

Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Announcements

Announcements

- No lectures on November 19 and 20

Recap

Recap

- Cache Recap & Examples
- False Sharing
- Tips & Tricks

Chapter 5

Shared Memory Programming with OpenMP

OpenMP

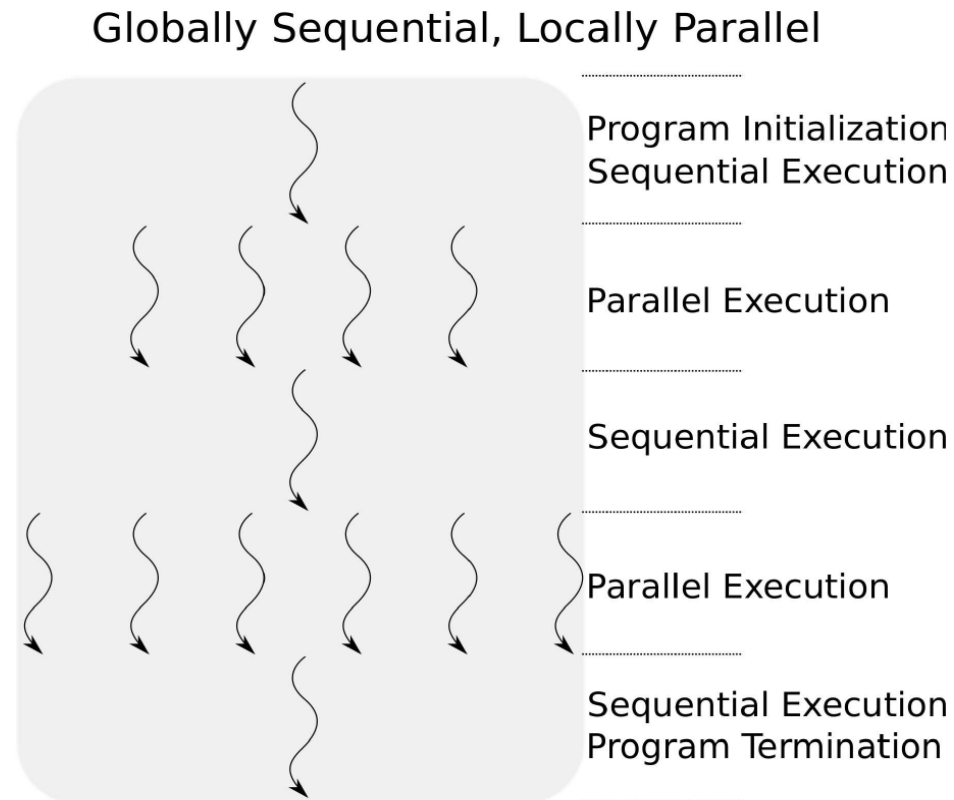
- An API for shared-memory parallel programming.
- Designed for shared-memory systems.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.

OpenMP

- OpenMP aims to decompose a sequential program into components that can be executed in parallel.
- OpenMP allows an “incremental” conversion of sequential programs into parallel ones, with the assistance of the compiler.
- OpenMP relies on compiler directives for decorating portions of the code that the compiler will attempt to parallelize.

OpenMP

- OpenMP programs are Globally Sequential, Locally Parallel.
- Programs follow the fork-join paradigm:



Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C specification.
- Compilers that don't support the pragmas ignore them.

`#pragma`

OpenMP pragmas

- `# pragma omp parallel`
 - Most basic parallel directive.
 - The number of threads that run the following structured block of code is determined by the run-time system.

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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

```
./omp_hello 4
```

compiling



running with 4 threads



```
Hello from thread 0 of 4  
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 3 of 4
```

possible
outcomes



```
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 0 of 4  
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4  
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 0 of 4
```

Thread Team Size Control

- **Universally:** via the `OMP_NUM_THREADS` environmental variable:
\$ echo \${OMP_NUM_THREADS} # to query the value
\$ export OMP_NUM_THREADS=4 # to set it in BASH
- **Program level :** via the `omp_set_num_threads` function, outside an OpenMP construct.
- **Pragma level :** via the `num_threads` clause.
- **Precedence:**
 - Universally/env. Variable
 - Program level
 - Pragma level

Questions?

Thread Team Size Control

- The `omp_get_num_threads` call returns the active threads in a parallel region. If it is called in a sequential part it returns 1.
- The `omp_get_thread_num` returns the id of the calling thread (similar to the rank in MPI)




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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

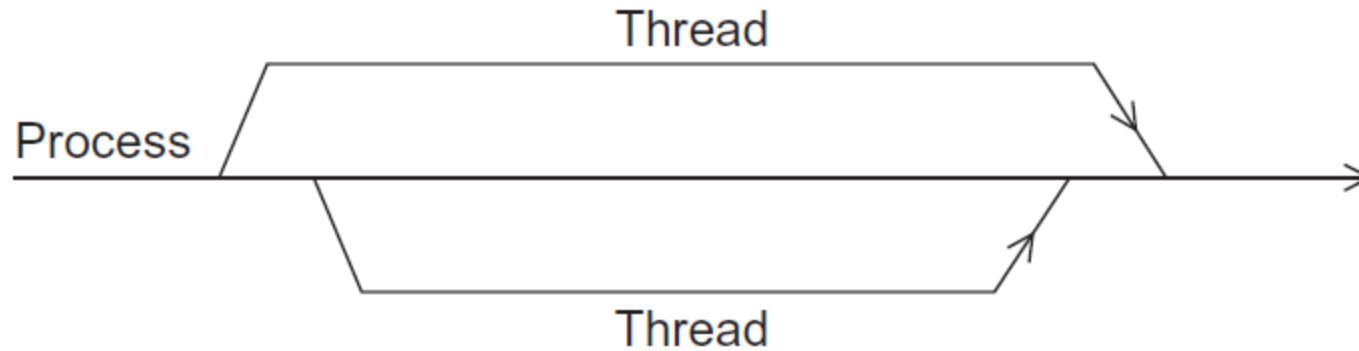
    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```

