Parallel Programming on Multicore Computers with OpenMP

Virtual Summer Training Program

Alexey Fedoseev

August 16 - 20, 2021

Concurrency vs Parallelism

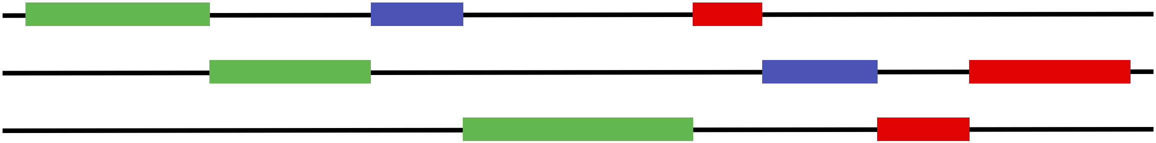


Figure 1: Concurrent, non-parallel execution

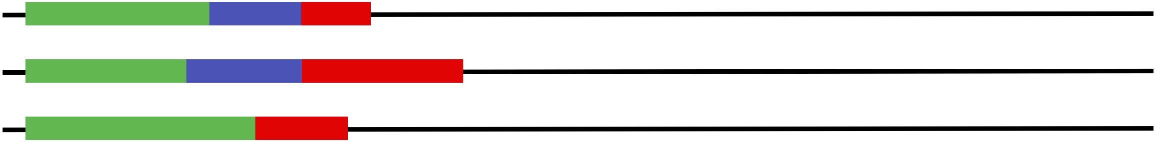
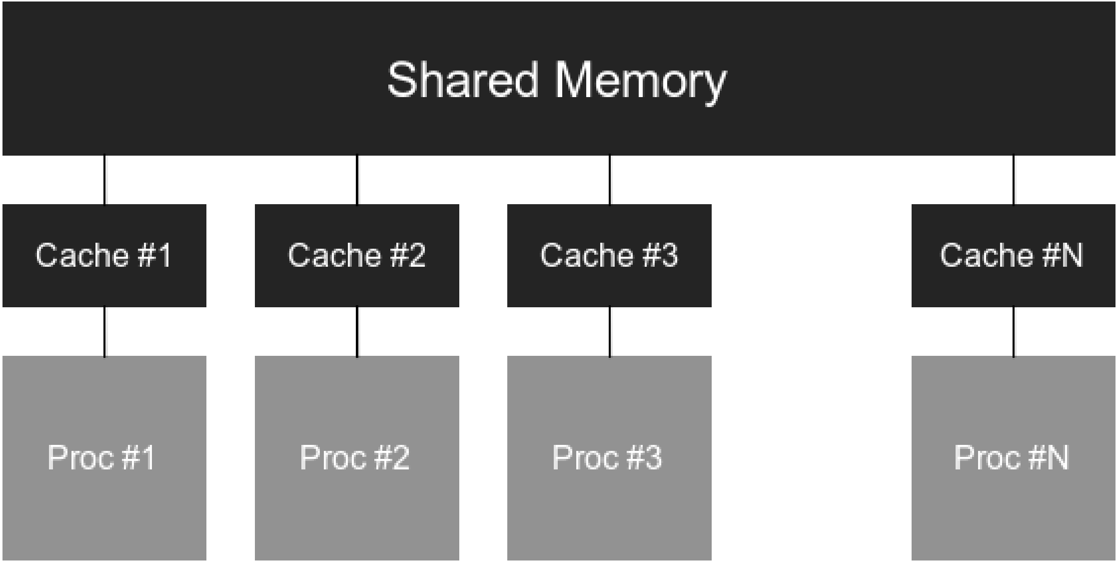


Figure 2: Concurrent, parallel execution

Shared Memory Computer



Shared Memory Computer

Symmetric Multiprocessor (SMP)

A shared address space where each processor has equal memory access time and OS treats all processors equally, reserving none for special purposes.

