Supercomputers Architecture

Hands on 3

November 24th, 2015

Constantino Gomez
Albert Segura
Cristobal Ortega

Exercise 1: Compile and run the following sequential “Hello World” program. What is the output?

```
$ gcc hello_world.c
:~/handson3 $ ./a.out
hello(0)world(0)
```

Exercise 2: Parallelize (compile and run) the “hello world” sequential code adding the most basic parallel directive. How many threads are created? Why?

```
 $ gcc hello_omp.c -fopenmp
:~/handson3 $ ./a.out
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
hello(0)world(0)
```

16 threads were created, because if the environment variable OMP_NUM_THREADS is not set then OMP creates 1 thread per CPU (and in MN3 nodes we can see 16 cpus per node)

Exercise 3: Write a multithreaded version of the same program where each thread prints its thread_num as a ID. What happened? Why?

```
$ gcc hello_omp.c -fopenmp
:~/handson3 $ ./a.out
hello(14)world(14)
hello(15)world(15)
hello(8)world(8)
hello(0)world(0)
hello(12)world(12)
hello(10)world(10)
hello(13)world(13)
hello(9)world(9)
```

```
hello(11)world(11)
hello(1)world(1)
hello(4)world(4)
hello(6)world(6)
hello(5)world(5)
hello(3)world(3)
hello(7)world(7)
hello(2)world(2)
```

They do not execute in order because we are not controlling it (we just created and told them what to execute)

Exercise 4
Create an OpenMP program to estimate definite integral (or area under curve) using trapezoidal rule. The number of threads is a parameter. The input is a, b, n (estimate of integral from a to b of f(x) using n trapezoids). You can assume that the function f(x) is hardwired (*). Use a critical directive for global sum (each thread explicitly computes the integral over its assigned subinterval). You can assume that n is evenly divisible by the number of threads.

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

double f(double x); /* Function we're integrating */
void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
      double global_result = 0.0; /* Store result in global_result
*/
      double a, b; /* Left and right endpoints */
      int n; /* Total number of trapezoids */
      int thread_count;

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

      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", &a, &b, &n);

      #pragma omp parallel
      Trap(a, b, n, &global_result);

      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.14e\n",
```

```
      a, b, global_result);

      return 0;
}

double f(double x) {
      double return_val;
      return_val = x*x;
      return return_val;
}

void Trap(double a, double b, int n, double* global_result_p) {
      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);
      }
      my_result = my_result*h;

      #pragma omp critical
      *global_result_p += my_result;
}
```

Exercise 5: Create and submit a LSF jobscript with the execution of OpenMP version with 16 processors. Add the LSF jobscript in the answer.

```
#BSUB -J trapezoidal_omp
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 00:15
#BSUB -n 8
#BSUB -R "span[ptile=16]"
```

```
#BSUB -x
#BSUB -U sa

#Executing
./trapezoidal 16
```

Exercise 6:
Create a lsf jobscript with the execution of OpenMP version with 2,4,8 and 16 threads/processors and obtain the execution time of each of them. Compare the execution time of all the executions with some value for N, a and b (very high values).

```
LSF:
$ cat exercise_6.sh
#BSUB -J trapezoidal_omp
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 02:00
#BSUB -n 16
#BSUB -R "span[ptile=16]"
#BSUB -x
#BSUB -U sa

#Executing
for i in 2 4 8 16
do
     /usr/bin/time -f %E -ao ${i}_threads.txt ./trapezoidal $i <
input.txt
     /usr/bin/time -f %E -ao ${i}_threads.txt ./trapezoidal $i <
input.txt
done