A process forking and joining two threads



clause

- Text that modifies a directive.
- The `num_threads` clause can be added to a `parallel` directive.
- It allows the programmer to specify the number of threads that should execute the following block.

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

Of note...

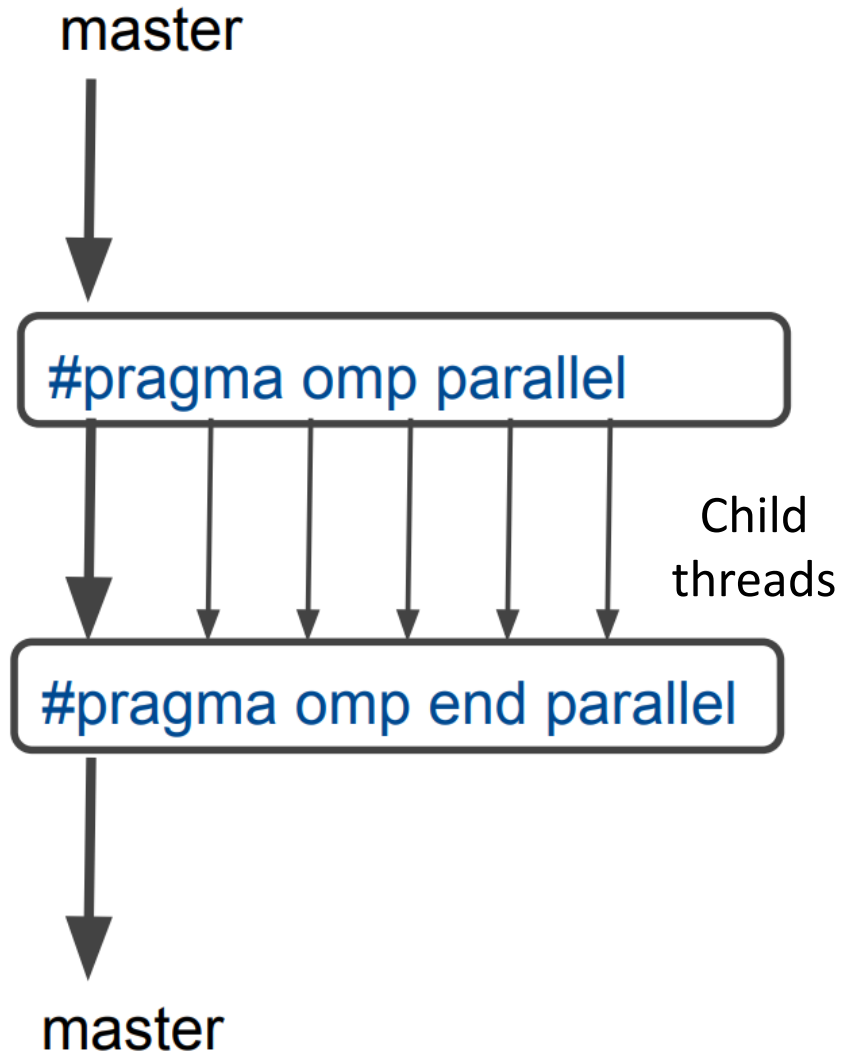
- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn't guarantee that this will actually start `thread_count` threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we're trying to start a lot of threads, we will almost always get the desired number of threads.
- After a block is completed, there is an implicit barrier.

Some terminology

- In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a **team**
- **master:** the original thread of execution
- **parent:** thread that encountered a parallel directive and started a team of threads.
- In many cases, the parent is also the master thread.
- **child:** each thread started by the parent is considered a child thread.

Parallel Construct

```
#pragma omp parallel  
{  
...  
}
```



Parallel Construct

```
1  #include<stdio.h>
2  #include<omp.h>
3
4  int main()
5  {
6      printf("Only master thread here \n");
7
8      #pragma omp parallel
9      {
10         int tid = omp_get_thread_num();
11         printf("Hello I am thread number %d \n", tid);
12     }
13
14     printf("Only master thread here \n");
15 }
```

1

3

Only master thread here
Hello I am thread number 0
Hello I am thread number 1
Hello I am thread number 2
Hello I am thread number 3
Only master thread here

2

int tid

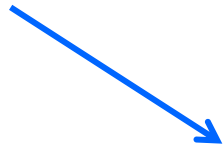
4

*Each thread has a
private copy of
variable*

- Specifying number of threads
 - `export OMP_NUM_THREADS=4`
- Execute:
 - `./basic.exe`

