Into to Parallel Programming Analysis

Mohamad Ibrahim Cheikh

November 29, 2020

1 Introduction

Parallel computing refers to simulations in which multiple computational resources are implemented to solve a problem at the same time. Parallelism is usually achieved by dividing the problem into many smaller ones, in which each one is solved separately and also at the same time. Different forms of parallel programing has been developed like bit-level, instruction-level, data, and task parallelism.

The current report will analyze the performance of two parallel execution models, that give the software the capability to control multiple different flows of work at the same time, which are *pthreads* and *OpenMp*.

2 Experimental Configuration

The supercomputer on which we are going to conduct our performance analysis is Beocat, which is a high performance computer cluster at the Kansas State University [1]. The performance analysis will be performed on the Elves nodes of Beocat [2]. Table 1 gives a brief description of the nodes configuration.

Elve Nodes				
Nodes	1-56	57-72,77	73-76,78, 79	80-85
Processors	2x 8-Core Xeon E5-2690	2x 10-Core Xeon E5-2690 v2	2x 10-Core Xeon E5-2690 v2	2x 10-Core Xeon E5-2690v2
Ram	64GB	96GB	384GB	64GB
Hard Drive	1x 250GB 7,200 RPM SATA	1x 250GB 7,200 RPM SATA	1x 250GB 7,200 RPM SATA	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350	4x Intel I350	4x Intel I350	4x Intel I350
10GbE and QDR Infiniband	MT27500 Family (ConnectX-3)	MT27500 Family (ConnectX-3)	MT27500 Family (ConnectX-3)	MT27500 Family (ConnectX-3)

Table 1: Elves node configuration taken from [2]

3 Pthreads and OpenMp

This section will detail two of the widely used parallel execution models, **pthreads** and **OpenMp**. These two model are applied to an existent `C` code example, itrates for a set of numbers, computes `x`, where `x` is defined as,

$$\texttt{x} = (\texttt{i} + 0.25) * (1/\texttt{NUM\_ITER})$$

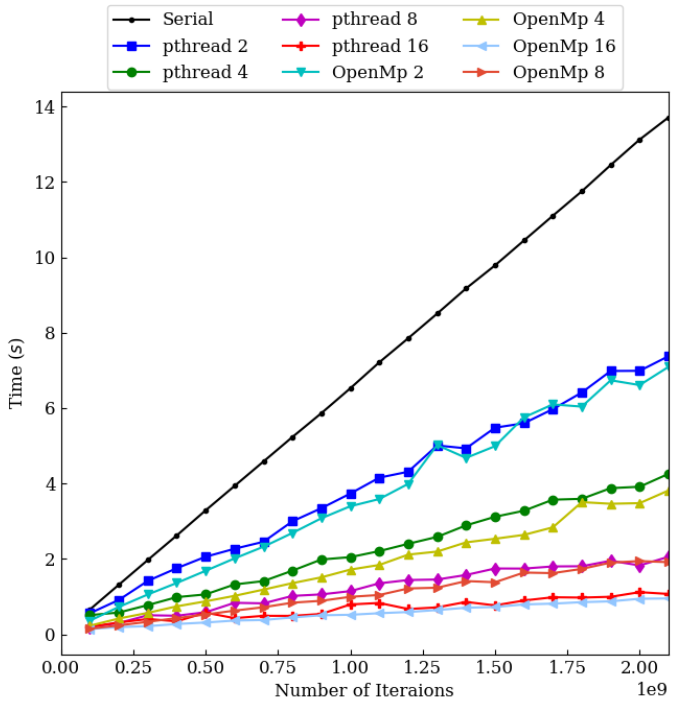


Figure 1: Computational time versus number of iterations (`NUM_ITER`) for the serial, pthread, and openmp code

where `i` represents the loop iteration starting from 0 and going to the maximum number of iterations (Known as `NUM_ITER`), and then sums up all the values of `x`.

Before implementing the parallelized veriosn, the original code was modified in a way to accept input from the terminal. The input was assigned to be the `NUM_ITER` variable which controls the maximum number of iterations. During this process the timer will start calculating the amount of time it takes for the summation function to finish. One reason as to why the timer is only used on the summation function and not the at the beginning of the code is because we want to compare the performance the two parallelization models on the performance of the summation function.