Non-Uniform Memory Access (NUMA)

A shared address space where memory access time depends on the memory location relative to the processor.

OpenMP

- Provides a set of compiler directives and library routines that used together to write multi-threaded applications
- Simplifies writing multi-threaded programs in C, C++ and Fortran
- Most of the constructs in OpenMP are compiler directives.

```
#pragma omp parallel num_threads(4)
```

Running OpenMP on Teach cluster

- Connect to the Teach login node

```
$ ssh username@teach.scinet.utoronto.ca
```

Now you are on the login node `teach01`. This node is shared between students. Use this node to develop and compile code, to run short tests, and to submit computations to the scheduler.

- Request the part of the cluster resources

```
$ debugjob -n 4
```

- Load the compiler

```
$ module load gcc
```

Running OpenMP on OS X

- Install Homebrew from https://brew.sh/
- Install `gcc` using `brew`

```
$ brew install gcc
```

- Use `gcc-11` instead of `gcc`

```
$ which gcc-11
/usr/local/bin/gcc-11
```

- To compile the code using OpenMP add `-fopenmp`

```
$ gcc-11 -fopenmp program.c
```

Exercise 1: Hello World

```c
#include <stdio.h>
int main()
{
    int ID = 0;
    printf("hello(%d) ", ID);
    printf("world(%d) \n", ID);
    return 0;
}
```

```
$ gcc hello-world.c
```

Exercise 1: Hello World - Parallel version

```
#include <stdio.h>
#include <omp.h>
int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d) ", ID);
        printf("world(%d) \n", ID);
    }
    return 0;
}
```

```
$ gcc -fopenmp hello-world.c
```

Exercise 1: Hello World - Parallel version

```
$ ./a.out
hello(2) hello(1) hello(0) hello(3) world(2)
world(1)
world(0)
world(3)
```

Fork-Join

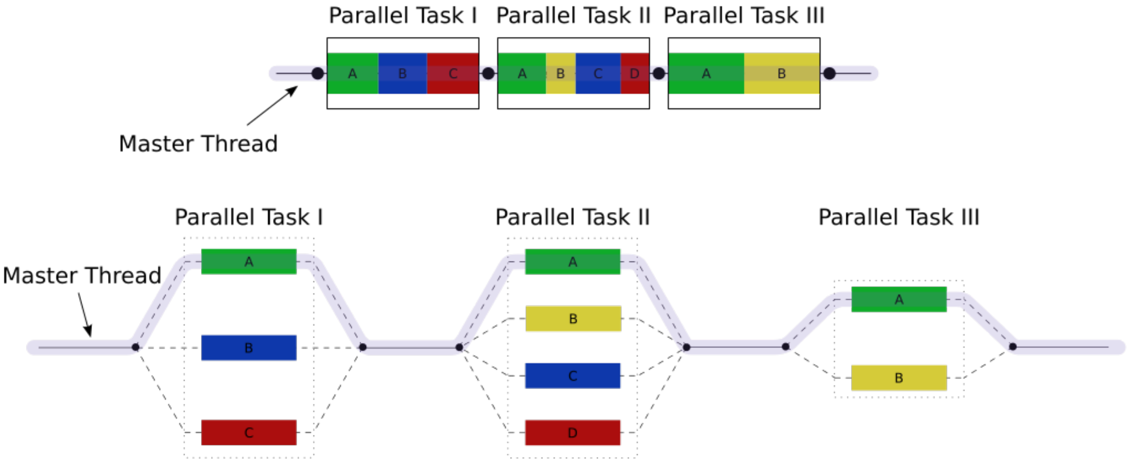


Figure 3: Fork-join model on Wikipedia

Requesting global number of threads

```
#include <stdio.h>
#include <omp.h>
int main() {
    omp_set_num_threads(8);
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int n_threads = omp_get_num_threads();
        if (thread_id == 0) printf("There are %d threads\n", n_threads);
    }
    return 0;
}
```

