Latin American Introductory School on Parallel Programming
and Parallel Architecture for High-Performance Computing

Introduction to OpenMP

Dr. Richard Berger

High-Performance Computing Group
College of Science and Technology
Temple University
Philadelphia, USA

richard.berger@temple.edu

The Abdus Salam International Centre for Theoretical Physics

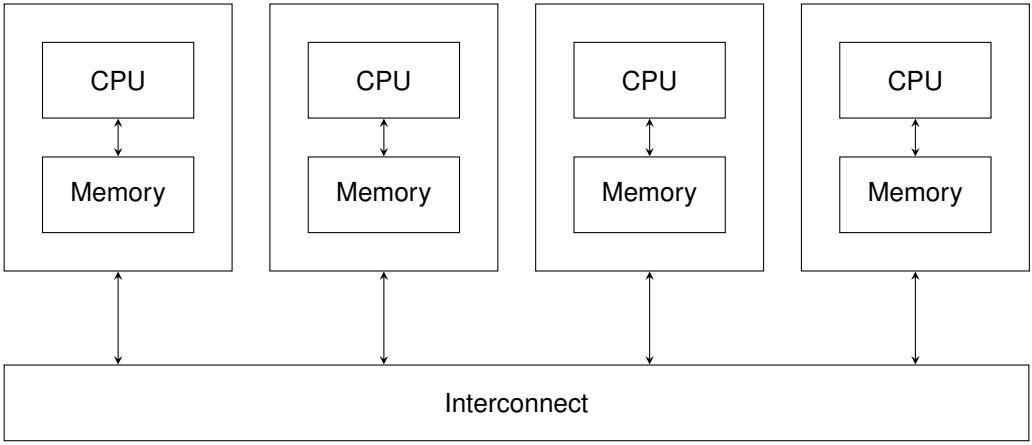
TEMPLE UNIVERSITY

Outline

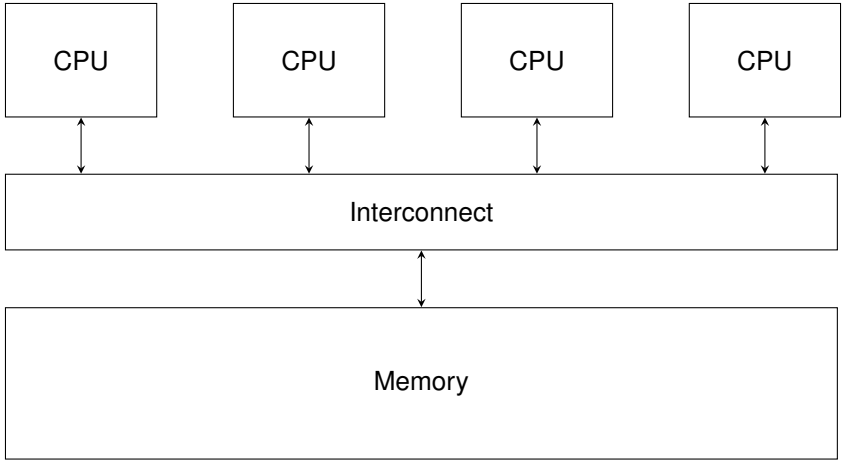
Outline

Outline

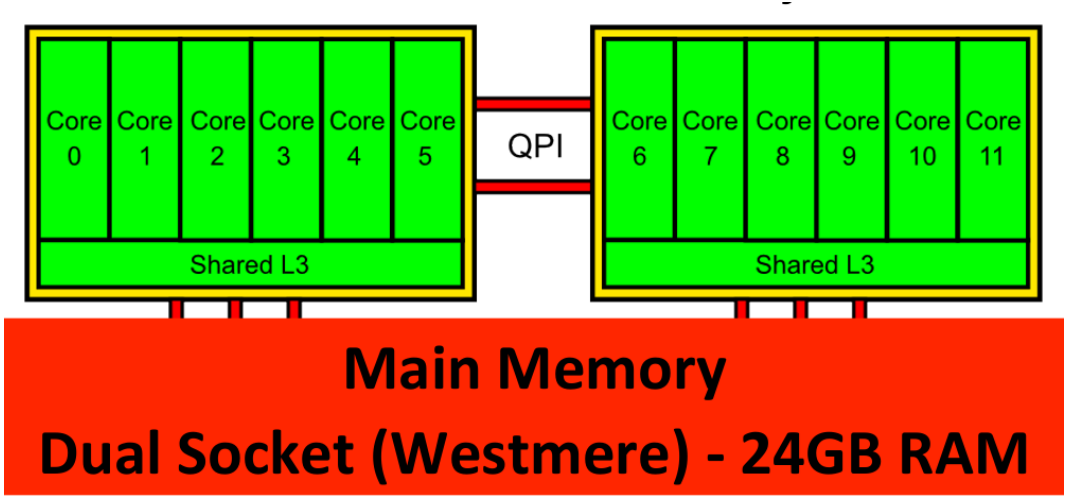
A distributed memory system



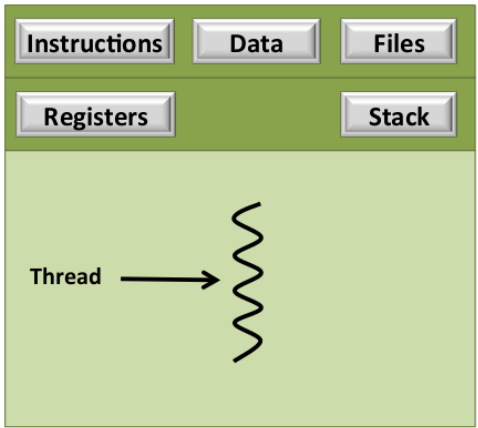
A shared-memory system

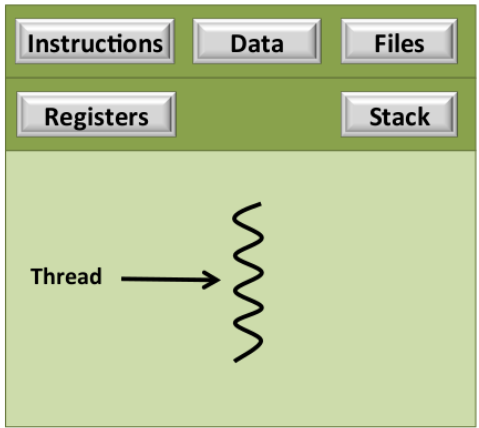


Real World: Multi-CPU and Multi-Core NUMA System

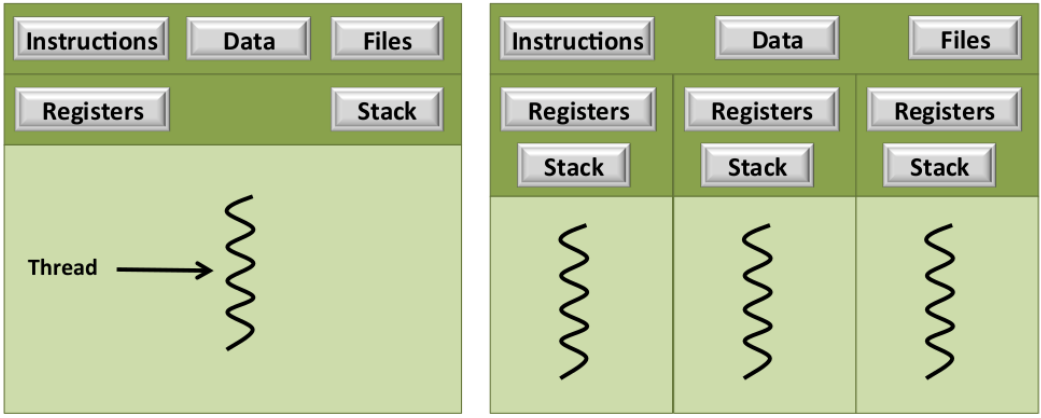


Processes vs. Threads





Processes vs. Threads



Process vs. Thread

Process