$ cat input.txt
0 3221225472 3221225472
```

OMP	**2 threads**	**4 threads**	**8 threads**	**16 threads**
Time	9.715	4.645	2.385	1.26

Exercise 7:
Create a lsf jobscript with the execution of MPI version with 2,4,8 and 16 processors and obtain the execution time of each of them. Use the same values for N, a and b.

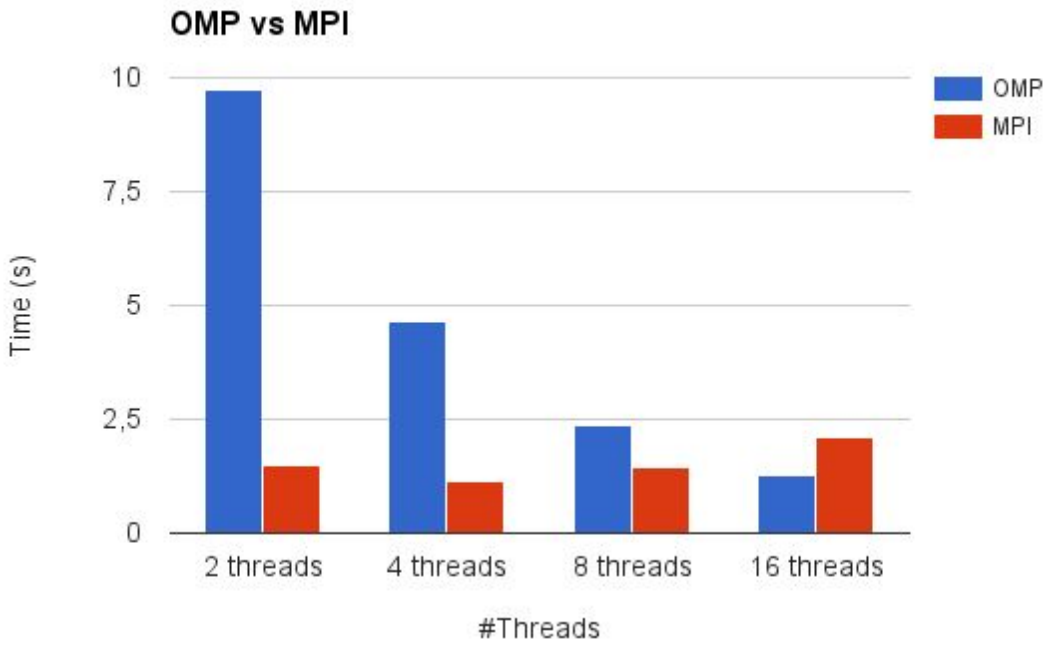
```
$ cat exercise_7.sh
#BSUB -J trapezoidal_omp
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 02:00
#BSUB -n 16
#BSUB -R "span[ptile=16]"
#BSUB -x

#Executing
for i in 2 4 8 16
do
/usr/bin/time -f %e -ao ${i}_threads_mpi.txt mpirun -np $i
./trapezoidal_mpi < input.txt
/usr/bin/time -f %e -ao ${i}_threads_mpi.txt mpirun -np $i
./trapezoidal_mpi < input.txt
done
```

MPI	**2 threads**	**4 threads**	**8 threads**	**16 threads**
Time	1.48	1.12	1.45	2.095

Exercise 8:
Compare and discuss the results of the previous two exercises 6 and 7.

	2 threads	**4 threads**	**8 threads**	**16 threads**
OMP	9.715	4.645	2.385	1.26
MPI	1.48	1.12	1.45	2.095



The plot shows how OMP implementation scales up to 16 threads and MPI implementation just scales up to 4 threads, then the execution time gets worse.
But, MPI implementation has more performance in terms of execution time than OMP implementation until 8 threads, then OMP implementation takes less time than MPI implementation.
This can be explain because MPI implementation has not to create a new threads (just a new process, so there is no data copy from the parent to the new process), but it has to communicate the results to the master thread, that is the reason we can see that performance starts to degrade with 8 threads.
Probably, with a larger input MPI implementation could afford having more process spawned.

In the other hand we see that OMP implementation have a bad performance for 2 threads but keep scaling up to 16 threads, this is due to have to pay the overhead for creating threads, that overhead is more visible when only creating 1 thread (master and thread) because after that the work to do is just shared between 2 threads. But, when we create 15

threads (plus the master) the work and the spawning process can be overlapped, so the master threads keep spawning threads, but there are already threads working.
Probably, if we keep increasing the number of threads we would see a performance lost due to the `#pragma critical` that there is the in code or due to the amount of work is so tiny that the overhead in spawning threads is larger than the work.

Exercise 9:
Implement a parallel MPI matrix-vector multiplication using one-dimensional arrays to store the vectors (x) and the matrix (A). Vectors use block distributions and the matrix is distributed by block rows. As a input parameters of the program indicate the dimensions of the matrix (m = number of rows, n = number of columns). The output (printf) will be the product vector y = Ax. For simplicity you may assume that the number of processes should be evenly divisible by both m and n. The program will use the MPI directives in "MPI advanced" module presented in the theory class.