Performance analysis of the two parallel execution models, was carried out with a range of values for `NUM_ITER` starting with of 100 million and stopping at 2.1 billion. As for the number of cores each parallelization model implemented, they are listed in the table below

	Number of Cores			
pthreads	2	4	8	16
OpenMp	2	4	8	16

Table 2: Number of cores implemented for each parallel execution model

4 Results

Figure 1 plots the computational time taken by the serial,**pthreads**, and **OpenMp** code vs the max number of iterations (`NUM_ITER`). The graph shows that as the max number of iterations (`NUM_ITER`) increases, the serial code would take the longest to finish, followed by the parallelized codes that use 2 cores, then 4, 8, and 16 cores. Thus we can conclude that as the number of threads increase, the computational time

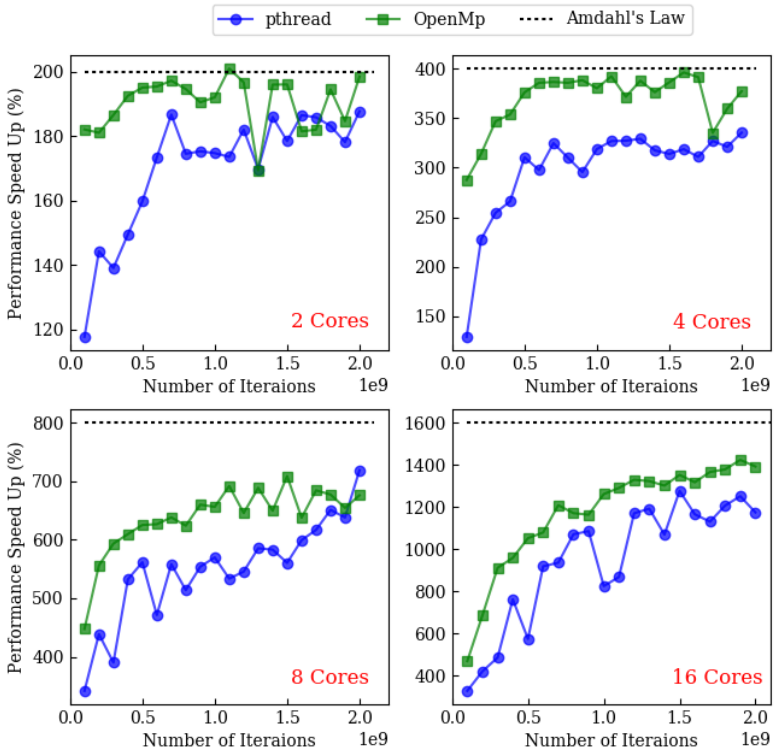


Figure 2: Avaerage performance speed up of the parallelized code versus the serial code for different number of cores

would decrease as expected. Moreover from the graph we can see that when the number of cores are the same both **pthreads**, and **OpenMp** have almost the same computational time.

In order to access the performance boost associated with each parallel model, the following equation was implemented on the computational time:

$$\texttt{Percentage\_Performance\_Speed\_Up} = \frac{\texttt{Serial\_Computation\_Time}}{\texttt{Parallel\_Computation\_Time}} \times 100\% \tag{1}$$

The formula is simply trying to use the serial computational time as a reference, and then divide the serial computational time by the estimated parallel execution time for each value of the max number of iterations `NUM_ITER`. The performance analysis results are plotted in figure 2 versus Amdahl's Law. Amdahl's Law is used as an upper bound on the speedup of a parallelized application, Amdahl's Law is shown below,

$$\texttt{Amdahl\_Performance\_Speed\_Up} \leq \frac{1}{(1 - \texttt{pctPar}) + \frac{\texttt{pctPar}}{\texttt{\# cores}}} \times 100\% \tag{2}$$

where `pctPar` represent the percentage of the code that was parallelized, in our case since we are timing the function that iterates and sums up, and this function is the one that was parallelized then we can say that `pctPar` = 1.