In case the compiler doesn't support OpenMP

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



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

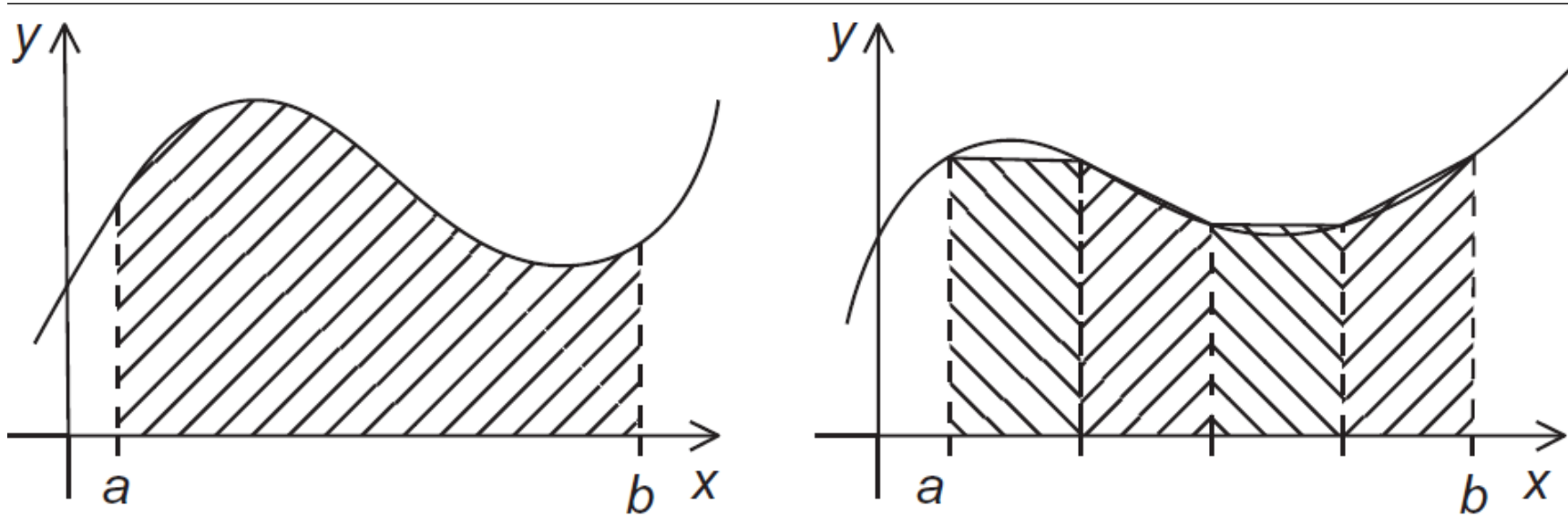

In case the compiler doesn't support OpenMP

```
# 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
```

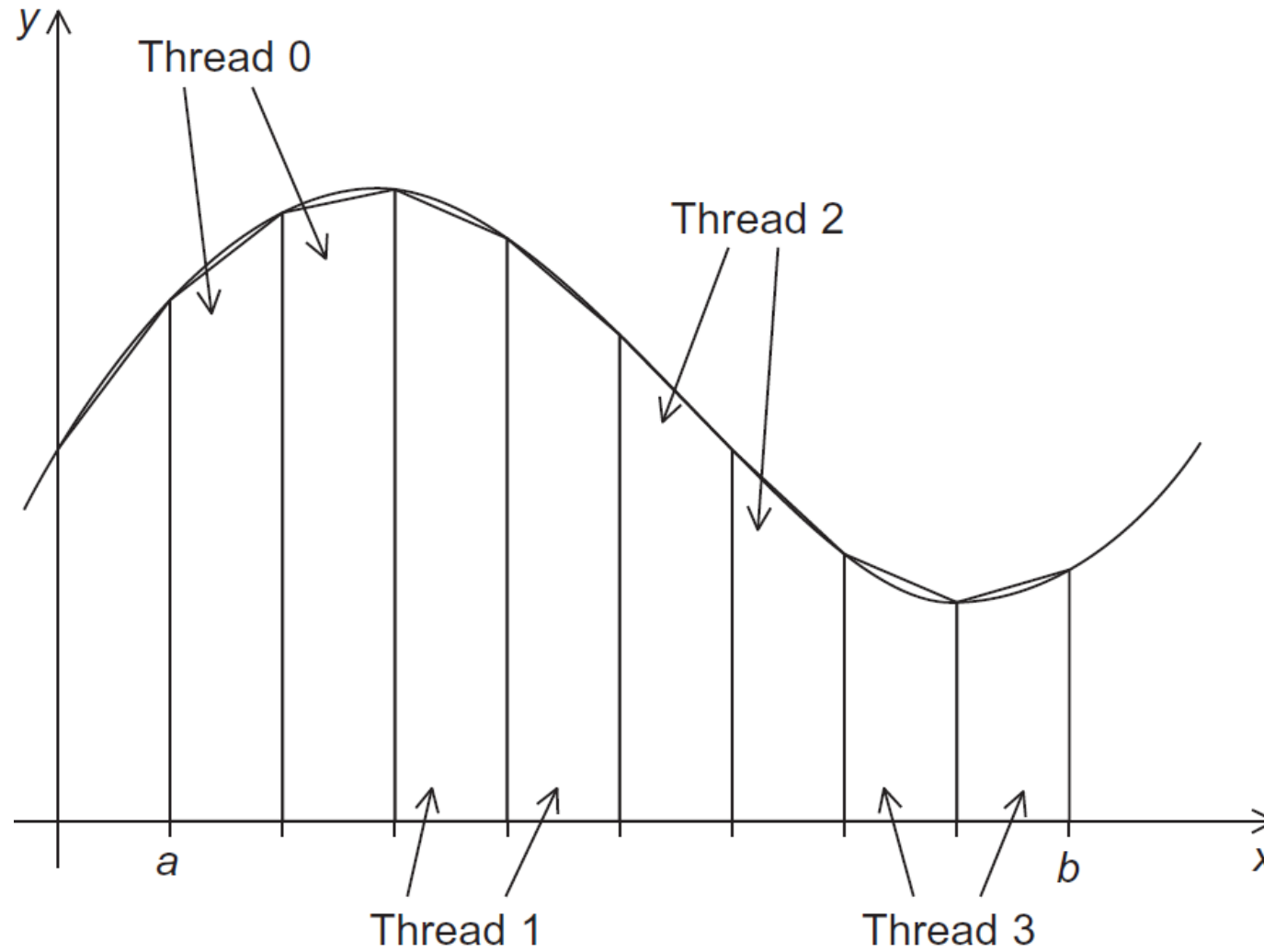
Questions?

