CMCS 180 First Semester, AY 2018-2019
Exercise No.2

Runtime-efficient Threaded Computation of the Pearson Correlation Coefficient

Jefferson Louise Basilio[1]*

Abstract
Users are made to input *t* threads. The matrix is divided into *n x n/t* and *n/t x n* sub-matrices. The C Programming Language along with the POSIX Thread library were used for threading implementations. Each thread is assigned to computing the Pearson Correlation Coefficient of a sub-matrix, and append the computed values to an array of global scope. Using a compiled language (C Programming Language) made it significantly faster than an interpreted one (Python 3). The matrix is modeled both modeled as column-major and row major, having an access method of *[j][i]* and *[i][j]* respectively, assuming that *i* is the iterator for the outer loop.

[1] *Institute of Computer Science, University of the Philippines Los Baños*
***Corresponding author**: 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].

In this exercise, the students are tasked with writing a program which calculates the Pearson Correlation Coefficient of a randomly generated vector and the columns of a randomly generated matrix. Asking for an integer input *n* and *t*, the program will generate an *n* x *n* matrix and an *n*-sized vector. The matrix will be then divided into chunks of sub-matrices having *n x n/t* and *n/t x n* sizes. The outputs of each sub-matrices will be combined in an array of global scope.

Objectives

The objective of this exercise is to determine the runtime by dividing the task of calculating the Pearson Correlation Coefficient of vectors of a matrix. This is done by instantiating threads simultaneously to compute the values in concurrency. Moreover, this will serve as baseline for further exercise topics such as *core-affine* computations or using *sockets*.

1. Methodology

The specifications of the machine that will be used are the following:

- Processor - Intel Core i7-7700HQ
- Memory - 8GB DDR4 2400Mhz

Additionally, processes are, if not terminated, minimized. This includes browsers, CPU monitors, text editors, and other background services. Only the terminal is left open. The formula for the Pearson Correlation Coefficient is given by the following:

Given an m x n matrix X with m rows and n columns and an m x 1 vector Y, a 1 x n vector r holds the Pearson Correlation Coefficient of the columns in X and Y, such that:

$$r(j) = \{m[X_j y] - [X]_j [y]\} / \{[m[X^2]_j - ([X])^2][m[y^2] - ([y])^2]\}^{1/2} \tag{1}$$

1.1 Matrix and Thread model

The default size *n x n* for matrix and *n*-sized vector *Y* is *25000*. To be able to implement threading, POSIX Thread library (*pthread.h*) is used. Using the said library requires the function *pearson_cor*'s multiple arguments to be enclosed in a structure show below.

```
float *pearsonValues;
typedef struct node{
  int **ownMatrixX, *vectorY;
  int startIndex, size, colSize, rowSize;
} arg;
```

Where *ownMatrixX* is the sub-matrix, *vectorY* is the local copy of vector *Y*. Other variables such as *startIndex* are for positioning the computed values in the global array, *pearsonValues*. Size variables are to keep track and guide the loops inside the function *pearson_cor()*.

The reason why *vectorY* is stored in each sub-matrix instead of just using a single global variable is performance. Generally, global variables are stored in heap memory, which are slower to access than local variables, which are stored in registers. The drawback in this is that there will be a *t-1* times *n*, where $t \geq 2$ additional memory consumption when it comes into multiple threads.

1.2 Dividing the matrix in an *n x n/t* manner

The *n* x *n* matrix is divided column-wise into *n* x *n/t* sub-matrices, with the integer *t* coming from user input. The *n* is divided by *t* to slice the matrix among the threads. The remainder sub-matrices are assigned into the *t*th thread. Only during this part are the elements of the matrix are randomized.

```
 startIndex = 0;
endIndex = subMatrices;       // n / t
rowSize = subMatrices;        // n / t
for (i = 0; i < threadSize; i++) {

  // last thread gets the remainder
  if (i == threadSize - 1) {
    colSize += remainder;
    endIndex += remainder;
  }

  // Allocate 2D array of each thread and fill with randomized
       elements
  argArray[i].ownMatrixX = (int **) malloc(size * sizeof(int*));
  for (j = 0; j < size; j++) {
    argArray[i].ownMatrixX[j] = (int *) malloc(colSize * sizeof(int
       ));
    for (k = 0; k < colSize; k++)
      argArray[i].ownMatrixX[j][k] = rand() % 10 + 1;
  }

  argArray[i].startIndex = startIndex;
  argArray[i].size = size;
  argArray[i].colSize = colSize;
  argArray[i].vectorY = vectorY;
   startIndex = endIndex;
  endIndex += subMatrices;
}
```

1.3 Dividing the matrix in an *n/t x n* manner

For this part the matrix is divided in an *n/t x n* manner, meaning, sliced row-wise. The loop iterates on each thread, and instantiates the said thread's matrix. The instantiated matrix is then populated by the inner loop. The loop stops when it reaches the maximum size of a single sub-matrix. The *t*th thread then gets the remainder rows in the matrix.

```
 startIndex = 0;
endIndex = subMatrices;       // n / t
rowSize = subMatrices;        // n / t
for (i = 0; i < threadSize; i++) {

  // last thread gets the remainder
  if (i == threadSize - 1) {
    rowSize += remainder;
    endIndex += remainder;
  }
  // Allocate 2D array of each thread and fill with randomized
       elements
  argArray[i].ownMatrixX = (int **) malloc(rowSize * sizeof(int*));
  for (j = 0; j < rowSize; j++) {
    argArray[i].ownMatrixX[j] = (int *) malloc(size * sizeof(int));
    for (k = 0; k < size; k++)
      argArray[i].ownMatrixX[j][k] = rand() % 10 + 1;
  }

  argArray[i].startIndex = startIndex;
  argArray[i].size = size;
  argArray[i].rowSize = rowSize;
  argArray[i].vectorY = vectorY;
   startIndex = endIndex;
  endIndex += subMatrices;
}
```