It can be seen from figure 2 that as the max number of iterations increase, the parallel performance increases, and gets closer to Amdahl's Law. The reasons as to why the parallel code is far from Amdahl's Law for small max number of iterations is is due to the fact that Amdahl's Law doesn't take into account

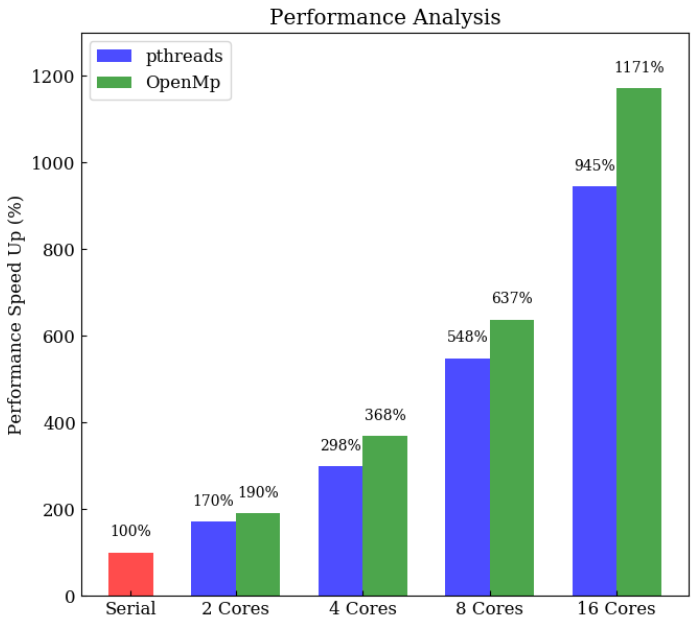


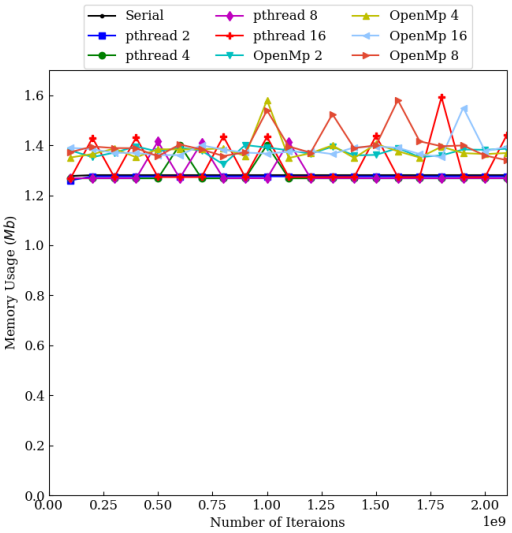
Figure 3: Avaerage performance speed up of the parallelized code versus the serial code for different number of cores

the real-world overheads such as communication, synchronization, and thread management. So for small number of iterations (`NUM_ITER`), the code will spend most of its time in dividing the process and giving instructions to each core to what to run, while a little bit time is spent on computation, and performing the required task. As the number of iterations increases, more time will be spent on computation in comparison to communication and thread management, and so we see that the performance gets closer to Amdahl's law for large number of iterations (`NUM_ITER`). From that figure it can also be seen that the performance of **OpenMp** is better then the performance **pthreads** since it gets to Amdahl's law faster with the same number of cores.

Finally we averaged the data plotted in figure 2, and plotted the average performance in a bar graph that is shown in figure 3. The graph shows that on average as the number of cores doubles the performance is almost doubled, with **OpenMp** being a little bit faster (10% to 20% faster) than **pthreads**.

The final two figures that are plotted, are the memory usage (figure 4a) which shows that the serial code is using the least amount of memory, while **OpenMp** is using the most amout (probably due to the **OpenMp**library) and the final results from the function (figure 4b), which validates that all of the codes are getting the same output.

(a) Memory Usage



(b) Final Answer

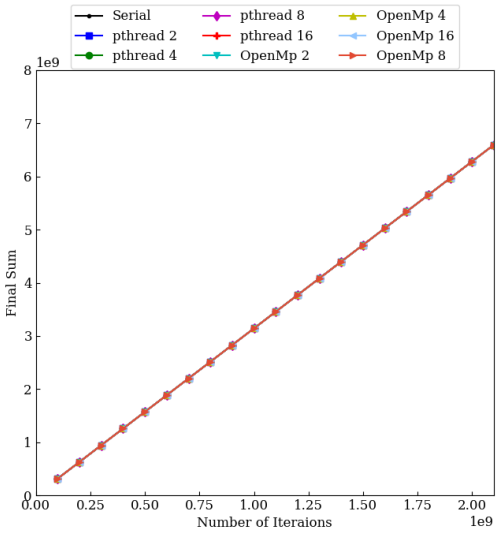


Figure 4: (a) Memory used by each code, (b) Final answer outputed by each code

5 References:

1.$https://support.beocat.ksu.edu/BeocatDocs/index.php/Main_Page$
2.$https://support.beocat.ksu.edu/BeocatDocs/index.php/Compute_Nodes$

