    printf("\n");
    return(0);
}
```

OpenMP Tasks - Example

You want some code that prints *either* “A race car is fun to watch” or “A car race is fun to watch” - how would you do this using OpenMP?

```
int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            { printf("race ");   }
            #pragma omp task
            { printf("car ");   }
        }
        printf(“is fun to watch”);
    }
    printf("\n");
    return(0);
}
```

OpenMP Tasks - Example

You want some code that prints *either* "A race car is fun to watch" or "A car race is fun to watch" - how would you do this using OpenMP?

```
int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            { printf("race ");   }
            #pragma omp task
            { printf("car ");   }
        }
    }
    printf("is fun to watch");
    printf("\n");
    return(0);
}
```

OpenMP Tasks - Example

You want some code that prints *either* “A race car is fun to watch” or “A car race is fun to watch” - how would you do this using OpenMP?

```
int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            { printf("race "); }
            #pragma omp task
            { printf("car "); }
            #pragma omp taskwait
            printf(“is fun to watch”);
        }
    }
    printf("\n");
    return(0);
}
```

Questions?

Synchronization is very important in executing things in the correct order

You can insert synchronization everywhere so that things execute in the correct order

Unfortunately synchronization is extremely expensive (in real applications with complex tasks and dependencies)

You want to use the **minimum** number of synchronization that yields **correct** execution

OpenMP - Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- Critical
- Atomic
- Barrier
- Ordered
- Locks

OpenMP - Synchronization

```
int sum = 0;
#pragma omp parallel
{
    sum += omp_get_thread_num();
}
printf("sum = %d\n", sum);
```

What would happen if you ran this with 16 threads?

OpenMP - Synchronization

Critical section

- Mutual exclusion - only one thread at a time can enter the critical region

```
int sum = 0;
#pragma omp parallel
{
    #pragma omp critical
    sum += omp_get_thread_num();
}
printf("sum = %d\n", sum);
```

OpenMP - Synchronization

Atomic variables

- Mutual exclusion - but only to the memory location (i.e., *sum* in this example).

```
int sum = 0;
#pragma omp parallel
{
    #pragma omp atomic
    sum += omp_get_thread_num();
}
printf("sum = %d\n", sum);
```

OpenMP - Synchronization

Barrier

- Synchronizes all threads in the team - when a barrier directive is reached a thread will wait until all other threads have reached the barrier, and the threads will continue executing in parallel again.

```
int n = omp_get_num_threads();
int x[n];
for(int i = 0; i < n; i++) {
    x[i] = 0;
}
#pragma omp parallel
{
   int my_tid = omp_get_thread_num();
   for(int i = 0; i < n; i++) {
       x[(i + my_tid) % n] += my_tid;
   }
}
```

OpenMP - Synchronization

Barrier

- Synchronizes all threads in the team - when a barrier directive is reached a thread will wait until all other threads have reached the barrier, and the threads will continue executing in parallel again.

```
int n = omp_get_num_threads();
int x[n];
for(int i = 0; i < n; i++) {
    x[i] = 0;
}
#pragma omp parallel
{
   int my_tid = omp_get_thread_num();
   for(int i = 0; i < n; i++) {
       // each x[i] gets the sum of all thread IDs.
       x[(i + my_tid) % n] += my_tid;
   }
}
```

OpenMP - Synchronization

Barrier

- Synchronizes all threads in the team - when a barrier directive is reached a thread will wait until all other threads have reached the barrier, and the threads will continue executing in parallel again.