Example: Trapezoidal rule

The trapezoidal rule



Assignment of trapezoids to threads



Unpredictable results when two (or more) threads attempt to simultaneously execute:

`global_result += my_result ;`

Time	Thread 0	Thread 1
0	<code>global_result = 0 to register</code>	<code>finish my_result</code>
1	<code>my_result = 1 to register</code>	<code>global_result = 0 to register</code>
2	<code>add my_result to global_result</code>	<code>my_result = 2 to register</code>
3	<code>store global_result = 1</code>	<code>add my_result to global_result</code>
4		<code>store global_result = 2</code>

Mutual exclusion

```
# pragma omp critical  
  global_result += my_result ;
```



only one thread can execute
the following structured block
at a time

```

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0;  /* Store result in global_result */
    double a, b;                 /* Left and right endpoints */
    int n;                       /* Total number of trapezoids */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    # pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n",
        a, b, global_result);
    return 0;
} /* main */

```



```

void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
        *global_result_p += my_result;
}.../*..Trap..*/

```

Mutual exclusion

```
# pragma omp atomic  
global_result += my_result ;
```



If atomic increment is provided by the CPU, this will be more efficient than 'critical'.

However, this only protects the read/update of the global_result variable.

E.g., global_result += my_function()

- my_function will still be executed in parallel

Questions?

OpenMP vs. Pthreads

- OpenMP provides a higher abstraction interface
 - On POSIX systems, it is most likely implemented on top of Pthreads
 - On non-POSIX systems (e.g., Windows), it will be implemented on top of some other threading abstraction
 - Thus, it is more portable than Pthreads (you write the OpenMP code and run it «everywhere» without modifying it)
 - Some OpenMP constructs can be used to run code on GPUs (we won't see this)
- Pthreads provides you with much more fine-grained control
 - As usual is a matter of trade-offs between ease-of-use and flexibility/performance
- Mixing OpenMP and Pthreads in the same code should work, but there might be corner cases where it does not. It is usually better to use either one or the other unless you have very strong reasons to mix them

Variables Scope

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

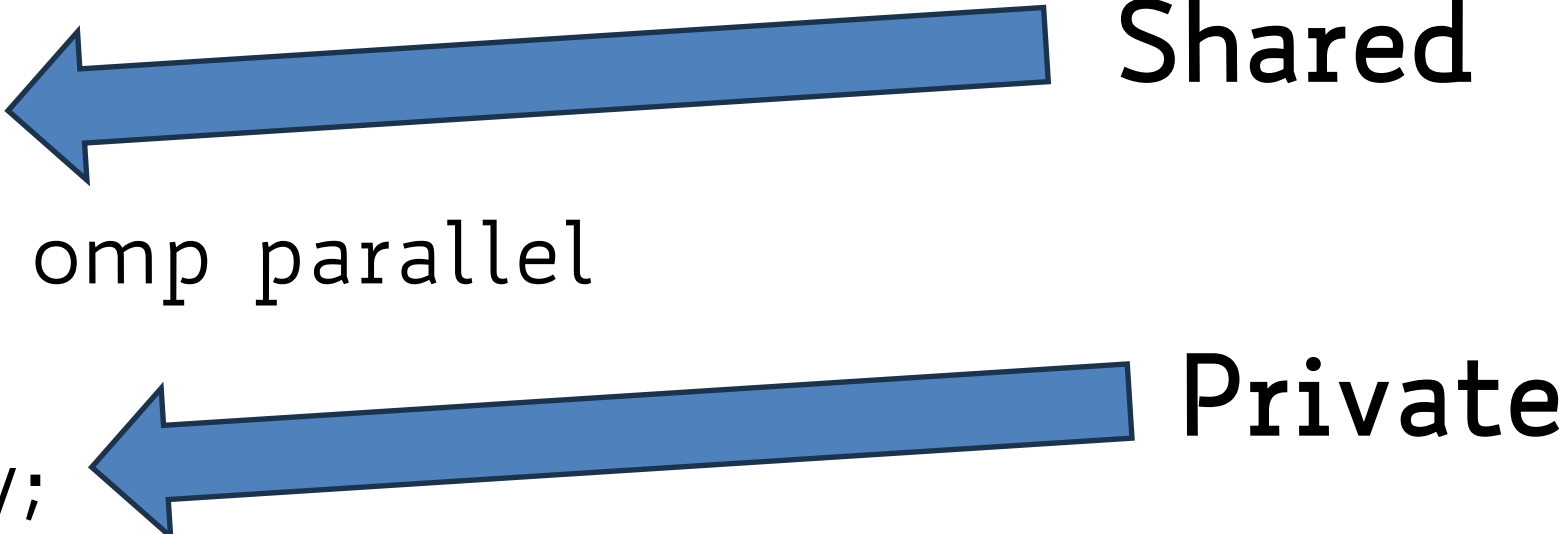
- A variable that can be accessed by all the threads in the team has **shared** scope.
- A variable that can only be accessed by a single thread has **private** scope.
- The default scope for variables declared before a parallel block is **shared**.