- a block of memory for the stack
- a block of memory for the heap
- descriptors of resources allocated by the OS for the process, such as file descriptors (STDIN, STDOUT, STDERR)
- security information about what the process is allowed to access hardware, who is the owner, etc.
- process state: content of registers, program counter, state (ready to run, waiting on resource)

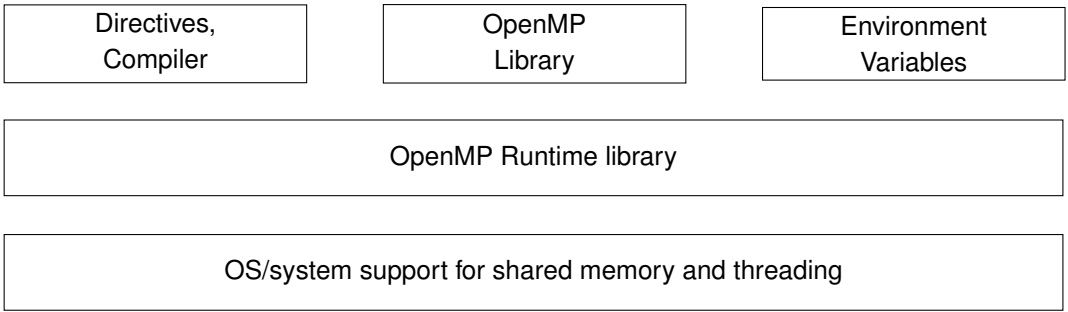
Thread

- “light-weight” processes, that live within a process and have access to its data and resources
- have their own process state, such as program counter, content of registers, and stack
- share the process heap
- each thread follows its own flow of control
- works on private data and can communicate with other threads via shared data

Outline

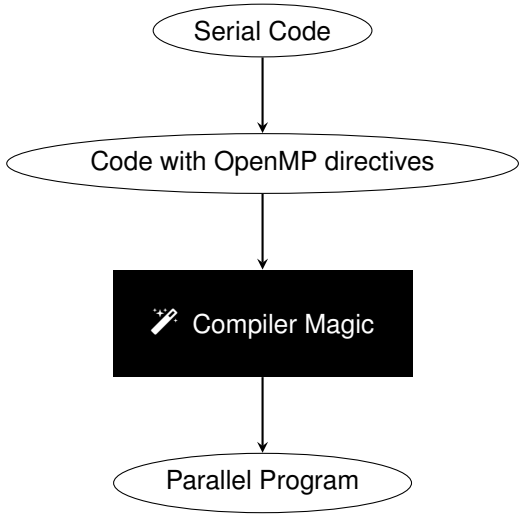
What is OpenMP?