```
int n = omp_get_num_threads();
int x[n];
for(int i = 0; i < n; i++) {
    x[i] = 0;
}
#pragma omp parallel
{
   int my_tid = omp_get_thread_num();
   for(int i = 0; i < n; i++) {
       x[(i + my_tid) % n] += my_tid;
       #pragma omp barrier
   }
}
```

OpenMP - Synchronization

Assuming n = 16

- How many flops to calculate this function (i.e., each x[i] gets sum of 0 ~ 15) if only 1 thread was working on it?

OpenMP - Synchronization

How many flops to calculate this function if only 1 thread was working on it?

- 16 adds per element of x and 16 elements -> 16 x 16 = 256 flops

How many flops if 16 threads were working on it?

OpenMP - Synchronization

How many flops to calculate this function if only 1 thread was working on it?

- 16 adds per element of x and 16 elements -> 16 x 16 = 256 flops

How many flops if 16 threads were working on it?

- Still 256 flops
- It doesn't matter how many threads are working on it, as long as the total work done remains the same

OpenMP - Synchronization

Assuming 1 add requires 1 “epoch” to calculate - how many epochs for 1 thread?

OpenMP - Synchronization

Assuming 1 add requires 1 “epoch” to calculate - how many epochs for 1 thread?

- 256 epochs

Assuming 16 thread, how many epochs?

OpenMP - Synchronization

Assuming 1 add requires 1 "epoch" to calculate - how many epochos for 1 thread?

- 256 epochs

Assuming 16 thread, how many epochs?

- 16 epochs? **No, it depends**.
- If there are at least 16 add units on the processors, 16 epochs (assuming other conditions are satisfied)
- If there are fewer than 16 add units, more than 16 epochs.

OpenMP - Synchronization

Ordered

- specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

```
#pragma omp parallel
{
    #pragma omp for ordered schedule(dynamic)
    for(int i = 0; i < 16; i++) {
        printf("%d\n", i);
        #pragma omp ordered
        printf(">> %d\n", i);
    }
}
```

OpenMP - Synchronization

Locks

- Similar to a critical section - it guarantees that some instructions can only be executed by one thread at a time.
- Locks are about data (vs. critical section is about code).

```
int count[100];
for(int i = 0; i < 100; i++) {
    count[i] = 0;
}

#pragma omp parallel
{
    for(int i = 0; i < 1000; i++) {
        int x = rand() % 100;
        count[x]++;
    }
}
int sum = 0;
for(int i = 0; i < 100; i++) {
    sum += count[i];
}
```

OpenMP - Synchronization

Locks

- Similar to a critical section - it guarantees that some instructions can only be executed by one thread at a time.
- Locks are about data (vs. critical section is about code).

```
int count[100];
for(int i = 0; i < 100; i++) {
    count[i] = 0;
}

#pragma omp parallel
{
    for(int i = 0; i < 1000; i++) {
        int x = rand() % 100;
        #pragma omp critical
        count[x]++;
    }
}
int sum = 0;
for(int i = 0; i < 100; i++) {
    sum += count[i];
}
```

OpenMP - Synchronization

Locks

- Similar to a critical section - it guarantees that some instructions can only be executed by one thread at a time.
- Locks are about data (vs. critical section is about code).

Using a critical section is unnecessarily restrictive because threads become serialized at that point

Instead, use locks

OpenMP - Synchronization

Locks

- Similar to a critical section - it guarantees that some instructions can only be executed by one thread at a time.
- Locks are about data (vs. critical section is about code).

Create an array of locks - one for each element of count

Create/destroy

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set/release

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

`int omp_test_lock();` – this is useful for checking if a lock has been released and if not, do some other work

OpenMP - Synchronization

```
omp_lock_t writelock[100];
for(int i = 0; i < 100; i++) {
    omp_init_lock(&(writelock[i]));
}
#pragma omp parallel
{
    for(int i = 0; i < 1000; i++) {
        int x = rand() % 100;
        omp_set_lock(&(writelock[x]));
        count[x]++;
        omp_unset_lock(&(writelock[x]));
    }
}
for(int i = 0; i < 100; i++) {
    omp_destroy_lock(&(writelock[i]));
}
```

Data Scope

Also called “Data Sharing” - since OpenMP is for “shared memory” systems, most data are shared by default by the threads (except for loop variables in *parallel for* constructs).

However, you can explicitly define how variables are scoped, using the following:

- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- SHARED
- DEFAULT
- REDUCTION
- COPYIN

Data Scope

The PRIVATE clause declares variables in its list to be private to each thread.

- Creates a local copy of the variable and is **uninitialized**

The DEFAULT clause allows the user to specify a default scope for **all variables** in the lexical extent of any parallel region.

- By **default**, it is `default(shared)` - no need to use this.
- Common use case is `default(none)` - now, you must list storage attribute for each variable (good programming practice)

The SHARED clause declares variables in its list to be shared among all threads in the team.

- Typically used if default(none) is used

Data Scope - Example