Scope in OpenMP

```
...  
int x;  
#pragma omp parallel  
{  
    int y;  
    ...  
}  
...
```

Shared

Private



Questions?

Reduction Clause

In the version we shown, we use `global_result_p` as the output parameter, where each thread accumulates the result.

```
void Trap(double a, double b, int n, double* global_result_p);
```

Can we do it by having the function returning the computed area? E.g.,:

```
double Local_trap(double a, double b, int n);
```

If we fix it like this...

```
global_result = 0.0;  
# pragma omp parallel num_threads(thread_count)  
{  
#   pragma omp critical  
    global_result += Local_trap(double a, double b, int n);  
}
```

... we force the threads to execute sequentially.

We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
        double my_result = 0.0;  /* private */

        my_result += Local_trap(double a, double b, int n);
#       pragma omp critical
        global_result += my_result;
    }
```

Regardless of how we do it, this is a Reduce (similar to what we have seen in MPI)

Does OpenMP provide a native way of doing that?

Reduction operators

- A **reduction operator** is a binary operation (such as addition or multiplication).
- A **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

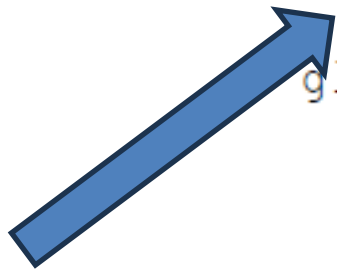
A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```



+, *, -, &, |, ^, &&, ||

```
global_result = 0.0;  
# pragma omp parallel num_threads(thread_count) \  
  reduction(+: global_result)  
  global_result += Local_trap(double a, double b, int n);
```



ATTENTION: If I do not specify the reduction clause I would have a race condition

Reduction

- The private variables created for a **reduction** clause are initialized to the *identity value* for the operator. For example, if the operator is multiplication, the private variables would be initialized to 1.
- The reduction at the end of the parallel section accumulates the *outside* value and the private values computed inside the parallel region

Example

```
int acc = 6;  
#pragma omp parallel num_threads(5) reduction(* : acc)  
{  
    acc += omp_get_thread_num();  
    printf("thread %d: private acc is  
           %d\n",omp_get_thread_num(),acc);  
}  
printf("after: acc is %d\n",acc);
```

Example

```
int acc = 6;
#pragma omp parallel num_threads(5) reduction(* : acc)
{
    acc += omp_get_thread_num(); // 1,2,3,4,5 (1+tid)
    printf("thread %d: private acc is\n",omp_get_thread_num(),acc);
}
printf("after: acc is %d\n",acc); //acc=720
```

output:

```
tid=0 sum = 1
tid=3 sum = 4
tid=4 sum = 5
tid=2 sum = 3
tid=1 sum = 2
Final sum = 720
```

Questions?

More on Scope

The default clause

- Lets the programmer specify the scope of each variable in a block.

default(none)

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

Scope modifying clauses

- **shared**: the default behavior for variables declared outside of a parallel block. It needs to be used only if *default(none)* is also specified.
- **reduction** : a reduction operation is performed between the private copies and the "outside" object. The final value is stored in the "outside" object.
- **private** : creates a separate copy of a variable for each thread in the team. Private variables are not initialized, so one should not expect to get the value of the variable declared outside the parallel construct.

Example

```
int x = 5;
#pragma omp parallel private(x)
{
    x = x+1; // dangerous (x not initialized)
    printf("thread %d: private x is
           %d\n",omp_get_thread_num(),x);
}
printf("after: x is %d\n",x); // also dangerous (prints the x declared in the first line)
```

Example