6 Appendice:

Listing 1: 'hw2_serial.c'

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>

/* Global Variables:
--------------------*/
long NUM_ITER;
double sum = 0.0;
double x   = 0.0;
double elapsedTime;

/* Function:
--------------------*/
void divide( )
{
    int i;
    double st  = 1.0 / ( (double) NUM_ITER);

    for (i = 0; i < NUM_ITER; i++)
    {
        x = (i + 0.25)*st;
        sum += 4.0/(x*x+1);
    }
}

/* Main:
--------------------*/
int main(int argc, char *argv[]) {
    struct timeval t1, t2;
    struct rusage ru;

    NUM_ITER = 100000000; // Default Value
    if (argc >= 2){
        NUM_ITER = atol(argv[1]);  // Updated Value taken from terminal argumentes
    }

    gettimeofday(&t1, NULL);
    divide();
    gettimeofday(&t2, NULL);

    elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
    elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms

    getrusage(RUSAGE_SELF, &ru);
    long MEMORY_USAGE = ru.ru_maxrss;   // Memory usage in Kb

    printf("DATA, %lf, %ld, %lf, %ld \n",
                sum, NUM_ITER, elapsedTime, MEMORY_USAGE);
    return 0;
}
```

Listing 2: 'hw2_pthreads.c'

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>

/* Global Variables:
--------------------*/
long NUM_ITER;
double sum = 0.0;
double elapsedTime;
double st;

/* A mutex variable acts like a "lock" protecting access to
a shared data resource */
pthread_mutex_t mutexsum;

int NUM_THREADS;  // Will be updated by the terminal
/* Function:
--------------------*/
void *divide(void *myID)
{
    int i;
    int startPos = ((int) myID) * (NUM_ITER / NUM_THREADS);
    int endPos = startPos + (NUM_ITER / NUM_THREADS);

    printf("myID = %d startPos = %d endPos = %d \n",
                (int) myID, startPos, endPos);

    double local_x   = 0.0;
    double local_sum = 0.0;

    // Add up our section of the local sum
    for (i = startPos; i < endPos; i++)
    {
        local_x = (i + 0.25)*st;
        local_sum += 4.0/(local_x*local_x+1);
    }
    // sum up the partial sum into the global sum
    pthread_mutex_lock (&mutexsum);
    sum += local_sum;
    pthread_mutex_unlock (&mutexsum);
    pthread_exit(NULL);
}

/* Main:
--------------------*/
int main(int argc, char *argv[])
{
    int i,rc;
    void *status;
    struct timeval t1, t2;
    struct rusage ru;

```

```
55      //NUM_THREADS = getenv("NSLOTS");
56      NUM_THREADS = 1;
57      NUM_ITER = 100000000; // Default Value
58      if (argc >= 2){
59          /* Update value taken from terminal argumentes */
60          NUM_ITER = atol(argv[1]);
61          NUM_THREADS = atol(argv[2]);
62      }
63
64      st  = 1.0 / ( (double) NUM_ITER);
65      /* Create, initialize and set thread detached attribute */
66      pthread_t threads[NUM_THREADS]; // creates an array of pthread_t
67      pthread_attr_t attr;
68      pthread_attr_init(&attr); /* initializes the thread attributes
69       object pointed to by attr with default attribute values */
70      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
71       /*sets the detach state attribute of the thread
72       attributes object to joinable*/
73
74      // Starting the Division:
75      gettimeofday(&t1, NULL);
76      printf("DEBUG: starting loop on %s\n", getenv("HOST"));
77      for (i = 0; i < NUM_THREADS; i++ ) {
78          // starts a new thread in the calling proces
79          rc = pthread_create(&threads[i], &attr, divide, (void *)i);
80          if (rc) {
81              printf("ERROR; return code from pthread_create() is %d\n", rc);
82              exit(-1);
83          }
84      }
85      /* Free attribute and wait for the other threads */
86      pthread_attr_destroy(&attr); // destroy a thread attributes object.
87      for(i=0; i<NUM_THREADS; i++) {
88           rc = pthread_join(threads[i], &status);
89           if (rc) {
90             printf("ERROR; return code from pthread_join() is %d\n", rc);
91             exit(-1);
92           }
93      }
94      gettimeofday(&t2, NULL);
95
96      elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
97      elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
98
99      getrusage(RUSAGE_SELF, &ru);
100     long MEMORY_USAGE = ru.ru_maxrss;    // Memory usage in Kb
101
102     pthread_mutex_destroy(&mutexsum);
103     printf("Main: program completed. Exiting.\n");
104     printf("DATA, %lf, %ld, %lf, %ld, %ld \n",
105                 sum, NUM_ITER, elapsedTime, MEMORY_USAGE, NUM_THREADS);
106     pthread_exit(NULL);
107
108     return 0;
109 }
```

