

PHYS52015 Core Ib: Introduction to High Performance Computing (HPC)

Session III: OpenMP (2/3)

Christopher Marcotte

Michaelmas term 2023

Outline



Race Conditions
Thread communication
Barriers

pollev.com/christophermarcotte820

Race Conditions

- ▶ Race conditions arise from interdependency on data access across threads
- ▶ These result from dependencies (read-write, write-read, write-write) on a position in memory
- ⇒ These manifest as indeterminacy in your program, and are sensitive to the memory model of your machine and how many threads are used.

Race Conditions

- ▶ Race conditions arise from interdependency on data access across threads
- ▶ These result from dependencies (read-write, write-read, write-write) on a position in memory
- ⇒ These manifest as indeterminacy in your program, and are sensitive to the memory model of your machine and how many threads are used.

Example: write a short program to compute this sum manually,

$$\sum_{n=1}^N n = \frac{N(N+1)}{2}$$

for some (not too large) N .

Race condition example

```
#include <omp.h>
#include <stdio.h>
int main() {
    int sum=0;
    const int N=100;
    #pragma omp parallel for
    for (int n=1; n<N+1; n++){
        sum +=n;           // Race condition here
    }
    printf("Result is %d. It should be %d\n", sum, N*(N+1)/2);
    return 0;
}
```

- ▶ The variable `sum` is accessed for both reading and writing in parallel
 - ⇒ Thread 0 reads `sum` into (local) `sum`
 - ⇒ Thread 1 reads `sum` into (local) `sum`
 - ⇒ Thread 0 increments (local) `sum` by N/p
 - ⇒ Thread 1 increments (local) `sum` by N/p
 - ⇒ Thread 0 writes (local) `sum` to `sum`
 - ⇒ Thread 1 writes (local) `sum` to `sum`
- ▶ So long as the variables are initialised correctly, the race condition means you'll have $\text{sum} \leq N(N+1)/2$.

Concept of building block: Race Conditions

- ▶ Content
 - ▶ Race conditions
- ▶ Expected Learning Outcomes
 - ▶ The student can identify potential race conditions in an OpenMP parallel code
 - ▶ The student can ameliorate race conditions in simple loops using alternative OpenMP constructions

Thread communication

We distinguish two different communication types:

- ▶ Communication through the join
- ▶ Communication inside the BSP part

Critical section: Part of code that is ran by at most one thread at a time – a manual serialisation block.

Reduction: Join variant, where all the threads reduce a value into one single value.

- ⇒ Reduction maps a vector of $(x_0, x_1, x_2, x_3, \dots, x_{p-1})$ onto one value x , i.e. we have an all-to-one data flow
- ⇒ Inter-thread communication realised by data exchange through shared memory
- ⇒ Fork can be read as one-to-all information propagation (done implicitly by shared memory)

Critical sections

```
#pragma omp critical (mycriticalsection)
{
    x *= 2;
}
...
#pragma omp critical (anothersection)
{
    x *= 2;
}
...
#pragma omp critical (mycriticalsection)
{
    x /= 2;
}
```

- ▶ Name is optional (default name i.e. does not block with other sections with a name)
- ▶ Single point of exit policy \Rightarrow return, break, ... not allowed within critical section
- ▶ For operations on built-in/primitive data types, an *atomic operation* is usually the better, i.e. faster choice

critical vs. single regions

- ▶ critical sections specify that code is executed by **one thread at a time**
- ▶ single sections specify that a section of code should be executed **by a single thread** (not necessarily the manager thread)

```
for (int i=0; i<size; i++){  
    a[i] = i;  
}  
#pragma omp critical  
{  
    for (int i=0; i<size; i++){  
        a[i] = a[i]*2;  
    }  
}  
#pragma omp single  
{  
    for (int i=0; i<size; i++){  
        a[i] = a[i]/2;  
    }  
}
```

With $p = 1$ threads, will $a[i] == i$? With $p > 1$ threads, will $a[i] == i$?