```
int x = 5;
#pragma omp parallel private(x)
{
    x = x+1; // dangerous (x not initialized)
    printf("thread %d: private x is
           %d\n",omp_get_thread_num(),x);
}
printf("after: x is %d\n",x); // also dangerous (prints the x declared in the first line)
```

output:

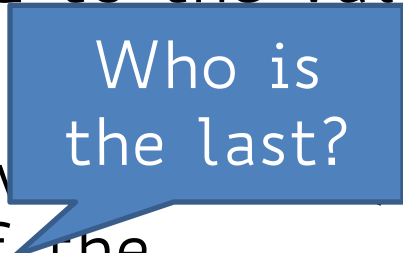
```
thread 0: private x is 1
thread 1: private x is 1
thread 3: private x is 1
thread 2: private x is 1
after: x is 5
```


Scope modifying clauses

- `firstprivate`: behaves the same way as the `private` clause, but the private variable copies are initialized to the value of the “outside” object.
- `lastprivate`: behaves the same way as the `private` clause, but the thread finishing the last iteration of the sequential block copies the value of its object to the “outside” object.

Scope modifying clauses

- `firstprivate` : behaves the same way as the `private` clause, but the private variable copies are initialized to the value of the "outside" object.
- `lastprivate` : behaves the same way as the `private` clause, but the thread finishing the last iteration of the sequential block copies the value of its object to the "outside" object.



Who is
the last?

Scope modifying clauses

When we declare a variable as „private“ it won't exist anymore outside of the scope. How to declare a „private“ variable which persist between different parallel sections?

- `threadprivate` : creates thread-specific, *persistent* storage (i.e. for the duration of the program) for global data. The variable needs be a global or static variable in C, or a static class member in C++.
- `copyin`: Used in conjunction with the `threadprivate` clause to initialize the `threadprivate` copies of a team of threads from the master thread's variables.

threadprivate example

```
int tp;

int main(int argc, char **argv) {
    #pragma omp threadprivate(tp)

    #pragma omp parallel num_threads(7)
        tp = omp_get_thread_num();

    #pragma omp parallel num_threads(9)
        printf("Thread %d has %d\n", omp_get_thread_num(), tp);
    return 0;
}
```

Thread 3 has 3
Thread 2 has 2
Thread 8 has 0
Thread 7 has 0
Thread 5 has 5
Thread 4 has 4
Thread 6 has 6
Thread 1 has 1
Thread 0 has 0

threadprivate example

If the thread private data starts out identical in all threads

```
#pragma omp threadprivate(private_var)
...
private_var = 1;

#pragma omp parallel copyin(private_var)
    private_var += omp_get_thread_num()
```

threadprivate example

If one thread needs to set all thread private data to its value:

```
#pragma omp parallel
{
...
#pragma omp single copyprivate(private_var)
    private_var = 4;
...
}
```

Example

```
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void default_none(int a) {
    const int c = 1;
    int i = 0;

    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_threads();
        /* O.K. - j is declared within parallel region */
        a = z[j]; /* O.K. - a is listed in private clause */
        /* - z is listed in shared clause */
        x = c; /* O.K. - x is threadprivate */
        /* - c has const-qualified type */
        z[i] = y; /* Error - cannot reference i or y here */

        #pragma omp for firstprivate(y)
        /* Error - Cannot reference y in the firstprivate clause */
        for (i=0; i<10 ; i++) {
            z[i] = i; /* O.K. - i is the loop iteration variable */
        }

        z[i] = y; /* Error - cannot reference i or y here */
    }
}
```

Questions?

Parallel For

Parallel for

- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

Trapezoid Example

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```



```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
# pragma omp parallel for num_threads(thread_count) \  
    reduction(+: approx)  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```

Legal forms for parallelizable for statements

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

Why? It allows the runtime system to determine the number of iterations prior to the execution of the loop

Caveats

- The variable `index` must have integer or pointer type (e.g., it can't be a float).
- The expressions `start`, `end`, and `incr` must have a compatible type. For example, if `index` is a pointer, then `incr` must have integer type.
- The expressions `start`, `end`, and `incr` must not change during execution of the loop.
- During execution of the loop, the variable `index` can only be modified by the "increment expression" in the `for` statement.

examples

```
for (i=0; i<n; i++) {  
    if (...) break;    //cannot be parallelized  
}
```

```
for (i=0; i<n; i++) {  
    if (...) return 1; //cannot be parallelized  
}
```

```
for (i=0; i<n; i++) {  
    if (...) exit();   //can be parallelized  
}
```

```
for (i=0; i<n; i++) {  
    if (...) i++;      //CANNOT be parallelized  
}
```

Questions?

Example: Odd-Even Sort

Odd-Even Sort

This might fork/join new threads everytime it is called (depends on the implementation)

If it does so, we would have some **overhead**

Can we just create the threads at the beginning (before line 1)?

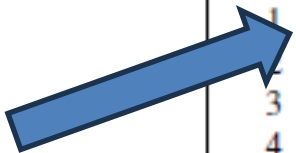
```

1   for (phase = 0; phase < n; phase++) {
2       if (phase % 2 == 0)
3           #pragma omp parallel for num_threads(thread_count) \
4               default(none) shared(a, n) private(i, tmp)
5           for (i = 1; i < n; i += 2) {
6               if (a[i-1] > a[i]) {
7                   tmp = a[i-1];
8                   a[i-1] = a[i];
9                   a[i] = tmp;
10            }
11        }
12    else
13        #pragma omp parallel for num_threads(thread_count) \
14            default(none) shared(a, n) private(i, tmp)
15        for (i = 1; i < n-1; i += 2) {
16            if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20            }
21        }
22    }

```

Odd-Even Sort

Fork threads
only here



```
1  # pragma omp parallel num_threads(thread_count) \  
2    default(none) shared(a, n) private(i, tmp, phase)  
3    for (phase = 0; phase < n; phase++) {  
4      if (phase % 2 == 0)  
5        # pragma omp for  
6        for (i = 1; i < n; i += 2) {  
7          if (a[i-1] > a[i]) {  
8            tmp = a[i-1];  
9            a[i-1] = a[i];  
10           a[i] = tmp;  
11         }  
12       }  
13     else  
14       # pragma omp for  
15       for (i = 1; i < n-1; i += 2) {  
16         if (a[i] > a[i+1]) {  
17           tmp = a[i+1];  
18           a[i+1] = a[i];  
19           a[i] = tmp;  
20         }  
21       }  
22     }
```