OMP_NUM_THREADS environmental variable

```
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int n_threads = omp_get_num_threads();
        if (thread_id == 0) printf("There are %d threads\n", n_threads);
    }
    return 0;
}
```

```
$ export OMP_NUM_THREADS=8
$ ./a.out
There are 8 threads
```

Requesting local number of threads

```c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel num_threads(8)
    {
        int thread_id = omp_get_thread_num();
        int n_threads = omp_get_num_threads();
        if (thread_id == 0) printf("There are %d threads\n", n_threads);
    }
    return 0;
}
```

```
$ export OMP_NUM_THREADS=16
$ ./a.out
There are 8 threads
```

False sharing - example

```
#include <stdio.h>
#include <omp.h>
int main() {
    int *x = new int[100];
    #pragma omp parallel
    {
        int i = omp_get_thread_num(),
                   stride = 16;
        for (int k = 0; k < 2000000000; k++)
            x[i*stride]++;
    }
    delete [] x;
    return 0;
}
```

False sharing - example

- Stride = 1

```
$ time ./a.out

real    0m22.894s
user    1m24.759s
sys     0m0.134s
```

- Stride = 16

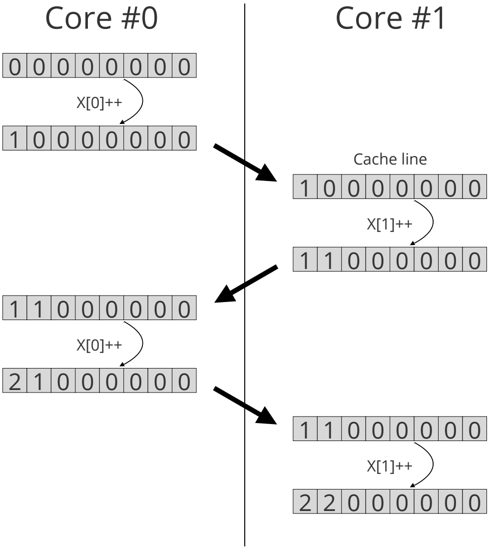
```
$ time ./a.out

real    0m6.200s
user    0m22.300s
sys     0m0.039s
```

Figure 4: MacBook Pro (Retina, 13-inch, Early 2015), no optimization

On newer versions of OS X to achieve the same formatting of the `time` command use the following command instead:

```
/usr/bin/time -p ./a.out
```

False sharing



Explanation

False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces a memory update.

Synchronization

High level synchronization

- `critical`

  A section of code can only be executed by one thread at a time.

- `atomic`

  Update of a single memory location.

- `barrier`

  A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point.

Synchronization - critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```
double sum = 0;
#pragma omp parallel
{
    int id = omp_get_thread_num();
    #pragma omp critical
    sum += work(id);
}
```

Synchronization - `atomic`

- An atomic operation applies only to the single assignment statement that immediately follows it. It is commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously.

```
double sum = 0;
#pragma omp parallel
{
    int id = omp_get_thread_num();
    #pragma omp atomic
    sum += work(id);
}
```

Synchronization - barrier

- Each tread waits until all threads arrive

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    var[id] = work(id);
    #pragma omp barrier
    res[id] = calc(id, var);
}
```

single work sharing construct

- The single construct denotes a block of code that is executed by only one thread.
- A barrier is implied at the end of the single block (can remove the barrier with a nowait clause).

```
#pragma omp parallel
{
    do_work();

    #pragma omp single
    exchange_boundaries();

    do_more_work();
}
```

master construct

- The master construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
    do_work();
    #pragma omp master
    exchange_boundaries();
    #pragma omp barrier
    do_more_work();
}
```

Parallel for loop

```c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel for
    for (int i = 0; i < 4*omp_get_num_threads(); i++)
        printf("Thread %d, i = %d\n",
            omp_get_thread_num(), i);
    return 0;
}
```