- an Open specification for Multi Processing
- a collaboration between hardware and software industry
- a **high-level** application programming interface (API) used to write multi-threaded, portable shared-memory applications
- defined for both C/C++ and Fortran


Directives,
Compiler
OpenMP
Library
Environment
Variables
OpenMP Runtime library
OS/system support for shared memory and threading

OpenMP in a Nutshell

- OpenMP is **NOT** a programming language, it extends existing languages
- OpenMP makes it easier to add parallelization to existing serial code
- It can be added incrementally
- You annotate your code with OpenMP directives
- This gives the compiler the necessary information to parallelize your code
- The compiler itself can then be seen as a black box that transforms your annotated code into a parallel version based on a well-defined set of rules



Directives Format

A directive is a special line of source code which only has a meaning for supporting compilers.
These directives are distinguished by a sentinel at the start of the line

Fortran

`!$OMP` (or `C$OMP` or `*$OMP`)

C/C++

`#pragma` `omp`

OpenMP in C++

- Format

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

- Library functions are declared in the `omp.h` header

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

Outline

Serial Hello World

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");

    return 0;
}
```

Output:

```
Hello World!
```

Hello OpenMP

```
#include <stdio.h>

int main() {
    #pragma omp parallel
    printf("Hello OpenMP!\n");

    return 0;
}
```

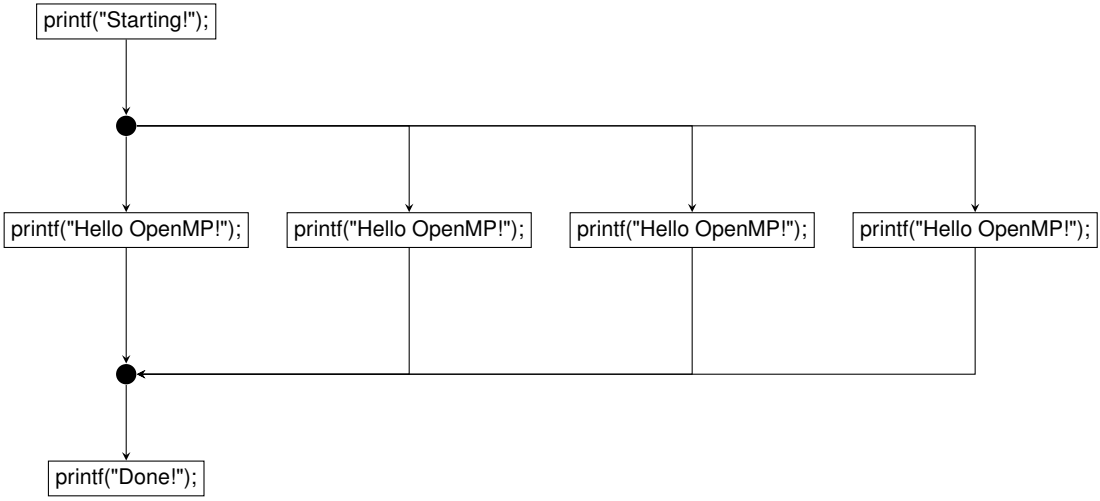
Hello OpenMP

```
#include <stdio.h>

int main() {
    printf("Starting!\n");

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

    printf("Done!\n");

    return 0;
}
```

Output:

```
Starting!
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
Done!
```

Hello OpenMP



Compiling an OpenMP program

GCC

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

```
g++ -fopenmp -o omp_hello omp_hello.cpp
```

Intel

```
icc -qopenmp -o omp_hello omp_hello.c
```

```
icpc -qopenmp -o omp_hello omp_hello.cpp
```

Running an OpenMP program

```
# default: number of threads equals number of cores
./omp_hello
```

```
# set environment variable OMP_NUM_THREADS to limit default
$ OMP_NUM_THREADS=4 ./omp_hello

# or

$ export OMP_NUM_THREADS=4
$ ./omp_hello
```

Outline

Outline

`parallel` region

Launches a team of threads to execute a block of structured code in parallel.