Odd-Even Sort

Table 5.2 Odd-Even Sort with Two parallel for Directives and Two for Directives (times are in seconds)

thread_count	1	2	3	4
Two parallel for directives	0.770	0.453	0.358	0.305
Two for directives	0.732	0.376	0.294	0.239

Reusing the same threads provide faster execution times

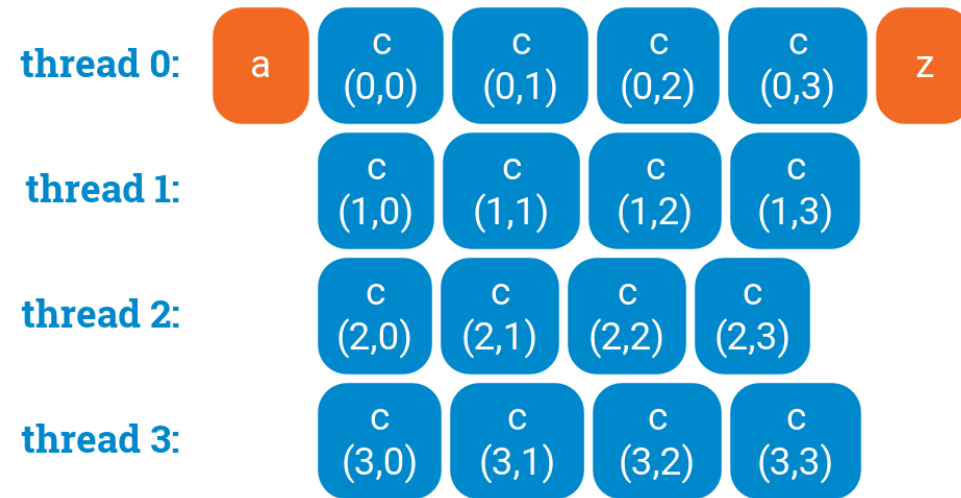
Questions?

Nested Loops

Nested for loops

- If we have nested for loops, it is often enough to simply **parallelize the outermost loop**

```
a();  
#pragma omp parallel for  
for (int i = 0; i < 4; ++i) {  
    for (int j = 0; j < 4; ++j) {  
        c(i, j);  
    }  
}  
z();
```



Nested for loops

- Sometimes the outermost loop is so short that not all threads are utilized:
3 iterations, so it won't have sense to start more than 3 threads

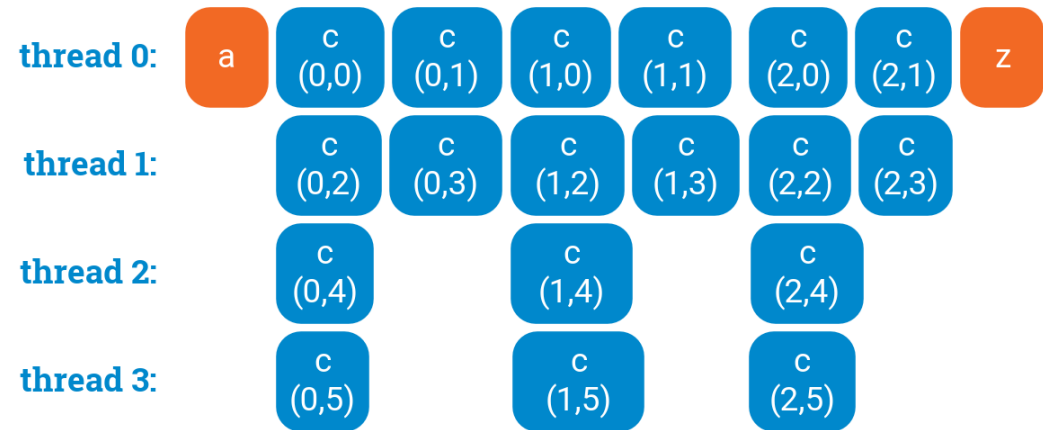
```
a();  
#pragma omp parallel for  
for (int i = 0; i < 3; ++i) {  
    for (int j = 0; j < 6; ++j) {  
        c(i, j);  
    }  
}  
z();
```



Nested for loops

- We could try to parallelize the inner loop, but there is no guarantee that the thread utilization is better

```
a();  
for (int i = 0; i < 3; ++i) {  
    #pragma omp parallel for  
    for (int j = 0; j < 6; ++j) {  
        c(i, j);  
    }  
}  
z();
```



