# Getting started with parallel programming models

authors: Arias Rodrigo, Burca Horia

date: 15/10/2018

## Task 3.1

Question: Check the loaded modules in your environment.

This can be done using the `module list` command:

```
% module list
Currently Loaded Modules:
  1) intel/2017.4   2) impi/2017.4   3) mkl/2017.4   4) bsc/1.0
```

We can see that the modules related to the Intel compilers, the Intel MPI software stack, the math kernel libraries MKL and the BSC helpful scripts are loaded.

## Task 3.2

Question: Create, compile and run a Hello World program with MPI.

We use the following basic code that makes use of the MPI paradigm:

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

int main (int argc, char **argv)
{
    int size, rank;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf ( "I am %d of %d\n", rank, size );
    MPI_Finalize();

    return 0;
}
```

We compile and run using the following commands:

```
% mpicc mpi_helloworld.c -o mpi_helloworld

% mpirun ./mpi_helloworld
I am 0 of 48
I am 4 of 48
I am 15 of 48
I am 27 of 48
```

```
I am 28 of 48
I am 37 of 48
I am 38 of 48
I am 40 of 48
I am 41 of 48
I am 42 of 48
I am 46 of 48
I am 1 of 48
I am 2 of 48
...
I am 36 of 48
I am 20 of 48
I am 24 of 48
```

### Task 3.3

Question: Submit your "Hello World" program. Check the output of the execution. What happened with the order of outputs?

We use the following job script:

```
#!/bin/bash
#SBATCH --job-name="MPI_HelloWorld"
#SBATCH --workdir=.
#SBATCH --output=output_%J.out
#SBATCH --error=output_%J.err
#SBATCH --ntasks=16
#SBATCH --tasks-per-node=8
#SBATCH --time=00:01:00
#SBATCH --exclusive
#SBATCH --qos=debug
```

In order to submit we use `sbatch` and then `squeue` to inspect the queue:

```
% sbatch job

% squeue
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
       2500123      main MPI_Hell sam14015 PD       0:00      2 (Priority)
% squeue
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
       2500123      main MPI_Hell sam14015  R       0:05      2 s05r2b[64,66]
% squeue
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
       2500123      main MPI_Hell sam14015 CG       0:10      2 s05r2b[64,66]
% squeue
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
       2500123      main MPI_Hell sam14015 CG       0:10      1 s05r2b66
% squeue
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

If we take a look at the output we can notice that the execution is not sorted by rank:

```
I am 1 of 16
I am 2 of 16
I am 5 of 16
I am 6 of 16
I am 7 of 16
I am 0 of 16
I am 3 of 16
I am 4 of 16
I am 10 of 16
I am 9 of 16
I am 8 of 16
I am 15 of 16
I am 12 of 16
I am 11 of 16
I am 14 of 16
I am 13 of 16
```

## Task 3.4

Question: Modify your solution that just prints a line of output from each process so that the output is printed in process rank order: process 0 output first, then process 1, and so on.

In order to make the processes execute in order of their rank we make use of the methods `MPI_Send` and `MPI_Receive`. For each process with `rank > 0` we expect to receive something from the process with `rank - 1`. And in turn, for each process with `rank < size - 1` we send something to the process with `rank + 1`:

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

int main (int argc, char **argv)
{
    int size, rank, dummy=0;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    if (rank != 0)
    {
        MPI_Recv(&dummy, 1, MPI_INT, rank-1,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    printf("I am %d of %d\n", rank, size);

    if (rank < size-1)
    {
        MPI_Send(&dummy, 1, MPI_INT, rank+1,
            0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

When we compile and run we observe that we have an ordered execution:

```
% mpicc mpi_helloworld.c -o mpi_helloworld

% mpirun ./mpi_helloworld
I am 0 of 48
I am 1 of 48
I am 2 of 48
I am 3 of 48
I am 4 of 48
...
I am 42 of 48
I am 43 of 48
I am 44 of 48
I am 45 of 48
I am 46 of 48
I am 47 of 48
```

## Task 3.5

Question: Compile and run the following sequential Hello World program. What is the output? Then, parallelize (compile and run) the hello world sequential code adding the most basic OpenMP parallel directive. How many threads are created? Why?

