Introduction to OpenMP

A guide by WITS HPC

Unleashing parallel power

We are limited by single processor chips:

- As data sizes and processing demands grow, single processors struggle to keep up.
- Sequential programs execute instructions one after another, leading to bottlenecks

Enter OpenMP

What is parallel computing?

- **Breakdown**
  Parallel programming breaks down large tasks into smaller, independent subtasks.
- **Execution**
  These subtasks can be executed simultaneously on multiple processors (cores) or computers.

OpenMP: The Message Passing Maestro

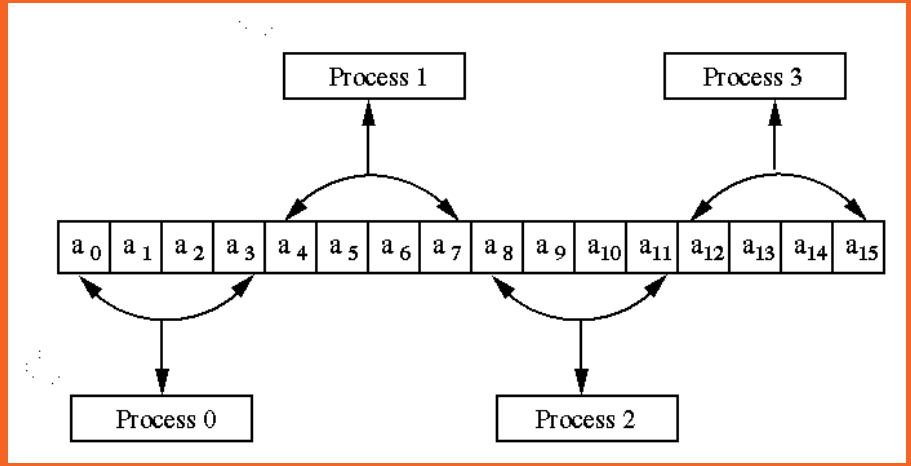
What even is OpenMPI?

- **What**
  Open Message Passing Interface (OpenMPI) is a free, open-source library for parallel programming.
- **How**
  It provides a standardized set of functions for processes to communicate and exchange data.

So how does it work?

- **Create**
  OpenMPI creates multiple processes (copies of your program).
- **Communicate**
  Processes reside on different processors or computers and communicate by sending and receiving messages.
- **Partitioned**

  Data is partitioned and distributed among processes.



—

#pragma omp parallel

- Creates a team of threads for parallel execution of the following code block.

#pragma omp for

- Distributes iterations of a loop across threads for parallel execution.

#pragma omp sections

- Defines multiple independent sections of code to be executed by different threads.

#pragma omp single

- Specifies a code block to be executed by only one thread.

#pragma omp master

- Specifies a code block to be

MP Directives.

All MPI sections start with

#pragma omp <construct> <cluases>

{

//code block to run in parallel

}

Some MP clauses.

These give us more control over the program.

private(list)
- Creates a separate copy of listed variables for each thread.

shared(list)
- Makes listed variables accessible by all threads.

firstprivate(list)
- Like private, but initializes each copy with the value of the original variable.

reduction(operator: list)
- Performs a reduction operation on variables at the end of the parallel region (e.g., +, *, max, min).

nowait
- Removes the implicit barrier at the end of a parallel construct, allowing threads to continue without waiting.

Story for illustration purposes only

Basic hello world program

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

using namespace std;

int main(){
        int num_threads = omp_get_num_threads(); //used to get the number of threads avaliable
        cout << "We have a total of: " << num_threads << " threads avaliable!" << endl;
        cout << "We only have one since we are running outside of a omp parallel region" << endl;
        #pragma omp parallel //define an omp parallel region
        {

                int ID = omp_get_thread_num();
                cout << "Hello from thread: " << ID << endl; //race condition!

        }

        return 0;


}
```

To compile :

g++ -fopenmp *<filename.cpp>* -o *execname*

```cpp
#include <iostream>

using namespace std;

static long num_steps = 1000000;
double step;


int main(){
        int i;
        double x, pi, sum = 0.0;

        step = 1.0/( double ) num_steps ;

        for(i=0; i < num_steps; ++i){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }

        pi = step * sum;
        cout << "sequential: pi with " << num_steps << " is: " << pi << endl;
        return 0;
}
```

Concrete example of why MPI is better

Here is a simple program to find find the sequential sum

The serial runs in:

```
anandpatel@pop-os:~/University/PC/week-7$ time ./pi_serial_intel
sequential: pi with 1000000 is: 3.14159

real    0m0.011s
user    0m0.007s
sys     0m0.004s
```

Whereas in parallel we get:

```
anandpatel@pop-os:~/University/PC/week-7$ ./pi_par_intel
 running on 1 threads: PI = 3.141592653589903 computed in 0.003019 seconds
 running on 2 threads: PI = 3.141592653589862 computed in 0.0017 seconds
 running on 3 threads: PI = 3.141592653589884 computed in 0.0011 seconds
 running on 4 threads: PI = 3.141592653589873 computed in 0.0008969 seconds
 running on 5 threads: PI = 3.141592653589874 computed in 0.001328 seconds
 running on 6 threads: PI = 3.141592653589876 computed in 0.001716 seconds
 running on 7 threads: PI = 3.141592653589876 computed in 0.001669 seconds
 running on 8 threads: PI = 3.141592653589878 computed in 0.001725 seconds
```

Challenge:

- Parallelise the code given for computing pi in serial

```
#include <iostream>

using namespace std;

static long num_steps = 1000000;
double step;


int main(){
        int i;
        double x, pi, sum = 0.0;

        step = 1.0/( double ) num_steps ;

        for(i=0; i < num_steps; ++i){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }

        pi = step * sum;
        cout << "sequential: pi with " << num_steps << " is: " << pi << endl;
        return 0;
}
```