Nested for loops

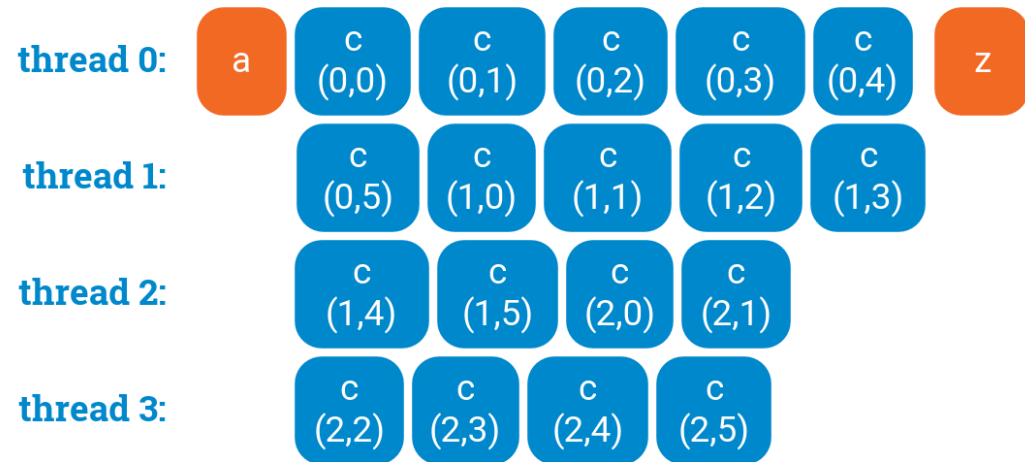
- The correct solution is to **collapse it into one loop** that does 18 iterations. We can do it manually:

<https://ppc.cs.aalto.fi/ch3/nested/>

Nested for loops

- The correct solution is to **collapse it into one loop** that does 18 iterations. We can do it manually:

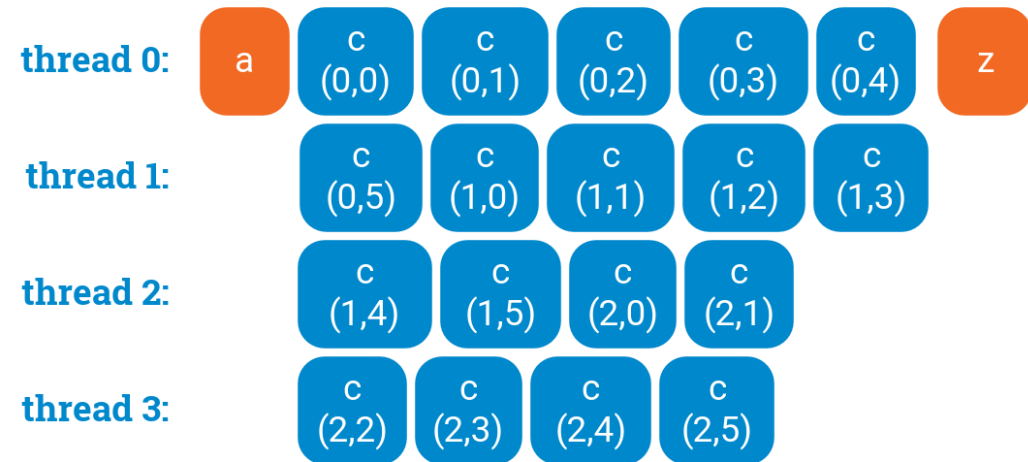
```
a();  
#pragma omp parallel for  
for (int ij = 0; ij < 3*6; ++ij) {  
    c(ij / 6, ij % 6);  
}  
z();
```



Nested for loops

- we can ask OpenMP to do it for us:

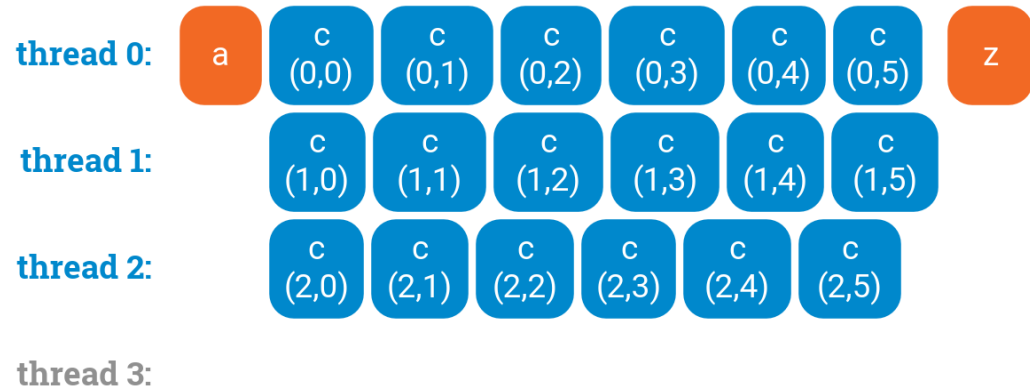
```
a();  
#pragma omp parallel for  
collapse(2)  
for (int i = 0; i < 3; ++i) {  
    for (int j = 0; j < 6; ++j)  
        c(i, j);  
}  
z();
```



Nested for loops

- **Wrong way:** “Nested parallelism” is disabled in OpenMP by default (i.e., inner parallel for pragmas will be ignored)

```
a();  
#pragma omp parallel for  
for (int i = 0; i < 3; ++i) {  
    #pragma omp parallel for  
        for (int j = 0; j < 6; ++j)  
            c(i, j);  
}  
z();
```



Nested for loops

- **Wrong way:** If "Nested parallelism" is enabled it will create 12 threads on a server with 4 cores (3*4)!

```
a();  
#pragma omp parallel for  
for (int i = 0; i < 3; ++i) {  
    #pragma omp parallel for  
        for (int j = 0; j < 6; ++j)  
            c(i, j);  
}  
z();
```