When compiling and running the sequential code the output is the following:

```
% ./hello
hello(0)world(0)
```

The most basic OpenMP parallelization is adding the `omp parallel` directive:

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

int main(){
        int ID = 0;
        #pragma omp parallel
        printf("hello(%d)", ID);
        printf("world(%d) \n", ID);
        return 0;
}
```

If we execute the program now we can observe the following output:

```
% ./hello
```

```
hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)\
hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)\
hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)\
hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)\
hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)\
hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)hello(0)world(0)
```

We notice that "hello" was printed 48 times, which means that 48 threads are created. In the case that we don't specify the number of threads in the `omp parallel` directive the runtime system determines how many of them to spawn.

### Task 3.6

Question: Write a multithreaded version of the same program where each thread prints its thread num as a ID. What happened? Why? How can we solve the problem?

We can change the code in order to print the rank of each thread as it executes. For this we use `omp_get_thread_num()` and `omp_get_num_threads()` functions:

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

void hello(void);

int main(){
        #pragma omp parallel
        hello();
        return 0;
}

void hello(void){
        int my_rank = omp_get_thread_num();
        int thread_count = omp_get_num_threads();

        printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

As we would expect, the output prints `x out of 48` for each thread:

```
% ./hello
Hello from thread 0 of 48
Hello from thread 16 of 48
Hello from thread 7 of 48
Hello from thread 4 of 48
Hello from thread 30 of 48
Hello from thread 34 of 48
Hello from thread 17 of 48
Hello from thread 43 of 48
Hello from thread 41 of 48
Hello from thread 45 of 48
Hello from thread 31 of 48
Hello from thread 19 of 48
Hello from thread 3 of 48
Hello from thread 47 of 48
Hello from thread 6 of 48
Hello from thread 27 of 48
Hello from thread 46 of 48
Hello from thread 9 of 48
Hello from thread 24 of 48
Hello from thread 25 of 48
Hello from thread 44 of 48
Hello from thread 12 of 48
Hello from thread 2 of 48
Hello from thread 5 of 48
Hello from thread 23 of 48
Hello from thread 20 of 48
Hello from thread 28 of 48
Hello from thread 22 of 48
Hello from thread 21 of 48
Hello from thread 11 of 48
Hello from thread 10 of 48
Hello from thread 15 of 48
Hello from thread 13 of 48
Hello from thread 14 of 48
Hello from thread 1 of 48
Hello from thread 18 of 48
Hello from thread 32 of 48
Hello from thread 35 of 48
Hello from thread 37 of 48
Hello from thread 39 of 48
Hello from thread 29 of 48
Hello from thread 42 of 48
Hello from thread 8 of 48
Hello from thread 40 of 48
Hello from thread 33 of 48
Hello from thread 26 of 48
Hello from thread 36 of 48
Hello from thread 38 of 48
```

We notice that threads don't execute in order of their rank. In fact, the order depends on how the scheduler assigns execution of each thread.

In order to make the threads execute in order of rank, use the `omp ordered` directive:

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

void hello(void);

int main(){
        #pragma omp parallel
        hello();
        return 0;
}

void hello(void){
        int my_rank = omp_get_thread_num();
        int thread_count = omp_get_num_threads();
        #pragma omp ordered
        printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

At this point, each thread executes in order of its rank:

```
% ./hello
Hello from thread 0 of 48
Hello from thread 1 of 48
Hello from thread 2 of 48
Hello from thread 3 of 48
Hello from thread 4 of 48
Hello from thread 5 of 48
Hello from thread 6 of 48
Hello from thread 7 of 48
Hello from thread 8 of 48
Hello from thread 9 of 48
Hello from thread 10 of 48
Hello from thread 11 of 48
Hello from thread 12 of 48
Hello from thread 13 of 48
Hello from thread 14 of 48
Hello from thread 15 of 48
Hello from thread 16 of 48
Hello from thread 17 of 48
Hello from thread 18 of 48
Hello from thread 19 of 48
Hello from thread 20 of 48
Hello from thread 21 of 48
Hello from thread 22 of 48
Hello from thread 23 of 48
Hello from thread 24 of 48
Hello from thread 25 of 48
Hello from thread 26 of 48
Hello from thread 27 of 48
Hello from thread 28 of 48
Hello from thread 29 of 48
Hello from thread 30 of 48
Hello from thread 31 of 48
Hello from thread 32 of 48
Hello from thread 33 of 48
Hello from thread 34 of 48
Hello from thread 35 of 48
Hello from thread 36 of 48
Hello from thread 37 of 48
Hello from thread 38 of 48
Hello from thread 39 of 48
Hello from thread 40 of 48
Hello from thread 41 of 48
Hello from thread 42 of 48
Hello from thread 43 of 48
Hello from thread 44 of 48
Hello from thread 45 of 48
Hello from thread 46 of 48
Hello from thread 47 of 48
```

## Task 3.7

Question: Compile and execute in your login node the sequential code 3.1 presented in section 3.4.1 .

We have the following code:

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

/* Function we are integrating */
double f(double x){
    return (4.0 / (1.0 + x*x));
}

/* Calculate local integral */
double TrapezoidalRule( double a, double b, double h, int n)
{
    double EstimatedArea, x;
    int i;

    EstimatedArea = (f(a) + f(b))/2.0;
    for (i = 1; i <= n-1; i++) {
        x = a+ i*h;
        EstimatedArea += f(x);
    }
    EstimatedArea = EstimatedArea*h;
    return EstimatedArea;
}

int main(int argc, char *argv[])
{
    int n;
    printf("Enter the number of subintervals: ");
    scanf("%d",&n);

    double a=0;
    double b=1;
    double h = (b-a) / (double) n;

    double pi = TrapezoidalRule(a, b, h, n);
    printf("With n = %d trapezoids, ", n);
    printf("the integral from %f to %f is %.15e\n", a, b, pi);
}
```

When we compile and run we have the following output:

```
% ./pi
Enter the number of subintervals: 10
```

```
With n = 10 trapezoids, the integral from 0.000000 to 1.000000 is
3.139925988907159e+00
% ./pi
Enter the number of subintervals: 100
With n = 100 trapezoids, the integral from 0.000000 to 1.000000 is
3.141575986923129e+00
```

### Task 3.8

Question: Create and execute a MPI program that estimates the number PI.

In order to parallelize the previous code, we calculate the trapezoidal rule on different intervals in each process and then in the main process we make the sum:

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#define TRAP 1000

/* Function we are integrating */
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

/* Calculate local integral */
double TrapezoidalRule( double a, double b, double h, int n)
{
    double EstimatedArea, x;
    int i;
    EstimatedArea = (f(a) + f(b))/2.0;
    for (i = 1; i <= n-1; i++)
    {
        x = a+ i*h;
        EstimatedArea += f(x);
    }
    EstimatedArea = EstimatedArea*h;
    return EstimatedArea;
}

int main(int argc, char *argv[])
{
    int my_rank, comm_sz, n, local_n;
    double local_a, local_b;
    int source;
    double local_int, total_int;
    double a, b, h;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    n = TRAP * comm_sz; /*number of trapezoids must be multiple of proc*/
    a = 0;
    b = 1;

    h = (b-a) / (double)n;
    local_n = n/comm_sz;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    local_int = TrapezoidalRule(local_a, local_b, h, local_n);

    if (my_rank != 0)
    {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
    else
    {
        total_int = local_int;
        for (source = 1; source < comm_sz; source++)
        {
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_int += local_int;
        }
    }

    if (my_rank == 0)
    {
        printf("With n = %d trapezoids, ", n);
        printf("from %f to %f is %.15e\n", a, b, total_int);
    }

    MPI_Finalize();

    return 0;
}
```

When we compile and run we have the following output:

```
% make
mpicc     pi.c  -lm -o pi

% mpirun ./pi
With n = 48000 trapezoids, from 0.000000 to 1.000000 is 3.141592653517455e+00
```

## Task 3.9

Question: Create and execute a OpenMP program pi.c to estimate the value of number PI. Consider the number of theads is a parameter. The input data a, b, n are hardwired for simplicity. Use the following code as a base and include the parallel pragma for parallelizing and use a critical directive for global sum (each thread explicitly computes the integral over its assigned subinterval).

In order to parallelize the sequential code with OpenMP we use a reduction to sum up the computation of the trapezoidal rule on different intervals:

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

double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double TrapezoidalRule(double a, double b, int n)
{
    double h, x, my_result;
    double local_a, local_b;

    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;


    for (i = 1; i <= local_n-1; i++)
    {
        x = local_a + i*h;
        my_result += f(x);
    }
    return my_result*h;
}

int main(int argc, char* argv[])
{
    double global_result = 0.0;
    double a, b;
    int n;
    int thread_count;

    if(!argv[1])
    {
        printf("Please, specify the number of threads\n");
        return 1;
    }

    n=268435456; /*16^7 */
    a=0;
    b=1;
    thread_count = strtol(argv[1], NULL, 10);

    global_result = 0.0;
#pragma omp parallel num_threads(thread_count) reduction(+:global_result)
    global_result += TrapezoidalRule(a, b, n);
    printf("OpenMP with %d threads and  n = %d trapezoids, ",
            thread_count,n);
    printf("the integral from %f to %f is  %.15e\n", a, b,
                global_result);
    return 0;
}
```

To measure the time, more reliably, the computation is repeated 100 times, and the time is measured of the whole execution.

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

#define RUNS 100
#define MAX_ERR 1e-6

double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double TrapezoidalRule(double a, double b, int n)
{
    double h, x, my_result;
    double local_a, local_b;

    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;


    for (i = 1; i <= local_n-1; i++)
    {
        x = local_a + i*h;
        my_result += f(x);
    }
    return my_result*h;
}

int comp(int thread_count)
{
    double global_result = 0.0;
    double a, b;
    int n;

    n=268435456; /*16^7 */
    a=0;
    b=1;

    global_result = 0.0;
#pragma omp parallel num_threads(thread_count) reduction(+:global_result)
```

```
    global_result += TrapezoidalRule(a, b, n);
    //printf("OpenMP with %d threads and  n = %d trapezoids, ",
    //      thread_count,n);
    //printf("the integral from %f to %f is  %.15e\n", a, b,
    //      global_result);

    if(fabs(M_PI - global_result) > MAX_ERR)
    {
        printf("The error is to big: %e\n", M_PI - global_result);
        return 1;
    }

    return 0;
}

int main(int argc, char* argv[])
{
    int i, thread_count;

    if(!argv[1])
    {
        printf("Please, specify the number of threads\n");
        return 1;
    }

    thread_count = strtol(argv[1], NULL, 10);

    for(i = 0; i<RUNS; i++)
    {
        if (comp(thread_count))
        {
            printf("Computation failed\n");
            return 1;
        }
    }

    return 0;
}
```

As the number of threads grow, the execution time is reduced, until we pass the number of threads of the machine (48):

```
% ./run.sh
Running with 1 thread(s)

real    0m27.233s
user    0m27.218s
sys     0m0.008s
Running with 2 thread(s)

real    0m13.661s
user    0m27.281s
sys     0m0.020s
Running with 4 thread(s)
```

```
real    0m6.858s
user    0m27.344s
sys     0m0.012s
Running with 8 thread(s)

real    0m3.435s
user    0m27.375s
sys     0m0.000s
Running with 16 thread(s)

real    0m1.728s
user    0m27.394s
sys     0m0.012s
Running with 32 thread(s)

real    0m0.896s
user    0m27.783s
sys     0m0.020s
Running with 64 thread(s)

real    0m1.672s
user    0m33.722s
sys     0m40.234s
```