Case study: scalar product

Serial starting point:

```
double result = 0;
for( int i=0; i<size; i++ ) {
    result += a[i] * b[i];
}
```

A parallel variant:

```
double result = 0.0;
#pragma omp parallel
{
    double myResult = 0.0;
    #pragma omp for
    for( int i=0; i<size; i++ ) {
        myResult += a[i] * b[i];
    }
    #pragma omp critical
    result += myResult;
}
```

Case study: scalar product

A parallel variant:

```
double result = 0.0;
#pragma omp parallel
{
    double myResult = 0.0;
    #pragma omp for
    for( int i=0; i<size; i++ ) {
        myResult += a[i] * b[i];
    }
    #pragma omp critical
    result += myResult;
}
```

Observations:

- ▶ Avoid excessive synchronisation
- ▶ Type of operation is called *reduction* (as defined before)
- ▶ We may not use `result` to accumulate because of races
- ▶ We may not hide `result` as we then lose access to outer variable

Recap: Requirements for parallel loops

```
#pragma omp parallel for
{
    for( int i=0; i<size; i++ ) {
        a[i] = a[i]*2;
    }
}
```

- ▶ Loop has to follow plain initialisation-condition-increment pattern:
 - ▶ Only integer counters
 - ▶ Only plain comparisons
 - ▶ Only increment and decrement (no multiplication or any arithmetics)
- ▶ Loop has to be countable (otherwise splitting is doomed to fail).
- ▶ Loop has to follow single-entry/single-exit pattern.
- ▶ Loop copies all share the memory.

Consistency observation:

- ▶ All attributes are shared
- ▶ Besides the actual loop counter (otherwise splitting wouldn't work)
- ⇒ There has to be support for non-shared data – important for memory consistency!

The shared default clause

```
double result = 0;
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
    result += a[i] * b[i]; // race, don't worry about it here
}
```

```
double result = 0;
#pragma omp parallel for shared(result)
for( int i=0; i<size; i++ ) {
    result += a[i] * b[i]; // race, don't worry about it here
}
```

- ▶ By default, all OpenMP threads share all variables, i.e. variables declared outside are visible to threads
- ▶ This sharing can be made explicit through the clause `shared`
- ▶ Explicit shared annotation is good practice (improved readability)

Thread-local variables

```
double result = 0;
for( int i=0; i<size; i++ ) {
    double result = 0.0;
    result += a[i] * b[i];
}
```

Without OpenMP pragma:

- ▶ C/C++ allows us to “redefine” variable in inner scope
- ▶ Hides/shadows outer `result`
- ▶ We may not forget the second initialisation; otherwise garbage

Thread-local variables

```
double result = 0;
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
    double result = 0.0;
    result += a[i] * b[i];
}
```

With OpenMP pragma:

- ▶ OpenMP introduces (concurrent) scope of its own
- ▶ Scope-local variables are thread-local variables
- ▶ These are *not* shared – *private variables in local scope*

shared **vs.** private clauses

```
double result = 0;
#pragma omp parallel for private(result)
for( int i=0; i<size; i++ ) {
    result += a[i] * b[i];
}
```

- ▶ private is the counterpart of shared, i.e. each thread works on its own copy
- ▶ Basically, the copying is similar to the following fragment:

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


shared **vs.** private **clauses**

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

- ⇒ In this example, `result` within thread is not initialised (garbage)!
- ⇒ In this example, data within `result` is lost throughout join!
- This code will not do what we want (the inner product of `a` & `b`)
- ⇒ We will first look at other copy policies and then come back to properly parallelise the inner product calculation a few different ways next time!

default(none) and scoping

```
double result = 0;
#pragma omp parallel for default(none) private(result) shared(a,b,size)
for( int i=0; i<size; i++ ) {
    result += a[i] * b[i];
}
```

- ▶ Using `default(none)` tells the compiler that, by default, the privacy/sharing of all variables is *unspecified*.
- ⇒ This means the programmer is responsible for all explicit sharing.
- ▶ This can dramatically improve legibility of your code when using a large number of temporary variables.

Example

```
int x;  
x=40;  
#pragma omp parallel for private (x)  
for (int i=0; i<10; i++) {  
    x = i; // try to remove this line  
    std::cout << "x=" << x << " on thread "  
                << omp_get_thread_num() << std::endl;  
}  
std::cout << "x=" << x << std::endl;
```

Example

```
int x;  
x=40;  
#pragma omp parallel for private (x)  
for (int i=0; i<10; i++) {  
    x = i; // try to remove this line  
    std::cout << "x=" << x << " on thread "  
               << omp_get_thread_num() << std::endl;  
}  
std::cout << "x=" << x << std::endl;
```

Observations:

- ▶ If we comment out `x=i`, `x` is not properly initialised.
- ▶ `x` in the last line always equals 40.

