Introduction to OpenMP

Alexey Fedoseev

March 4, 2024

Concurrency vs Parallelism

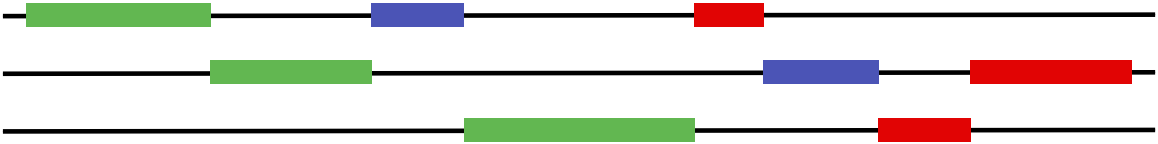


Figure 1: Concurrent, non-parallel execution

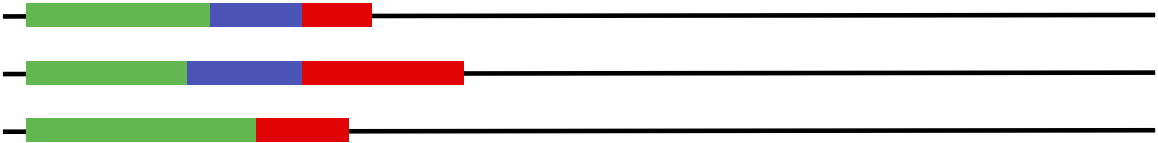
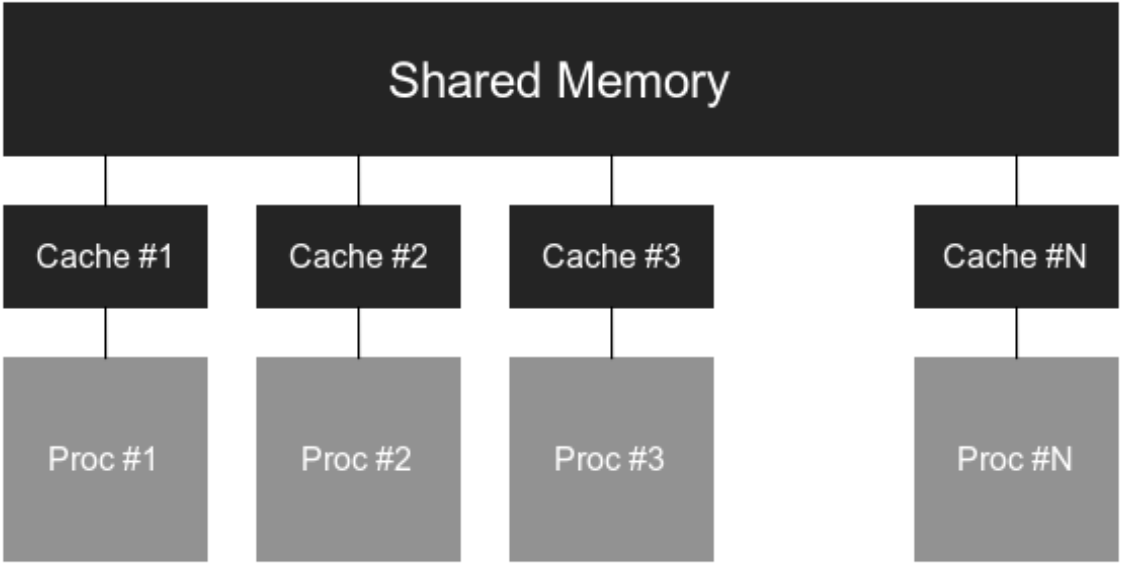


Figure 2: Concurrent, parallel execution

Shared Memory Computer



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)
```

Example 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
```

Example 1: Hello World - Parallel version

```c
#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
```

Example 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

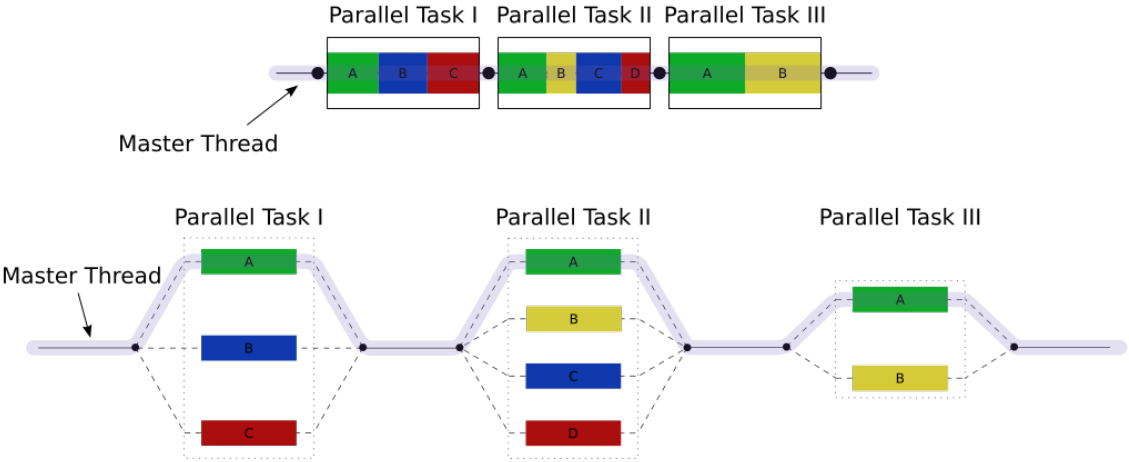


Figure 3: Fork-join model on Wikipedia

Requesting global number of threads

```c
#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

```c
#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
```

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
```

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

Operator	Initial value
`+`	0
`*`	1
`-`	0
`min`	Largest positive number
`max`	Most negative number
`&` (bitwise AND)	$\sim$0 (all bits are 1)
`|` (bitwise OR)	0
`^` (bitwise XOR)	0
`&&` (logical AND)	1
`||` (logical OR)	0

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

```
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 - 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.

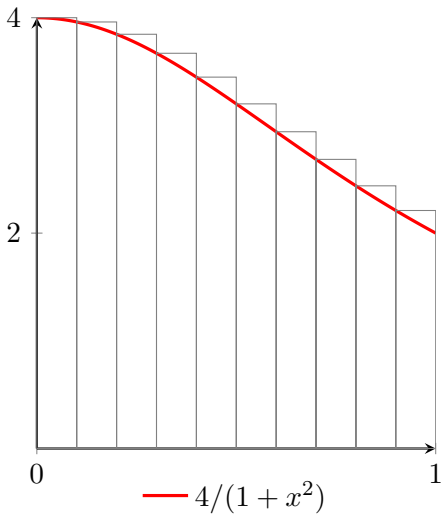
Data sharing - default clause

```c
#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
```

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

Example 2: Using data sharing and reduction

The following code is serial. We will use the knowledge of OpenMP to parallelize it with minimal changes to the code.

```
#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;
}
```

Example 2: Reductions

```
#include <stdio.h>
#include <omp.h>
#define N 1000000000
int main() {
    double h = 1.0/N, sum = 0.0, x, pi;
    #pragma omp parallel for \
        firstprivate(h) private(x) reduction(+:sum)
    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;
}
```

Example 2: Performance

Table 1: MacBook Pro (Retina, 13-inch, Early 2015)

	serial	**OpenMP**
1 thread	10.853s	10.689s
2 threads	9.435s	5.365s
3 threads	7.474s	3.637s
4 threads	6.129s	2.811s