```
$ gcc -fopenmp par-for.c
```

Parallel `for` loop

- Output

```
$ ./a.out
Thread 0, i = 0
Thread 2, i = 6
Thread 1, i = 3
Thread 3, i = 8
Thread 0, i = 1
Thread 2, i = 7
Thread 1, i = 4
Thread 3, i = 9
Thread 0, i = 2
Thread 1, i = 5
```

Parallel for loop

```c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel num_threads(3)
    {
        #pragma omp for
        for (int i = 0; i < 10; i++)
            printf("Thread %d, i = %d\n",
                omp_get_thread_num(), i);
    }
    return 0;
}
```

```
$ gcc -fopenmp specify-num-threads.c
```

Parallel `for` loop

- Output with 4 threads

```
$ ./a.out
Thread 0, i = 0
Thread 2, i = 6
Thread 1, i = 3
Thread 3, i = 8
Thread 0, i = 1
Thread 2, i = 7
Thread 1, i = 4
Thread 3, i = 9
Thread 0, i = 2
Thread 1, i = 5
```

- Output with 3 threads

```
$ ./a.out
Thread 1, i = 4
Thread 2, i = 7
Thread 0, i = 0
Thread 1, i = 5
Thread 2, i = 8
Thread 0, i = 1
Thread 1, i = 6
Thread 2, i = 9
Thread 0, i = 2
Thread 0, i = 3
```

Nested for loops - the collapse clause

```c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel for collapse(2)
    for (int x = -1; x <= 1; x+=1)
        for (int y = -1; y <= 1; y+=1)
            printf("Thread %d: (%d, %d)\n",
                omp_get_thread_num(), x, y);
    return 0;
}
```

```
$ gcc -fopenmp for-collapse.c
```

Nested for loops - the collapse clause

- Output

```
$ ./a.out
Thread 1: (0, -1)
Thread 2: (0, 1)
Thread 0: (-1, -1)
Thread 3: (1, 0)
Thread 1: (0, 0)
Thread 2: (1, -1)
Thread 0: (-1, 0)
Thread 3: (1, 1)
Thread 0: (-1, 1)
```

The reduction clause

```c
#include <stdio.h>
#include <math.h>
#include <omp.h>
#define N 1000000000
int main() {
    double calc = 0;
    #pragma omp parallel for reduction(+:calc)
    for (long i = 0; i < N; i++)
        calc += pow(-1,i) * 1.0/(2*i + 1);
    printf("%.12f\n", 4*calc); return 0;
}
```

```
$ gcc -fopenmp for-reduction.c
```

The `reduction` clause

- Parallel output

```
$ time ./a.out
3.141592652589

real	0m5.440s
user	0m19.835s
sys	0m0.038s
```

- Serial output

```
$ time ./a.out
3.141592652588

real	0m12.562s
user	0m12.413s
sys	0m0.026s
```

The reduction clause

<table>
<tr><th>Operator</th><th>Initial value</th></tr>
<tr><td>+</td><td>0</td></tr>
<tr><td>*</td><td>1</td></tr>
<tr><td>-</td><td>0</td></tr>
<tr><td>min</td><td>Largest positive number</td></tr>
<tr><td>max</td><td>Most negative number</td></tr>
<tr><td>& (bitwise AND)</td><td>~0 (all bits are 1)</td></tr>
<tr><td>| (bitwise OR)</td><td>0</td></tr>
<tr><td>^ (bitwise XOR)</td><td>0</td></tr>
<tr><td>&& (logical AND)</td><td>1</td></tr>
<tr><td>|| (logical OR)</td><td>0</td></tr>
</table>

Scheduling

`static`

Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is `loop_count/number_of_threads`.

`dynamic`

Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Involves extra overhead.

Scheduling - static

