Distributed Computation of the Pearson Correlation Coefficient via MPI

Jefferson Louise Basilio[1]*

Abstract
An exercise that is related from the second, except the four processors are combined into a supercomputer called clusters, acting as multiple brains. The computation for the *Pearson Correlation Coefficient* were distributed into the a maximum of four processors via *Message Passing Interface* (MPI) using the *C* Programming Language. The processor with rank *0* will be the master, *1-3* will be the slaves. The master will initiate the *n x n* matrix in one-dimensional manner, then broadcast the *n* and the corresponding elements using *MPI_Bcast()* and *MPI_Scatter()* respectively. All nodes will compute for the *Pearson Correlation Coefficient* individually and joined by the master using *MPI_Gatherv()*. Computations are limited to small *n* sizes because there have been unsolvable error returns for the large ones. Results show that the computational time decreases as the number of processors increases, although in some sizes this is not the case. The results also imply that serial computations are better although theoretically a cluster should be faster and this is just for academic demonstrations.

[1] *Institute of Computer Science, University of the Philippines Los Baños*
*jcbasilio1@up.edu.ph

Contents

Introduction

The Pearson Correlation Coefficient is a measurement of a linear correlation of two variables which has a range of -1 to +1, inclusive. Having a 0 value means no correlation. Meaning, there is no linear association between the two variables. Having a positive value means there is a positive association - there is a linear proportion existing between the two variables. Positive association means Y increases as X does. Otherwise there is a negative association, which means the two variables inhibit a indirect proportion - Y decreases as X increases[1].

Objectives

The objective of this exercise is to determine the run-time of the program by distributing the parts of a matrix into different processors via MPI. This also serves as a material for learning how an MPI and its functions work, and how supercomputer clusters are built and combined. Additionally, this is to demonstrate how the combined power of clustered *p* processors can be utilized to calculate problems remotely, which is not taught in the academic curriculum the students are under.

1. Methodology

The computer that will run the program will only act as an interface. The actual computations will occur on the clustered supercomputer which has four processors combined. In case the program needs to simulate more processors, the cluster will handle it by making phantom ones.

1.1 Starting the program

The program is compiled with an additional flag, *-lm*.
The command for compilation would be *mpicc -o lab05.exe lab05.c -lm*. The flag will link the math library to allow more advanced integer operations. Then the program will be ran with two command line arguments, *p* processors and *n* size using *mpirun*. In this exercise, the compiling and running is pre-configured by the cluster using *sbatch*.

1.2 Configuring MPI

```
#include <mpi.h>
 ...
MPI_Init(&argc, &argv);
/* Retrieve MPI Details */
MPI_Comm_size(MPI_COMM_WORLD, &csize);
MPI_Comm_rank(MPI_COMM_WORLD, &crank);
MPI_Get_processor_name(cname, &clen);
```

The library *mpi.h* is imported to use MPI functions. To initialize, *MPI_Init()* is called. Information such as rank, number of nodes, and processor names are retrieved using *MPI_Comm_size()*, *MPI_Comm_rank()*, and *MPI_Get_processor_name()* respectively. The node with rank *0* will be assigned as the master.

1.3 Dividing the Matrix

When the rank set by *MPI_Comm_rank()* is 0, the program running it will be the master. It will proceed to initialize the *n x n* one-dimensional array and generate random integer elements for each slot. Upon completing the randomization, arrays are initialized for slaves and their respective offsets.

```
x = malloc(mm*n*sizeof(*x));
y = malloc(n*sizeof(*y));
p = malloc(mm*sizeof(*p));
srand(time(NULL));
for(i=0;i<mm;i++){
  for(j=0;j<n;j++){
    x[i*n+j]= rand()%10+1;
    if(i==0) y[j] = rand()%10+1;
  }
}
// Generate n dimensions and offsets to be sent to slave nodes
zm = malloc(csize*sizeof(*zm));   // slave m dimension
zmn = malloc(csize*sizeof(*zmn)); // slave m*n array size
// individual offsets on master matrix x
zmo = malloc(csize*sizeof(*zmo));
zmno= malloc(csize*sizeof(*zmno));
tmpsize= mm/csize;
tmpr   = mm%csize;
tmpo   = 0;
for (int i = 0; i < csize; i++) {
  zm[i]  = tmpsize;
  zm[i]  = (tmpr>0) ? zm[i]+1:zm[i];
  zmn[i] = zm[i]*n;
  zmo[i] = tmpo;
  zmno[i]= tmpo*n;
  tmpo  += zm[i];
  tmpr  -= 1;
}
```