Listing 3: 'hw2_openmp.c'

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>

/* Global Variables:
--------------------*/
double sum = 0.0;
double elapsedTime;
double st;
pthread_mutex_t mutexsum; // A mutex variable acts like a "lock"
protecting access to a shared data resource


long NUM_ITER;     // Will be updated by the terminal
int NUM_THREADS;   // Will be updated by the terminal


/* Function:
--------------------*/
void *divide(void *myID)
{
    int i;
    int startPos, endPos;
    double local_x, local_sum;

    #pragma omp private(myID,theChar,charLoc,local_char_count
                ,startPos,endPos,i,j)
    {
        startPos = ((int) myID) * (NUM_ITER / NUM_THREADS);
        endPos = startPos + (NUM_ITER / NUM_THREADS);

        printf("myID = %d startPos = %d endPos = %d \n", (int) myID,
                    startPos, endPos);

        // Initailize Local Array
        local_x   = 0.0;
        local_sum = 0.0;

        // Add up our section of the local sum
        for (i = startPos; i < endPos; i++)
        {
        local_x = (i + 0.25)*st;
        local_sum += 4.0/(local_x*local_x+1);
        }

    // sum up the partial sum into the global sum
    #pragma omp critical
    sum += local_sum;
    }
}

/* Main:
```

```
55 --------------------*/
56
57 int main(int argc, char *argv[])
58 {
59     int i;
60     struct timeval t1, t2;
61     struct rusage ru;
62
63     // Default Values
64     NUM_THREADS = 1;
65     NUM_ITER = 100000000;
66
67     if (argc >= 2){
68         // Updated Value taken from terminal argumentes
69         NUM_ITER = atol(argv[1]);
70         NUM_THREADS = atol(argv[2]);
71     }
72
73     st  = 1.0 / ( (double) NUM_ITER);
74     /* Set the number of threads for the Openmp Enviroment */
75     omp_set_num_threads(NUM_THREADS);
76
77     // Starting the Division:
78     gettimeofday(&t1, NULL);
79     printf("DEBUG: starting loop on %s\n", getenv("HOST"));
80
81     #pragma omp parallel
82     {
83         divide(omp_get_thread_num());
84     }
85     gettimeofday(&t2, NULL);
86
87     elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; //sec to ms
88     elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
89
90     getrusage(RUSAGE_SELF, &ru);
91     long MEMORY_USAGE = ru.ru_maxrss;   // Memory usage in Kb
92
93     pthread_mutex_destroy(&mutexsum);
94     printf("Main: program completed. Exiting.\n");
95     printf("DATA, %lf, %ld, %lf, %ld, %ld, OpenMp \n", sum,
96             NUM_ITER, elapsedTime, MEMORY_USAGE, NUM_THREADS);
97     pthread_exit(NULL);
98
99     return 0;
100 }
```

Listing 4: Bash script used

```
#!/bin/bash
#$ -l mem=1G
#$ -l h_rt=12:00:00
#$ -l killable
#$ -cwd
#$ -q \*@@elves
#$ -pe single 16
#$ -t 1:10:1

for i in {100000000..2100000000..100000000}
do
/homes/mcheikh/CIS_625/hw2/hw2_serial.out $i
/homes/mcheikh/CIS_625/hw2/hw2_pthreads.out $i 2
/homes/mcheikh/CIS_625/hw2/hw2_openmp.out $i 2
/homes/mcheikh/CIS_625/hw2/hw2_pthreads.out $i 4
/homes/mcheikh/CIS_625/hw2/hw2_openmp.out $i 4
/homes/mcheikh/CIS_625/hw2/hw2_pthreads.out $i 8
/homes/mcheikh/CIS_625/hw2/hw2_openmp.out $i 8
/homes/mcheikh/CIS_625/hw2/hw2_pthreads.out $i 16
/homes/mcheikh/CIS_625/hw2/hw2_openmp.out $i 16
done
```