```
$ cat mat_vec.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

void Read_matrix(char prompt[], double A[], int m, int n);
void Read_vector(char prompt [], double x[], int n);
void Mat_vect_mult(double A[], double x[], double y[], int m, int
n);
void print_times();
struct timeval start_time, end_time;

int main(int argc, char *argv[]) {
      double* A = NULL;
      double* x = NULL;
      double* y = NULL;
      long long int size;
      int m, n;

      n = atoi(argv[1]);
      m = n;
      size=m*n*sizeof(double);

      A = malloc(size);
      x = malloc(n*sizeof(double));
      y = malloc(m*sizeof(double));
```

```
      Read_matrix("A", A, m, n);
      Read_vector("x", x, n);

      gettimeofday(&start_time, NULL);

      Mat_vect_mult(A, x, y, m, n);

      gettimeofday(&end_time, NULL);

      print_times();

      free(A);
      free(x);
      free(y);
      return 0;

} /* main */

void Read_matrix(char prompt[], double A[], int m, int n) {
      int i, j;
      for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
                A[i*n+j]=rand();
}

void Read_vector( char prompt[], double x[], int n) {
      int i;
      for (i = 0; i < n; i++)
                x[i]=rand();
}
void Mat_vect_mult(double A[], double x[], double y[], int m, int
n) {
      int i, j;
      for (i = 0; i < m; i++) {
                y[i] = 0.0;
                for (j = 0; j < n; j++)
                      y[i] += A[i*n+j]*x[j];
      }
}

void print_times()
{
      int total_usecs;
      float total_time, total_flops;
      total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
      (end_time.tv_usec-start_time.tv_usec);
```

```
      printf(" %.2f mSec \n", ((float) total_usecs) / 1000.0);
      total_time = ((float) total_usecs) / 1000000.0;
}
```

Exercise 10: Create a LSF jobscript and submit it with 64 processors. Add the LSF jobscript in the answer.

```
$ cat exercise_10.sh
#BSUB -J mat_vec_mpi
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 02:00
#BSUB -n 64
#BSUB -R "span[ptile=16]"
#BSUB -x

#Executing
for i in 64
do
/usr/bin/time -f %e  mpirun -np $i ./mat_vec_mpi 1024
done
```

Exercise 11: Recovered from your previous hands-on the time for executing the sequential version of matrix-vector multiplication (exercise 12 of Hands-on 1) for N=32.768

```
$ /usr/bin/time ./a.out 32768 32768
15.39user 1.64system 0:17.10elapsed 99%CPU (0avgtext+0avgdata
4194868maxresident)k
0inputs+0outputs (0major+1048757minor)pagefaults 0swaps
```

Now timing the parallel MPI matrix-vector multiplication from exercise 10 in Hands-on 3 for 2, 4, 8, 16, 32 and 64:

	1	2	4	8	16	32	64
Time	13,16	28,98	28,12	28,99	30,96	32,96	33,28

Obtain the Speedup of this application and discuss the results with your partner and the professor in the lab session. Plot the results in your report lab.
What is your conclusion?

We can see that we are getting slowdown instead of speedup, this can mean that overhead of communicating is so big that does not compensate for the shared work, and as we can see we have after using MPI with 2 processes the performance remains the same. So, probably the work to do is too less for the overhead of using MPI.