```c
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main() {
    #pragma omp parallel for schedule(static, 1) num_threads(10)
    for (int i = 0; i < 20; i++) {
        sleep(i);
        printf("Thread %d: iteration %d\n",
            omp_get_thread_num(), i);
    }
    return 0;
}
```

```
$ gcc -fopenmp static-schedule.c
```

Scheduling - `static`

- Default chunk size

```
$ time ./a.out
Thread 0: iteration 0
Thread 0: iteration 1
...
Thread 8: iteration 17
Thread 9: iteration 19

real    0m37.018s
user    0m0.002s
sys     0m0.005s
```

- Chunk size = 1

```
$ time ./a.out
Thread 0: iteration 0
Thread 1: iteration 1
...
Thread 8: iteration 18
Thread 9: iteration 19

real    0m28.009s
user    0m0.002s
sys     0m0.004s
```

Scheduling - dynamic

```c
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main() {
    #pragma omp parallel for schedule(dynamic, 1) num_threads(10)
    for (int i = 0; i < 20; i++) {
        sleep(i);
        printf("Thread %d: iteration %d\n",
            omp_get_thread_num(), i);
    }
    return 0;
}
```

```
$ gcc -fopenmp dynamic-schedule.c
```

Scheduling - `dynamic`

- Default chunk size

```
$ time ./a.out
Thread 3: iteration 0
Thread 1: iteration 1
...
Thread 9: iteration 18
Thread 0: iteration 19

real    0m28.013s
user    0m0.003s
sys     0m0.004s
```

- Chunk size = 2

```
$ time ./a.out
Thread 5: iteration 0
Thread 5: iteration 1
...
Thread 0: iteration 17
Thread 9: iteration 19

real    0m37.012s
user    0m0.002s
sys     0m0.004s
```

Data sharing

Shared data

The data defined outside of a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.

```
int x = 10;
#pragma omp parallel
{
    x++;
    printf("shared x is %d\n", x);
}
```

Shared data

```
$ gcc -fopenmp shared-data.c && ./a.out
shared x is 12
shared x is 11
shared x is 13
shared x is 14
```

Attention!

All threads increment the same variable, so after the loop it will have a value of 10 plus the number of threads; or maybe less because of the data races involved.

Data sharing

Private data

The data defined within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.

```
int x = 10;
#pragma omp parallel
{
    int x; x = 5;
    printf("private x is %d\n", x);
}
printf("shared x is %d\n", x);
```

Private data

```
$ gcc -fopenmp private-data.c && ./a.out
private x is 5
private x is 5
private x is 5
private x is 5
shared x is 10
```

Attention!

Stack variables in functions called from parallel regions are private.

Data Sharing Attribute Clauses

Some OpenMP clauses enable you to specify visibility context for selected data variables.

Attribute clause	Description
`private`	The `private` clause declares the variables in the list to be private to each thread in a team.
`firstprivate`	The `firstprivate` clause provides a superset of the functionality provided by the `private` clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
`lastprivate`	The `lastprivate` clause provides a superset of the functionality provided by the `private` clause. The final value of a private variable is transmitted to the shared variable outside the parallel construct.

Data Sharing Attribute Clauses

Attribute clause	Description
`shared`	The `shared` clause declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables.
`reduction`	The `reduction` clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
`default`	The `default` clause allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

Data sharing - private clause

```c
int x = 10;
#pragma omp parallel private(x)
{
    x = 1;
    printf("Inside x is %d\n", x);
}
printf("Outside x is %d\n", x);
```

```
$ gcc -fopenmp private-clause.c && ./a.out
Inside x is 1
Inside x is 1
Inside x is 1
Inside x is 1
Outside x is 10
```

Data sharing - firstprivate clause

```c
int x = 10;
#pragma omp parallel firstprivate(x)
{
    printf("Inside x is %d\n", x);
}
printf("Outside x is %d\n", x);
```