```
#pragma omp parallel
statement; // this is executed by a team of threads

// implicit barrier: execution only continues when all
// threads are complete
```

```
#pragma omp parallel
{
    // this is executed by a team of threads
}
// implicit barrier: execution only continues when all
// threads are complete
```

C/C++ and Fortran Syntax

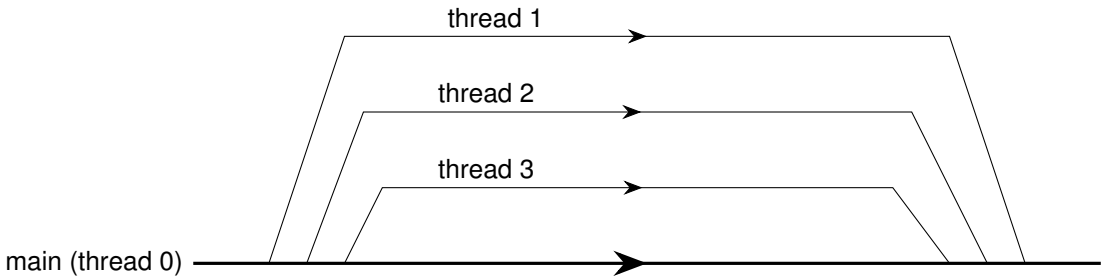
C/C++

```
#pragma omp parallel [clauses]
{
    ...
}
```

Fortran

```
!$omp parallel [clauses]
...
...
!$omp end parallel
```

Fork-Join



- Each thread executes the structured block independently
- The end of a parallel region acts as a **barrier**
- All threads must reach this barrier, before the main thread can continue.

Different ways of controlling the number of threads

1. At the `parallel` directive

```
#pragma omp parallel num_threads(4)
{
    ...
}
```

2. Setting a default via the `omp_set_num_threads(n)` library function

Set the number of threads that should be used by the **next** `parallel` region

3. Setting a default with the `OMP_NUM_THREADS` environment variable

number of threads that should be spawned in a parallel region if there is no other specification. By default, OpenMP will use all available cores.

`if`-clause

We can make a parallel region directive conditional. If the condition is **`false`**, the code within runs in serial (by a single thread).

```
#pragma omp parallel if(ntasks > 1000)
{
    // do computation in parallel or serial
}
```

Library functions

- Requires the inclusion of the `omp.h` header!

`omp_get_num_threads()`

Returns the number of threads in current team

`omp_set_num_threads(n)`

Set the number of threads that should be used by the **next** `parallel` region

`omp_get_thread_num()`

Returns the current thread's ID number

`omp_get_wtime()`

Return walltime in seconds

Hello World with OpenMP

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

int main() {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        printf("Hello from thread %d/%d!\n", tid, nthreads);
    }
    return 0;
}
```

Output of parallel Hello World

Output of first run:

```
Hello from thread 2/4!
Hello from thread 1/4!
Hello from thread 0/4!
Hello from thread 3/4!
```

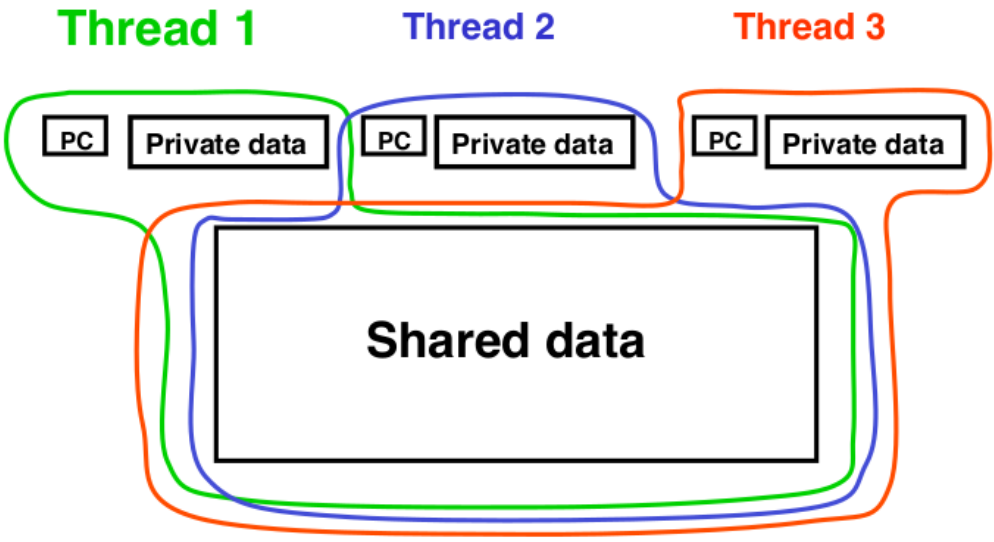
Output of second run:

```
Hello from thread 1/4!
Hello from thread 2/4!
Hello from thread 0/4!
Hello from thread 3/4!
```

Execution of threads is non-deterministic!

