Report

Parallel Programming

- Practical experience with parallel programming

10 Oct, 2017

Semester: Autumn 2017
Course: Parallel Computing
Author: Sarpreet Singh Buttar

Task 1

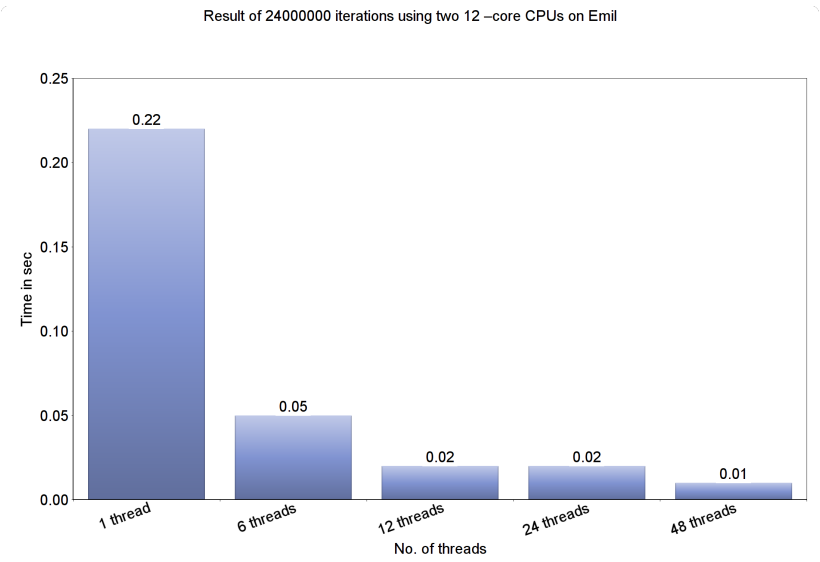
In order to complete this task, I made two methods, one which calculates the PI in parallel and return the execution time, other which calculate and print the average time based on given execution times. Then I call these methods from *main* method with different inputs(threads, iterations). Below is the representation of the code:

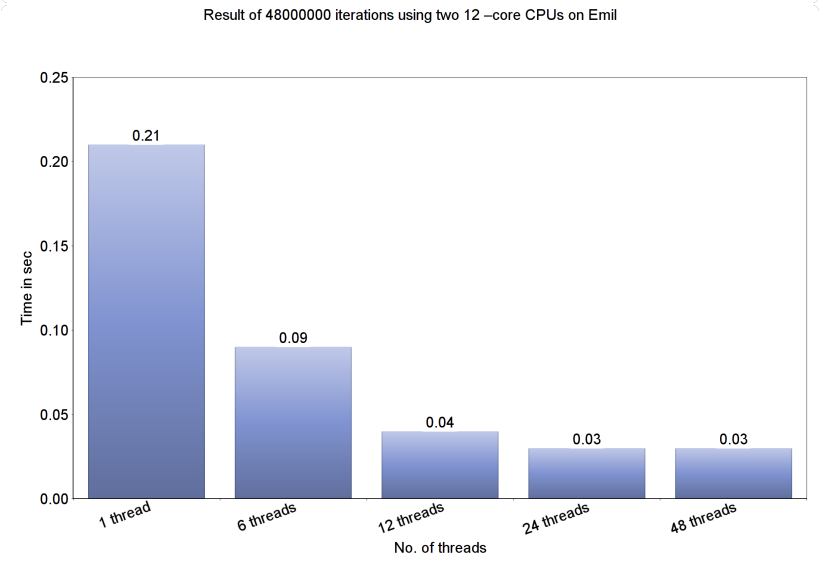
```
int main (int argc, char *argv[]) {
    ...
    int iterations[]= {24000000, 48000000, 96000000},
    threads[] = {1, 6, 12, 24, 48};
    double execution_times[5];
    for(int j = 0; j < 5; j++)
        for(int k = 0; k < 3; k++) {
            m = 1.0 / (double)iterations[k];
            for(int l = 0; l < 5; l++) {
                execution_times[l] = calculatePI(i, iterations[k],
                ni, m, mypi, threads[j]);
            }
        calculateAverage(execution_times, 5);
        }
}

double calculatePI (...) {
    double start_time = omp_get_wtime();
    #pragma omp parallel for num_threads(threads),
    reduction(+:ni), reduction(+:mypi)

    //calculate PI here with given code...
    double end_time = omp_get_wtime() - start_time;
    ..
    return end_time;
    }
}
```