After randomization, the master will scatter information to all nodes using *MPI_Bcast()* for the vector, and *MPI_Scatter()* for the matrix. All nodes will now calculate the *Pearson Correlation Coefficient* of their respective sub-arrays.

```
gettimeofday(& tstart , NULL);
MPI_Scatterv(x, zmn, zmno, MPI_INT, z, m*n, MPI_INT,ROOT,
    MPI_COMM_WORLD);
MPI_Bcast(y, n, MPI_INT, ROOT, MPI_COMM_WORLD);
pearson_cor(q, z, y, m, n);
 ...
```

1.4 Gathering Results

```
 ...
MPI_Gatherv(q, m, MPI_INT, p, zm, zmo, MPI_INT, ROOT,
    MPI_COMM_WORLD);
gettimeofday(&tend, NULL);
```

Results of the computations of all the nodes are gathered through the function *MPI_Gatherv()*. The said function's notable parameter is the first and fourth, namely the sending buffer and receiving buffer respectively. The array of calculated *Pearson Correlation Coefficient*s are stored in the sending buffer, to be received by the receiving buffer. The arrangement of elements are automatically in order, to which the implementation is abstracted by the MPI library itself.

Finally, *MPI_Finalize()* is called to terminate the program properly.

2. Results and Discussion

2.1 Exercise Limitations

The program cannot handle *n* sizes higher than 20000, due to it returning an unusual error. The *n = 20000* with *p = 1* also returns a constant error, something along invalid buffer. The program perfectly runs on other sizes though. To demonstrate sufficient results as shown in *Figure 1*, the *n* sizes were shrank into *1000*, *5000*, *10000*, *15000* and *20000*.

1000	1	0.011861	0.011152	0.011869	0.01162733333
1000	2	0.027565	0.030265	0.026783	0.02820433333
1000	3	0.028745	0.02813	0.028234	0.02836966667
1000	4	0.029855	0.030063	0.029788	0.029902
5000	1	0.2765	0.28991	0.282573	0.2829943333
5000	2	0.585951	0.571666	0.588726	0.5821143333
5000	3	0.670339	0.670634	0.669133	0.6700353333
5000	4	0.716838	0.716345	0.716725	0.716636
10000	1	0.889097	0.889422	0.910932	0.8964836667
10000	2	0.447376	0.462178	0.452888	0.4541473333
10000	3	0.296775	0.298686	0.309587	0.3016826667
10000	4	0.234451	0.237523	0.23466	0.2355446667
15000	1	2.433452	2.535784	2.417495	2.462243667
15000	2	5.096222	5.132269	5.142716	5.123735667
15000	3	5.96334	5.994345	5.991865	5.983183333
15000	4	6.402923	6.425701	6.429594	6.419406
20000	1				#DIV/0!
20000	2	17.310092	16.95787	17.777869	17.34861033
20000	3	13.854483	12.575489	13.144843	13.191605
20000	4	12.840003	12.954869	13.063963	12.952945

Figure 1. Run-time results of Pearson Correlation Coefficient via MPI

2.2 Increasing Trends

Based from running times, the *n* sizes of *1000*, *5000* and *15000* have increasing trends. The previous exercises are slightly better in comparison. However, these results are redeemed in terms of communication costs, which are counted.

2.3 Decreasing Trends

For some reason, the run-time of *n* sizes with *10000* and *20000* were decreasing as the number of processors increase, as expected in analysis.