Outline

OpenMP Data Environment



Variable scope: **private** and shared variables

- by default all variables which are visible in the parent scope of a parallel region are **shared**
- variables declared inside of the parallel region are by definition of the scoping rules of C/C++ only visible in that scope. Each thread has a **private** copy of these variables

```
int a; // shared

#pragma omp parallel
{
    int b; // private
    ...
    // both a and b are visible
    // a is shared among all threads
    // each thread has a private copy of b
    ...
} // end of scope, b is no longer visible
```

Variable scope: **private** and shared variables

- a variable's scope can be modified at the beginning of a parallel region using clauses
- useful for legacy code where all variables are declared at the beginning of a function. E.g. in Legacy Fortran an C code you need to declare all variables at the beginning of a function.

```
int a = 1.0;
int b = 3.0;
int c = 5.0;

#pragma omp parallel shared(a) private(b,c)
{
    // a is shared among all threads (a = 1.0)
    // each thread has a private copy of b and c
    // b = uninitialized
    // c = uninitialized
    // outside b and C are not visible
    ...
}
```

Variable scope: **private** and shared variables

Equivalent

```
int a = 1.0;
int b = 3.0;
int c = 5.0;

#pragma omp parallel shared(a)
{
    // a is shared among all threads (a = 1.0)
    // each thread has a private copy of b and c
    int b, c;
    // outside b and c are not visible
    ...
}
```

Variable scope: `firstprivate`

- the `firstprivate` clause does the same as **`private`**, but initializes each copy with the value of the parent thread.

```
int a = 1.0;
int b = 3.0;
int c = 5.0;

#pragma omp parallel firstprivate(b) private(c)
{
    // a is shared
    // the value of the private b is 3.0
    // the value of the private c is uninitialized
    ...
}
```

Variable scope: **default**

- you can change default scope of variables using the **default** clause
- valid values: `shared` and `none`
- when **default** is `none`, the compiler will complain about variables which aren't either marked `shared`, **private** or `firstprivate`
- this is useful to avoid bugs

```
int a = 1.0;
int b = 3.0;
int c = 5.0;

#pragma omp parallel default(none) firstprivate(b) private(c)
{
    // accessing a will create a compile error,
    // since it's not part of any shared, private or firstprivate
    // clause
    printf("%d\n", a);
    ...
```

Outline

A motivating example: Trapezoidal Rule

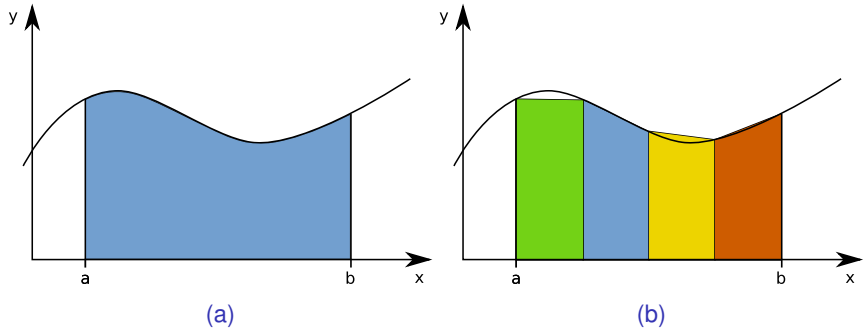


Figure: Approximating the area using the trapezoidal rule

$$\int_a^b f(x)\mathrm{d}x \approx \underbrace{\frac{(b-a)}{n}}_{h}\left[\frac{f(x_0)+f(x_1)}{2}+\frac{f(x_1)+f(x_2)}{2}+\ldots+\frac{f(x_{n-2})+f(x_{n-1})}{2}+\frac{f(x_{n-1})+f(x_n)}{2}\right]$$

$$\int_a^b f(x)\mathrm{d}x \approx \underbrace{\frac{(b-a)}{n}}_{h}\left[\frac{f(x_0)}{2}+f(x_1)+f(x_2)+\ldots+f(x_{n-1})+\frac{f(x_n)}{2}\right]$$

```
double h = (b - a) / n;
double approx = (f(a) + f(b)) / 2.0;

for(int i = 1; i <= n-1; ++i) {
    double x_i = a + i*h;
    approx += f(x_i);
}
approx = h * approx;
```