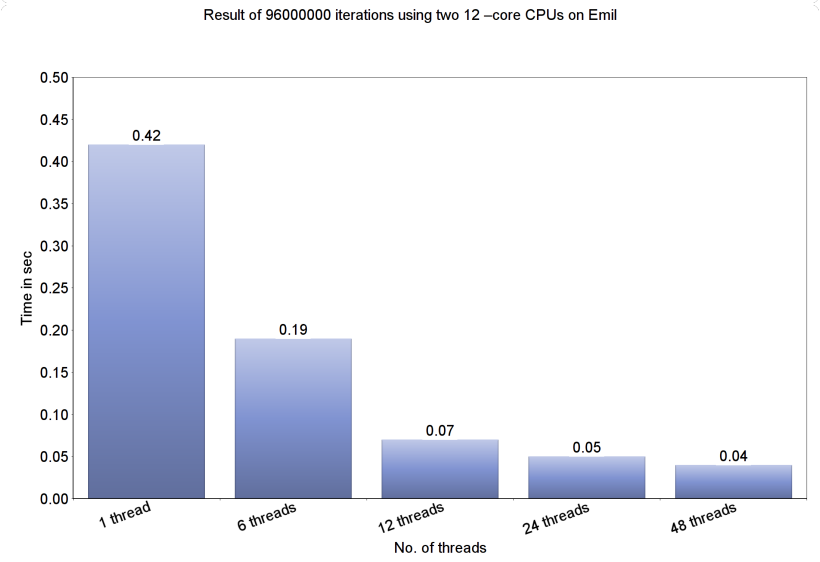
In *main* method I have used nested loops to run all the threads with all iterations 5 times automatically rather than changing the inputs manually. Once the most inner loop runs 5 times, I calculate and print the average of given execution times. Inside the *calculatePI* method, I used *#pragma* command by specifying the number of threads for calculating the PI in parallel. I also do reduction on the shared variables for getting the correct result as well as measure the execution time and return it. Below is the representation of the execution time in the form of table as well as graph.

	1	6	12	24	48
24000000	0.22s	0.05s	0.02s	0.02s	0.01s
48000000	0.21s	0.09s	0.04s	0.03s	0.03s
96000000	0.42s	0.19s	0.07s	0.05s	0.04s

In the above table 1, 6, 12, 24 and 48 represents the number of threads whereas 24000000, 48000000 etc represents the number of iterations.


Result of 24000000 iterations using two 12 –core CPUs on Emil
Time in sec
0.25
0.20
0.15
0.10
0.05
0.00
0.22
0.05
0.02
0.02
0.01
1 thread
6 threads
12 threads
24 threads
48 threads
No. of threads


Result of 48000000 iterations using two 12 –core CPUs on Emil
Time in sec
0.25
0.20
0.15
0.10
0.05
0.00
0.21
0.09
0.04
0.03
0.03
1 thread
6 threads
12 threads
24 threads
48 threads
No. of threads


Result of 96000000 iterations using two 12 –core CPUs on Emil
Time in sec
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.42
0.19
0.07
0.05
0.04
1 thread
6 threads
12 threads
24 threads
48 threads
No. of threads