```
$ gcc -fopenmp first-private-clause.c && ./a.out
Inside x is 10
Inside x is 10
Inside x is 10
Inside x is 10
Outside x is 10
```

Data sharing - lastprivate clause

```c
int x = 10;
#pragma omp parallel for lastprivate(x)
for (int i = 0; i < 4; i++) {
    x = i;
    printf("Inside x is %d\n", x);
}
printf("Outside x is %d\n", x);
```

```
$ gcc -fopenmp last-private-clause.c && ./a.out
Inside x is 1
Inside x is 0
Inside x is 2
Inside x is 3
Outside x is 3
```

Data sharing - default clause

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int arr[1000], x = 10;
    #pragma omp parallel default(none)
    {
        x = 1; arr[0] = 2;
        printf("Inside x is %d and arr[0] is %d\n",
            x, arr[0]);
    }
    printf("Outside x is %d and arr[0] is %d\n",
        x, arr[0]);
    return 0;
}
```

Data sharing - default clause

```
$ gcc -fopenmp default-clause.c
default-clause.c: In function 'main':
default-clause.c:7:5: error: 'x' not specified in enclosing 'parallel'
   x = 1; arr[0] = 2;
   ~~^~~
default-clause.c:5:10: error: enclosing 'parallel'
  #pragma omp parallel default(none)
          ^~~
default-clause.c:7:13: error: 'arr' not specified in enclosing 'parallel'
   x = 1; arr[0] = 2;
          ~~~^~~
default-clause.c:5:10: error: enclosing 'parallel'
  #pragma omp parallel default(none)
          ^~~
```

Let's fix it.

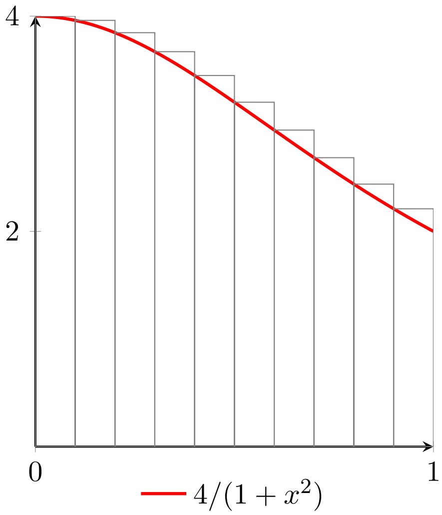
Data sharing - default clause

```
#include <stdio.h>
#include <omp.h>
int main() {
    int arr[1000], x = 10;
    #pragma omp parallel default(none) private(x) shared(arr)
    {
        x = 1; arr[0] = 2;
        printf("Inside x is %d and arr[0] is %d\n",
            x, arr[0]);
    }
    printf("Outside x is %d and arr[0] is %d\n",
        x, arr[0]);
    return 0;
}
```

Data sharing - default clause

```
$ gcc -fopenmp default-clause.c && ./a.out
Inside x is 1 and arr[0] is 2
Inside x is 1 and arr[0] is 2
Inside x is 1 and arr[0] is 2
Inside x is 1 and arr[0] is 2
Outside x is 10 and arr[0] is 2
```

Exercise 2: Numerical integration



Write a program that calculates the integral

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi.$$

Using the left Riemann sum we approximate the integral as follows

$$h\sum_{i=1}^{N} \frac{4}{1+x_i^2} \approx \pi,$$

where $x_i = ih$, $h = 1/N$.

Exercise 2: Using data sharing and reduction

The following solution is a serial code. Make it parallel using the knowledge of data sharing and reduction constructs.

```
#include <stdio.h>
#define N 1000000000
int main() {
    double h = 1.0/N, sum = 0.0, x, pi;
    for (long i = 0; i < N; i++) {
        x = i*h;
        sum += 4.0 / (1.0 + x*x);
    }
    pi = h * sum;
    printf("%.12f\n", pi);
    return 0;
}
```