Trapozoidal Rule

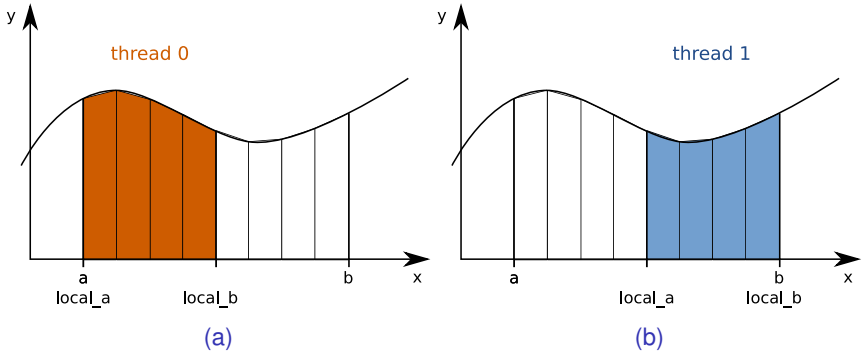
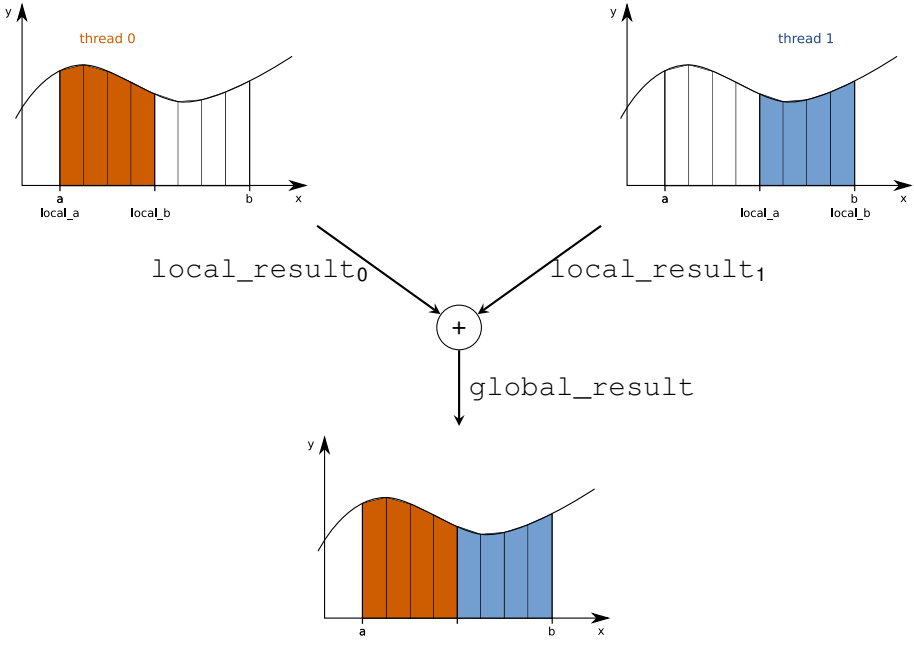


Figure: Splitting the work between two threads


thread 0
y
a
local_a
local_b
b
x
thread 1
y
a
local_a
local_b
b
x
local_result0
local_result1
+
global_result
y
a
b
x

First attempt

```
double global_result = 0.0;

#pragma omp parallel
{
    double h = (b - a) / n;
    int    local_n = ...
    double local_a = ...
    double local_b = ...

    double local_result = (f(local_a) + f(local_b)) / 2.0;
    for(int i = 1; i <= local_n-1; ++i) {
        double x_i = local_a + i*h;
        local_result += f(x_i);
    }
    local_result = h * local_result;

    ...
}
```

Each thread should be assigned a block of `nlocal` trapezoids[1]:

```
int nlocal = n / nthreads;
```

Then for each thread, the left endpoint of the range will be:

```
thread 0: local_a = a + 0*nlocal*h;
thread 1: local_a = a + 1*nlocal*h;
thread 2: local_a = a + 2*nlocal*h;
...
```

therefore:

```
double local_a = a + tid * nlocal * h;
```

Since the lengh of the assigned interval is `nlocal*h`, the right endpoint is set to:

```
double local_b = local_a + nlocal * h;
```

[1] let's assume `n` can be divided by `nthreads` without remainder

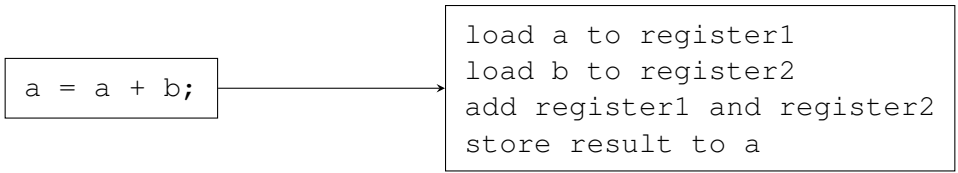
```
double global_result = 0.0; // shared variable

#pragma omp parallel
{
    double h = (b - a) / n;

    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    double local_result = (f(local_a) + f(local_b)) / 2.0;
    for(int i = 1; i <= local_n-1; ++i) {
        double x_i = local_a + i*h;
        local_result += f(x_i);
    }
    local_result = h * local_result;

    ...
}
```

```
double global_result = 0.0; // shared variable

#pragma omp parallel
{
    double h = (b - a) / n;

    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    double local_result = (f(local_a) + f(local_b)) / 2.0;
    for(int i = 1; i <= local_n-1; ++i) {
        double x_i = local_a + i*h;
        local_result += f(x_i);
    }
    local_result = h * local_result;

    global_result += local_result;
}
```


DANGER


WELCOME TO THE...
DANGER ZONE
DIYLOL.COM

A single line of source code is usually more than one instruction!