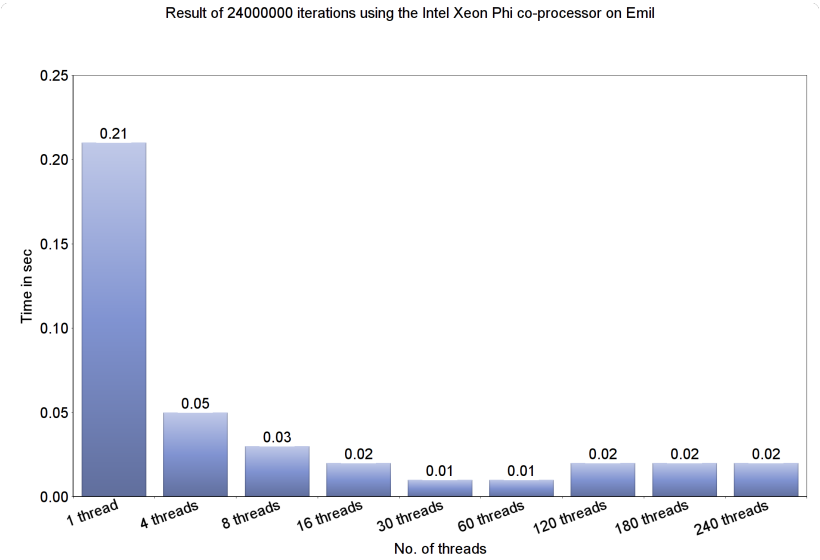
Task 2

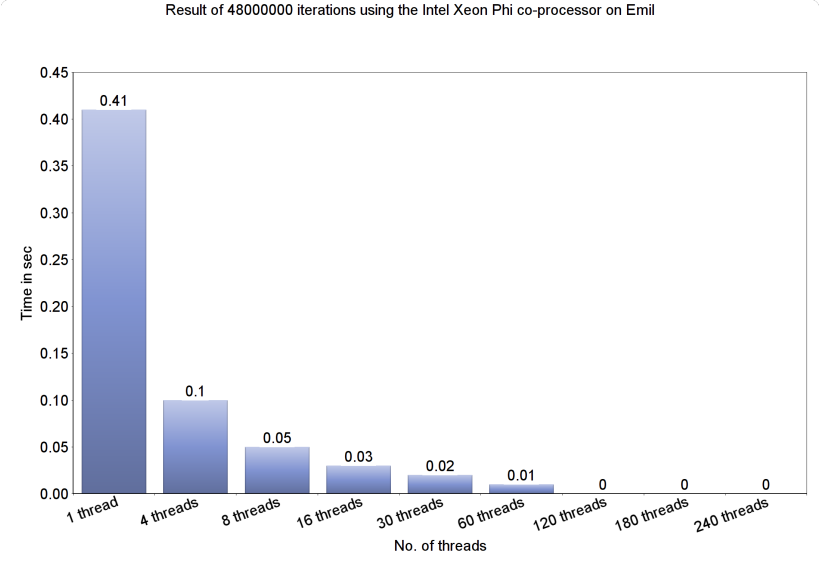
For completing this task, I used native programming mode. All the code is same as Task 1 however in this task number of threads increased which means the *main* method will look like

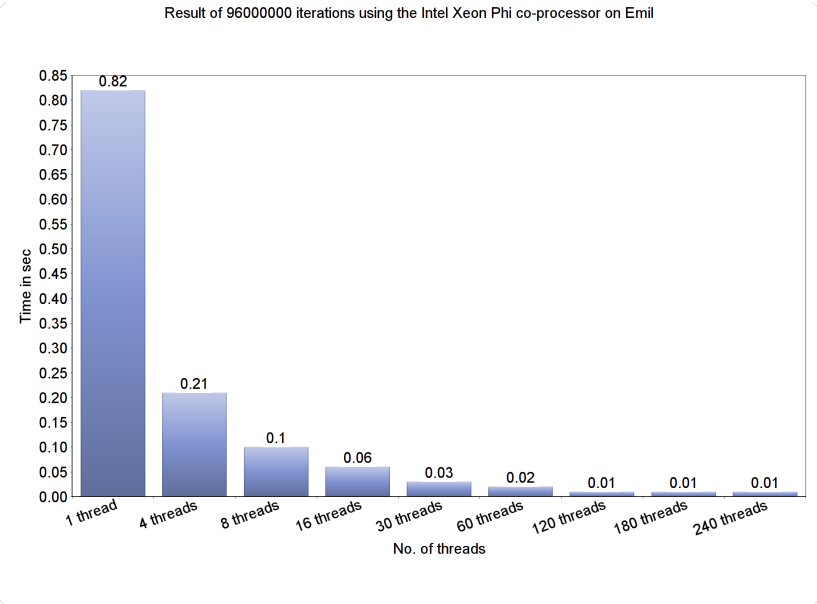
```
int main (int argc, char *argv[]) {
    ...
    int iterations[]= {24000000, 48000000, 96000000},
    threads[] = {1, 4, 8, 16, 30, 60, 120, 180, 240};
    double execution_times[5];
    for(int j = 0; j < 9; j++)
        .....
}
```

Below is the representation of the execution time in the form of table as well as graph.

	1	4	8	16	30	60	120	180	240
24000000	0.21s	0.05s	0.03s	0.02s	0.01s	0.01s	0.02s	0.02s	0.02s
48000000	0.41s	0.10s	0.05s	0.03s	0.02s	0.01s	0.00s	0.00s	0.00s
96000000	0.82s	0.21s	0.10s	0.06s	0.03s	0.02s	0.01s	0.01s	0.01s







Instructions for running and testing the programs

For running the task 1

1. copy the task1.c file to emil
2. compile it with gcc
3. run the file

For running the task 2

1. copy the task1.c file to emil
2. compile it with icc
3. copy the binary file as well as libiomp5.so to mic0
4. ssh to mic0 and export the library path
5. run the binary file