```
int a = 1;
int b = 2;
int c = 3;

#pragma omp parallel default(none) private(b,c) shared(a)
{
    int tid = omp_get_thread_num();
    b = tid + 1;
    c = tid + 2;
    printf("thread %d -- a is %d\n", tid, a);
    printf("thread %d -- b is %d\n", tid, b);
    printf("thread %d -- c is %d\n", tid, c);
}
```

Data Scope - Example

```
int a = 1;
int b = 2;
int c = 3;

#pragma omp parallel default(none) private(b,c) shared(a)
{
    int tid = omp_get_thread_num();
    b = tid + 1;
    c = tid + 2;
    printf("thread %d -- a is %d\n", tid, a);
    printf("thread %d -- b is %d\n", tid, b);
    printf("thread %d -- c is %d\n", tid, c);
}
```

main.c:199:5: error: ‘**a**’ not specified in enclosing ‘**parallel**’ **-** compile error

Data Scope

The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the **last loop** iteration or **section** to the original variable object.

Data Scope - Example

```
int a = 1;
int b = 2;
int c = 3;

#pragma omp parallel default(none) firstprivate(b,c)
shared(a)
{
    int tid = omp_get_thread_num();
    printf("thread %d -- a is %d\n", tid, a);
    printf("thread %d -- b is %d\n", tid, b);
    printf("thread %d -- c is %d\n", tid, c);
}
```

What would happen?

Data Scope - Example

```
int a = 1;
int b = 2;
int c = 3;

#pragma omp parallel default(none) firstprivate(b,c)
shared(a)
{
    int tid = omp_get_thread_num();
    printf("thread %d -- a is %d\n", tid, a);
    printf("thread %d -- b is %d\n", tid, b);
    printf("thread %d -- c is %d\n", tid, c);
}
```

b = 2 and c = 3 initialization for each thread.

Data Scope - Example

```
int a = 1;
int b = 2;
int c = 3;

#pragma omp parallel for default(none) firstprivate(b, c)
lastprivate(a)
for(int i = 0; i < 10; i++) {
    int tid = omp_get_thread_num();
    printf("%d %d %d\n", tid, i, b + c);
    a = i;
}
printf("a = %d\n", a);
```

What happens in the loop?

What is printed at the end?

The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the ***last loop*** *iteration or* ***section*** *to the original variable object.*

Data Scope

The THREADPRIVATE **directive** specifies that **global variables** are replicated, with each thread having its own copy (by default global variables are shared).

The COPYIN **clause** provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team. Copy source is the **master thread**.

Data Scope - Example

```
int b = 2;
int c = 3;
#pragma omp threadprivate(b,c)

void main()
{
    b = 100;
    c = 200;

    int a = 1;
    #pragma omp parallel default(none) shared(a)
copyin(b,c)
    {
        printf("%d %d %d %d\n", omp_get_thread_num(), a,
b, c);
    }
}
```

Data Scope - Example

```
1 1 100 200
0 1 100 200
2 1 100 200
3 1 100 200
```

Data Scope - Example

```
int b = 2;
int c = 3;
#pragma omp threadprivate(b,c)

void main()
{
    b = 100;
    c = 200;

    int a = 1;
    #pragma omp parallel default(none) shared(a)
copyin(b,c)
    {
        printf("%d %d %d %d\n", omp_get_thread_num(), a,
b, c);
    }
}
```

What would be the values of b and c at the end? (It does compile, because b and c are global variables)

Data Scope - Example

```
3 1 2 3
2 1 2 3
1 1 2 3
0 1 100 200
```

Why?

OpenMP Directives

Parallel region construct

A block of code that will be executed by multiple threads

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)
   structured_block
```

OpenMP Directives

```
if (scalar_expression)
```

- An integer expression that, if it evaluates to true (nonzero), causes the code in the parallel region to execute in parallel. If the expression evaluates to false (zero), the parallel region is executed in serial (by a single thread).