Execution Order: Variant A

Time	Thread 0	Thread 1
0	compute `local_result`	compute `local_result`
1	load `global_value=0` to register	
2	load `local_result=1` to register	
3	add registers	
4	store `global_value=1`	
5		load `global_result=1` to register
6		load `load_result=3` to register
7		add registers
8		store `global_value=4`

Final value: **`global_value = 4`**

Execution Order: Variant B

Time	Thread 0	Thread 1
0	compute `local_result`	compute `local_result`
1		load `global_result=0` to register
2		load `load_result=3` to register
3		add registers
4		store `global_value=3`
5	load `global_value=3` to register	
6	load `local_result=1` to register	
7	add registers	
8	store `global_value=4`	

Final value: **`global_value = 4`**

Execution Order: Variant C

Time	Thread 0	Thread 1
0	compute `local_result`	
1	load `global_value=0` to register	compute `local_result`
2	load `local_result=1` to register	load `global_result=0` to register
3	add registers	load `load_result=3` to register
4	store `global_value=1`	add registers
5		store `global_value=3`

Final value: **`global_value = 3`**

Execution Order: Variant D

Time	Thread 0	Thread 1
0		compute `local_result`
1	compute `local_result`	load `global_result=0` to register
2	load `global_value=0` to register	load `load_result=3` to register
3	load `local_result=1` to register	add registers
4	add registers	store `global_value=3`
5	store `global_value=1`	

Final value: **`global_value = 1`**

We have a data-race!

Race Condition

A race condition exists when the following is true:

1. one or more threads read the same data location
2. at least one of them is writing that data location

A block of code that causes a race condition is called a **critical section**.

Synchronization: `critical` directive

- ensures that threads have mutually-exclusive access to a block of code
- only one thread can enter a critical section
- effectively serializes execution of this block of code
- **all unnamed critical blocks in the same parallel block are treated as one**
- **expensive!**

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

```
#pragma omp critical
{
    global_result += local_result;
}
```

Synchronization: Named `critical` blocks

- Non overlapping critical sections can run in parallel
- Useful to minimize blocking if not needed

```
if(...) {
    // threads that go this way...
    #pragma omp critical(a)
    {
        // modify shared resource a
    }
} else {
    // ... don't block threads that
    // go this way.
    #pragma omp critical(b)
    {
        // modify shared resource b
    }
}
```

Synchronization: `atomic` directive

- atomic enables mutual exclusion for some simple operations
- these are converted into special hardware instructions if supported
- however, it only protects the read/update of the target location

```
#pragma omp parallel
{
    // compute my_result

    #pragma omp atomic
    x += my_result;
}
```

acceptable operations

- `x++`
- `x--`
- `++x`
- `--x`
- `x binop= expr`
- `x = x binop expr`
- `x = expr binop x`

where `binop` is one of

`+ * - / & ^ | << >>`

Synchronization: `atomic` directive

```
#pragma omp parallel
{
    #pragma omp atomic
    x += func(); // warning func() is not atomic!
}
```

Synchronization

```
#pragma omp parallel
{
    // initialize data in parallel

    // STOP! do not continue until all data is initialized!

    // process data in parallel
}
```

Synchronization: `barrier` directive

This directive synchronizes the threads in a team by causing them to wait until all of the other threads have reached this point in the code.

```
#pragma omp parallel
{
    // do something in parallel using your team of threads

    #pragma omp barrier
    // wait until all threads reach this point

    // continue computation in parallel
}
```

Measuring elapse walltime

```
double tstart = omp_get_wtime();
// do work
double duration = omp_get_wtime() - tstart;
```

```
#pragma omp parallel
{
    double tstart = omp_get_wtime();
    // do part 1

    #pragma omp barrier

    double part1_duration = omp_get_wtime() - tstart;

    // continue with part 2
}
```

Outline

Let's refactor our trapezoid example

Define a function which does the local sum in range [$\text{local}_a, \textit{local}_b$]

```
double local_sum(double local_a, double local_b, int local_n, double h) {
    double local_result = (f(local_a) + f(local_b)) / 2.0;
    for(int i = 1; i <= local_n-1; ++i) {
        double x_i = local_a + i*h;
        local_result += f(x_i);
    }
    local_result = h * local_result;
    return local_result;
}
```

Let's refactor our trapezoid example

```
double global_result = 0.0;

#pragma omp parallel
{
    double h = (b - a) / n;

    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    // what is wrong with this code?
    #pragma omp critical
    global_result += local_sum(local_a, local_b, local_n, h);
}
```

Use a local variable instead

```
double global_result = 0.0;

#pragma omp parallel
{
    double h = (b - a) / n;

    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    double local_result = local_sum(local_a, local_b, local_n, h);

    #pragma omp atomic
    global_result += local_result;
}
```

OpenMP `reduction` clause

- creates a private variable for each thread
- each thread works on private copy
- finally all thread results are accumulated using operator
- allowed operators: `+, -, *, &, |, ^, &&, ||, min, max`
- each operator has a default initialization value (e.g. 0 for addition, 1 for multiplication)

```
double global_result = 0.0;

#pragma omp parallel reduction(+:global_result)
{
    double h = (b - a) / n;
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    global_result += local_sum(local_a, local_b, local_n, h);
}
```

Outline

Work Sharing Constructs

- the parallel construct alone creates a Single Program Multiple Data (SPMD) program
- each thread executes the same code independently
- Work sharing constructs are used to split up the work
- They do NOT launch new threads!