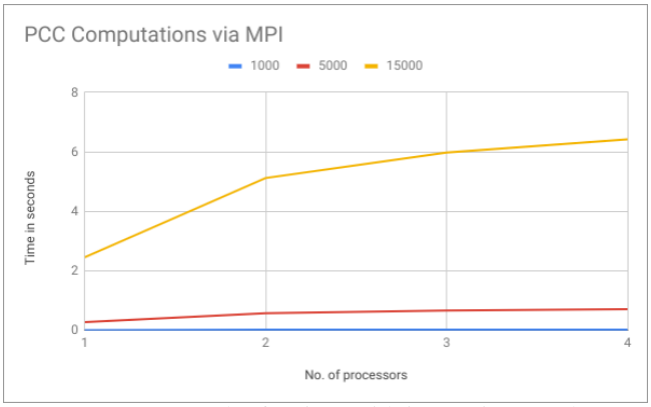

Figure 2. Graph of n sizes with increasing trends

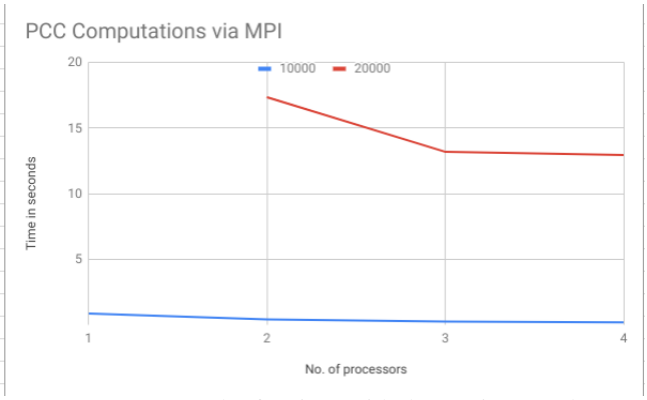

Figure 3. Graph of n sizes with decreasing trends

3. Conclusion

The results were very limited, one of the factors was due to the unstable behavior of the clustered supercomputer. Sometimes it will freeze while running, or at random times when performing commands. Another was an unusual behavior where it would not run when $n = 20000$. In spite of these, the program still ran on smaller n values. This was a challenging exercise, as the students are expected to write the program with a parallel mindset. The first step is to be familiarized in a simpler way - which was already done in the second exercise using threads. Then the MPI part will just be a bigger abstraction of threads.

4. List of Collaborators

1. Jefferson Louise Basilio - pearson computation codes
2. Jeriel G. Jaro - MPI codes
   Nothing follows

5. References

1. Pearson Product-Moment Correlation(2018). Retrieved from https://statistics.laerd.com/statistical-guides/pearson-correlation-coefficient-statistical-guide.php

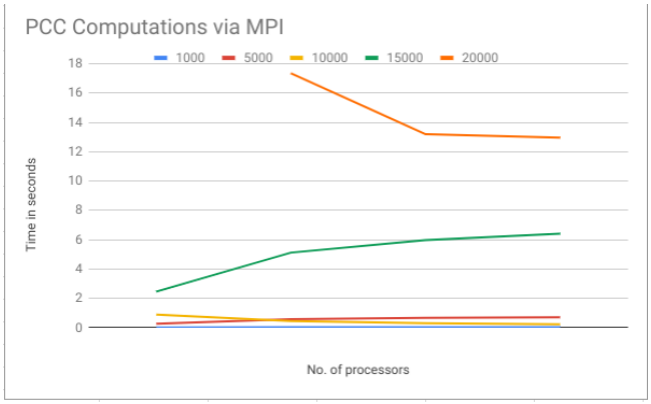

Figure 4. Overall comparison of graphs

6. Appendices

6.1 *MPI* functions

```
MPI_Comm_size(MPI_COMM_WORLD, &csize);
MPI_Comm_rank(MPI_COMM_WORLD, &crank);
MPI_Get_processor_name(cname, &clen);
 ...
MPI_Bcast(&n, 1, MPI_INT,ROOT, MPI_COMM_WORLD);
MPI_Scatter(zm, 1, MPI_INT, &m, 1, MPI_INT,ROOT,
    MPI_COMM_WORLD);
```

6.2 Timing of execution

```
...
gettimeofday(& tstart , NULL);
MPI_Scatterv(x,zmn,zmno,MPI_INT,z,m*n,MPI_INT,ROOT,
    MPI_COMM_WORLD);
MPI_Bcast(y,n,MPI_INT,ROOT,MPI_COMM_WORLD);
pearsonCC(q,z,y,m,n);
MPI_Gatherv(q,m,MPI_INT,p,zm,zmo,MPI_INT,ROOT,
    MPI_COMM_WORLD);
gettimeofday(&tend, NULL);
```