OpenMP Directives

```
num_threads (integer expression)
```

- Specifies the number of threads that should be used to execute the code section
- When might you want to use this instead of `OMP_NUM_THREADS`?

OpenMP Directives

```
reduction (operator:list)
```

- Performs a reduction operation (using the “operator”) on the variables that appear in the list

OpenMP Directives - Example

```
/* Initialization */
int i;
const int MAX = 100;
n = MAX;
result = 0.0;
for (i=0; i < n; i++) {
    a[i] = i;
    b[i] = i;
}

#pragma omp parallel for private(i) reduction(+:result)
for (i = 0; i < n; i++) {
    result += (a[i] + b[i]);
}
printf("Final result= %f\n",result);
```

OpenMP Directives - Example

```
/* Initialization */
const int MAX = 100;
n = MAX;
result = 0.0;
for (i=0; i < n; i++) {
    a[i] = i + 1;
    b[i] = i + 1;
}

#pragma omp parallel for private(i) reduction(*:result)
for (i = 0; i < n; i++) {
    result *= (a[i] + b[i]);
}
printf("Final result= %f\n",result);
```

Should this work?

OpenMP Directives - Example

```
/* Initialization */
const int MAX = 100;
n = MAX;
result = 1.0;
for (i=0; i < n; i++) {
    a[i] = i + 1;
    b[i] = i + 1;
}

#pragma omp parallel for private(i) reduction(*:result)
for (i = 0; i < n; i++) {
    result += (a[i] + b[i]);
}
printf("Final result = %f\n",result);
```

Should this work?

Without changing the code, can you make it print different numbers?

OpenMP Directives - Example

```
/* Initialization */
const int MAX = 100;
n = MAX;
result = 1.0;
for (i=0; i < n; i++) {
    a[i] = i + 1;
    b[i] = i + 1;
}

#pragma omp parallel for private(i) reduction(*:result)
for (i = 0; i < n; i++) {
    result += (a[i] + b[i]);
}
printf("Final result = %f\n",result);
```

Should this work?

Without changing the code, can you make it print different numbers?

- Change the number of threads

OpenMP Directives - Example

<table>
<tr><th>Operation</th><th>C/C++</th><th>Initialization</th></tr>
<tr><td>Addition</td><td>+</td><td>0</td></tr>
<tr><td>Multiplication</td><td>*</td><td>1</td></tr>
<tr><td>Subtraction</td><td>-</td><td>0</td></tr>
<tr><td>Logical AND</td><td>&&</td><td>.true. / 1</td></tr>
<tr><td>Logical OR</td><td>||</td><td>.false. / 0</td></tr>
<tr><td>AND bitwise</td><td>&</td><td>all bits on / ~0</td></tr>
<tr><td>OR bitwise</td><td>|</td><td>0</td></tr>
<tr><td>Exclusive OR bitwise</td><td>^</td><td>0</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Maximum</td><td>max</td><td>Most negative #</td></tr>
<tr><td>Minimum</td><td>min</td><td>Largest positive #</td></tr>
</table>

OpenMP Directives

```
#pragma omp for [clause ...]  newline
                schedule  (type [,chunk])
                ordered
                private  (list)
                firstprivate  (list)
                lastprivate  (list)
                shared  (list)
                reduction  (operator: list)
                collapse  (n)
                nowait

   for_loop
```

OpenMP Directives

`nowait`

- No implicit barrier at the end - each thread continues to the next section without waiting for all the threads working on the loop to finish
- Typically used when the next section does not rely on the result computed in the loop (because loops have an implicit barrier, even if it's not needed)