Now remove initialisation and change `private(x)` to `firstprivate(x)`.

Does the final value of `x` change?

What if we use `lastprivate(x)`? What value is `x` on the last line?

Copy policies

- ▶ `default` Specifies default visibility of variables
- ▶ `firstprivate` Variable is private, but initialised with value from surrounding
- ▶ `lastprivate` Variable is private, but value of very last iteration is copied over to outer variable

Concept of building block

- ▶ Content
 - ▶ Introduce three types of data flow/usage of shared memory: making memory available to all threads throughout fork, sharing data throughout computation, reducing data at termination
 - ▶ Introduce private variables
 - ▶ Study semantics and usage of critical sections
- ▶ Expected Learning Outcomes
 - ▶ The student *can use* critical sections
 - ▶ The student *knows difference* between private and shared variables
 - ▶ The student can *identify race conditions* and *resolve* them through critical sections for given code snippet

Barriers

- ▶ Barriers are synchronisation points in your code
- ▶ Places where each threads waits for all other threads to arrive at the barrier before proceeding
- ▶ Lots of implicit barriers when using default constructs in OpenMP

Barriers

- ▶ Barriers are synchronisation points in your code
- ▶ Places where each threads waits for all other threads to arrive at the barrier before proceeding
- ▶ Lots of implicit barriers when using default constructs in OpenMP

```
#pragma omp parallel
{
    // implicit barrier here
#pragma omp for
for (int i = 0; i < size; i++){
    a[i] = a[i]*2;
}
    // implicit barrier here
}
```


Explicit Barriers

```
// ...  
    #pragma omp parallel  
    {  
        printf("Thread %d prints 1\n", omp_get_thread_num());  
        printf("Thread %d prints 2\n", omp_get_thread_num());  
    }  
// ...
```

What does the output of this program with 4 threads look like?

Explicit Barriers

```
// ...  
    #pragma omp parallel  
    {  
        printf("Thread %d prints 1\n", omp_get_thread_num());  
        printf("Thread %d prints 2\n", omp_get_thread_num());  
    }  
// ...
```

What does the output of this program with 4 threads look like?

```
Thread 0 prints 1  
Thread 0 prints 2  
Thread 1 prints 1  
Thread 1 prints 2  
Thread 3 prints 1  
Thread 3 prints 2  
Thread 2 prints 1  
Thread 2 prints 2
```

Explicit Barriers

```
// ...  
    #pragma omp parallel  
    {  
        printf("Thread %d prints 1\n", omp_get_thread_num());  
        #pragma omp barrier  
        printf("Thread %d prints 2\n", omp_get_thread_num());  
    }  
// ...
```

What does the output of this program with 4 threads look like?

Explicit Barriers

```
// ...  
    #pragma omp parallel  
    {  
        printf("Thread %d prints 1\n", omp_get_thread_num());  
        #pragma omp barrier  
        printf("Thread %d prints 2\n", omp_get_thread_num());  
    }  
// ...
```

What does the output of this program with 4 threads look like?

```
Thread 0 prints 1  
Thread 3 prints 1  
Thread 2 prints 1  
Thread 1 prints 1  
Thread 3 prints 2  
Thread 1 prints 2  
Thread 2 prints 2  
Thread 0 prints 2
```

Barriers and `nowait` clause

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

- The implicit barriers here mean *all threads complete the first loop before starting the second loop*
- ⇒ What if we want the second loop to run as soon as there are threads available to do so?

Barriers and `nowait` clause

```
// ...  
#pragma omp parallel  
{  
    #pragma omp for nowait  
    for ... {  
    } // no more implicit barrier here  
    #pragma omp for  
    for ... {  
    } // still an implicit barrier here  
}  
// ...
```

- ▶ Adding the `nowait` clause to the first loop means that the threads will immediately progress to the second loop.
- ⇒ Since the `#pragma omp for` splits the range, the progressing threads will not necessarily work on the same range values in both loops, this can be quite dangerous!

Concept of building block: Barriers

- ▶ Content
 - ▶ Barriers
- ▶ Expected Learning Outcomes
 - ▶ The student can **identify** implicit synchronisation points in an OpenMP parallel code
 - ▶ The student can **add** explicit synchronisation points in an OpenMP parallel code