- **`for`**-loop construct
- `section` construct
- `single` and `master` construct
- task construct (explained in another lecture)

Loop worksharing: for loop

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

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int local_a = tid * N / nthreads;
    int local_b = (tid+1) * N / nthreads;
    if(id == threads - 1) local_b = N;

    for(int i = local_a; i < local_b; i++) {
        a[i] = a[i] + b[i];
    }
}
```

Loop worksharing: for loop

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

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

Combining `parallel` and **`for`** construct

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

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

Loops that can be parallelized

Loops that can't be parallelized

```
for (;;) {
}
```

```
for (int i = 0; i < n; i++) {
    if( ... ) break;
    ...
}
```

Loops with data-dependencies

```
fibo[0] = fibo[1] = 1;
#pragma omp parallel for
for(int i = 2; i < n; i++) {
    fibo[i] = fibo[i-1] + fibo[i-2];
}
```

Compiles, but is **broken code**.

1. OpenMP compilers do not check for dependencies among iterations
2. Loops, in which the result of one or more iterations depends on other iterations cannot be parallelized.

Loop worksharing: `schedule` clause

static schedule

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < 1000; ++i) {
    work(i);
}
```

0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

Loop worksharing: `schedule` clause

static schedule with custom chunk size

```
#pragma omp parallel for schedule(static,4)
for(int i = 0; i < 16; ++i) {
    work(i);
}
```

0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

```
#pragma omp parallel for schedule(static,1)
for(int i = 0; i < 16; ++i) {
    work(i);
}
```

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Loop worksharing: `schedule` clause

`schedule(static[, chunk])`

Divide iteration space into block of size chunk. Each thread is given a static set of blocks to work on. If chunk isn't specified, the iteration space is evenly divided among all threads.

`schedule(dynamic[, chunk])`

Divide iteration space into block of size chunk. At runtime, each thread grabs the next available block from a queue.

`schedule(guided[, chunk])`

Threads grab blocks dynamically. The block size starts out large and shrinks down to size chunk.

`schedule(runtime)`

Schedule and chunk size are set by `OMP_SCHEDULE` environment variable

`nowait` clause

Work sharing constructs have a implict barrier at their end. With nowait you can allow them to continue after they finish their part.

```
#pragma omp parallel
{
    #pragma omp for
    for(...) {
    }
    // implicit barrier

    #pragma omp for
    for(...) {
    }
}
```

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(...) {
    }
    // threads can continue

    #pragma omp for
    for(...) {
    }
}
```

`sections` directive

- breaks work into separate, discrete sections. Each section is executed by a thread

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            // calculation A
        }
        #pragma omp section
        {
            // calculation B
        }
        #pragma omp section
        {
            // calculation C
        }
    }
}
```

Pipeline and Nested Parallelism

```
#pragma omp parallel sections
{
    #pragma omp section
    for(int i = 0; i < N; ++i){
        read_input(i);
        signal_read(i);
    }
    #pragma omp section
    for(int i = 0; i < N; ++i){
        wait_read(i);
        process_data(i);
        signal_processed(i);
    }
    #pragma omp section
    for(int i = 0; i < N; ++i){
        wait_processed(i);
        write_output(i);
    }
}
```

```
void process_data(int i) {
    #pragma omp parallel for num_threads(4)
    for(int j = 0: j < M; ++j){
        do_compute(i, j);
    }
}
```

- Note: nested parallelism has to be enabled in your OpenMP implementation
- While useful for simple cases, you should be looking into tasks!

`master` directive

```
#pragma omp parallel
{
    #pragma omp master
    {
        // only master thread should execute this
        // useful for I/O or initialization
        // there is NO implicit barrier!
    }

    // add explicit barrier if needed
    #pragma omp barrier
    ...
}
```

`single` directive

```
#pragma omp parallel
{
    #pragma omp single
    {
       // only one thread will execute this block
       // all others wait until it completes
       // implicit barrier!
    }
}
```

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
       // only one thread will execute this block
       // others will go right past it
    }
}
```

Outline

Reasons for poor performance

Load-Imbalance

Not distributing work elements evenly among threads. Some threads finish earlier and wait, while others have to do more work.

Cost of synchronization

Having too many points of synchronization effectively serializes your application and limits your total speedup.

False Sharing

Multiple threads modifying data in the same cache line. This leads to forced flushes of memory, which cost extra cycles.

Reasons for poor performance

Data Locality

The placement of data relative to where your thread is executed is important. E.g., accessing memory regions which were allocated on a different socket are slower to access. This is one of the main challenges of dealing with Non-Uniform Memory Access (NUMA) architectures.

Ineffective use of caches and memory bandwidth saturation

Not taking advantage of caches to reuse data by multiple threads. Not taking advantage of available memory channels due to bad memory allocation strategy.