OpenMP Directives

`collapse (n)`

Specify how many loops (in a nested loop) should be merged/collapsed into one larger iteration space

```
for(i = 0; i < X; i++) {
  for(j = 0; j < Y; j++) {
    for(k = 0; k < Z; k++) {
      // do something
    }
  }
}
```

```
for(l = 0; l < (X * Y * Z); l++) {
  int i = ?;
  int j = ?;
  int k = ?;
  // do something
}
```

OpenMP Directives

`collapse (n)`

Specify how many loops (in a nested loop) should be merged/collapsed into one larger iteration space

```
for(i = 0; i < X; i++) {
  for(j = 0; j < Y; j++) {
    for(k = 0; k < Z; k++) {
      // do something
    }
  }
}
```

→

```
for(l = 0; l < (X * Y * Z); l++) {
  int i = l / (Y * Z);
  int j = (l % (Y * Z)) / Z;
  int k = (l % (Y * Z)) % Z;
  // do something
}
```

OpenMP Directives

`collapse (n)`

- Specify how many loops (in a nested loop) should be merged/collapsed into one larger iteration space

Why?

- Outer loop iteration is fewer than # of threads -> idle threads
- You can swap the inner and outer loops, but this may not be correct and/or lead to lower data locality (you are traversing your data in a different way)

OpenMP Directives

```
schedule (type, chunk)
```

Specify how the threads are assigned to the loop iterations

- `type`
  - `static`
  - `dynamic`
  - `guided`
  - `runtime`
  - `auto`

OpenMP Directives

`schedule (type, chunk)`

`static` – loop iterations are divided into pieces of size `chunk` and statically assigned to threads. If `chunk` is not specified, iterations are evenly distributed. Least amount of overhead.

`dynamic` – loop iterations are divided into pieces of size `chunk` and dynamically assigned to threads. When a thread finishes one chunk, it is assigned another. Default `chunk` size is 1.

When would this be good to use?

OpenMP Directives

`schedule (type, chunk)`

`static` – loop iterations are divided into pieces of size `chunk` and statically assigned to threads. If `chunk` is not specified, iterations are evenly distributed. Least amount of overhead.

`dynamic` – loop iterations are divided into pieces of size `chunk` and dynamically assigned to threads. When a thread finishes one chunk, it is assigned another. Default `chunk` size is 1.

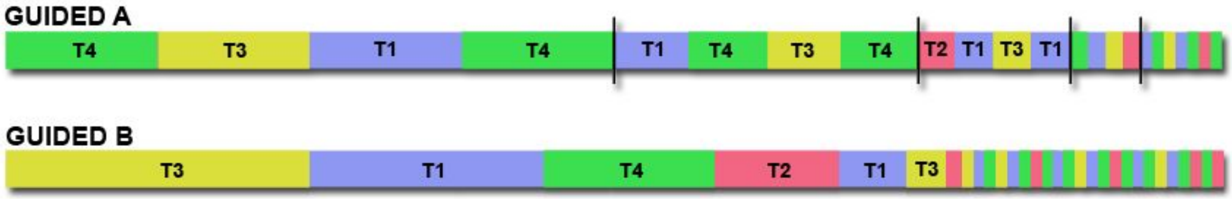
Good when loop iterations do not all take the same amount of time (and you do not know exactly by how much ahead of time).

OpenMP Directives

`schedule (type, chunk)`

`guided`

- Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned.
- The size of the initial block is proportional to: `number of iterations` / `number_of_threads`
- Subsequent blocks are proportional to `number of iterations_remaining` / `number of threads`
- The chunk parameter defines the minimum block size. The default chunk size is 1.
- How `guided` is scheduled depends on the OpenMP implementation (Guided A and B are from different impl.)



OpenMP Directives

`schedule (type, chunk)`

`runtime` – The scheduling decision is deferred until runtime by the environment variable `OMP_SCHEDULE`.

`auto` – The scheduling decision is delegated to the compiler and/or the runtime system.