1.4 Starting the threads

The *t* threads are started using *pthread_create* which takes a minimum of four(4) arguments. First is the address of the thread, second are thread attributes which when set to *NULL* becomes default, third is the thread function to be executed, and fourth is the thread function argument. The thread function argument should be enclosed in a *struct* if it is more than one, hence the creation of *struct arg*.
Next, the threads are joined using *pthread_join* which takes two(2) arguments. The first is the thread itself, second is the return value of the thread. Since the values are returned directly to the *pearsonValues*, the second argument is set to *NULL*.

```
for (i = 0; i < threadSize; i++)
   pthread_create (&threadArray[i], NULL, pearson_cor,
  (void *) &argArray[i]);
for(i = 0; i < threadSize; i++)
   pthread_join (threadArray[i], NULL);
```

1.5 Function execution

Below is the core sub-function of the caller *pearson_cor()*. The difference of column and row-wise access is the way of accessing the sub-matrix. The size of the function arguments are long, because it allows all the necessary values to be computed in one run. For example, *xSummation* is an array of summation of rows/columns of the sub-matrix - values that are needed by the caller. Because the values are stored immediately in arrays, the caller does not need to compute again in the next iteration. This significantly reduces the number of instructions.

```c
void multiOperation(int **ownMatrixX, int *vectorY, int size, int
        colSize, int *Xjy, int *xSummation, int *xSquaredInside) {

  int i, j, XjyElement, summationElement, squaredInsideElement;

  i = 0;
  while (i < colSize) {
    XjyElement = 0;
    summationElement = 0;
    squaredInsideElement = 0;

    /* For n x n/t */
    for (j = 0; j < size; j++) {
      XjyElement += ownMatrixX[j][i] * vectorY[j];
      summationElement += ownMatrixX[j][i];
      squaredInsideElement += ownMatrixX[j][i] * ownMatrixX[j][i];
    }

    /* For n/t x n */
    for (j = 0; j < size; j++) {
      XjyElement += ownMatrixX[i][j] * vectorY[j];
      summationElement += ownMatrixX[i][j];
      squaredInsideElement += ownMatrixX[i][j] * ownMatrixX[i][j];
    }

    Xjy[i] = XjyElement;
    xSummation[i] = summationElement;
    xSquaredInside[i] = squaredInsideElement;
    i++;
  }
}
```

2. Results and Discussion

2.1 Column-wise Division

The results of *n x n/t* is shown in *Figure 1*. The average runtime when *t = 1* is slightly higher than of *serial* computation which is *17.19728267* seconds. This is because the instruction of creating threads and joining them takes up extra execution time in hardware level - the program is better off not having threads if the thread to be used is 1.

		C (Threaded) Column Major (n x n/t)			
		Time Elapsed (seconds)			
n	t	Run 1	Run 2	Run 3	Average
25000	1	17.143065	17.381236	17.301774	17.27535833
25000	2	12.693784	12.69638	12.695594	12.69525267
25000	4	10.836387	10.825037	10.843464	10.83496267
25000	8	9.959742	9.915792	9.896817	9.924117
25000	16	9.319824	9.319824	9.318312	9.31932
25000	32	8.937523	8.896809	8.898288	8.910873333
25000	64	8.009972	7.957888	8.029328	7.999062667

Figure 1. The runtime results of *n x n/t* computation

2.2 Row-wise Division

Shown in *Figure 3* are the results of dividing the matrix in an *n/t x n* manner. The results are significantly - more than

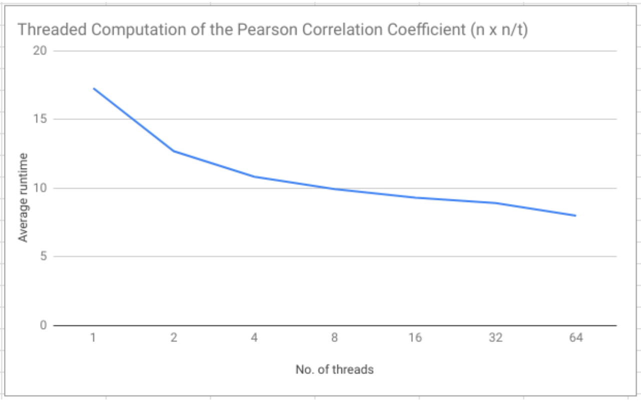


Figure 2. Line graph of *Figure 1*

900% improvement when *t = 64*. However, the runtime stops improving starting from *t = 8*. TheseN results effectively proves that storing and accessing values row-wise induces *cache-hits*. The C programming language is designed such as it stores values row-wise contiguously, therefore accessing them is more convenient as they are beside each other in memory. Line graph comparison of the two is shown in *Figure 4*.

		C (Threaded) Row Major (n/t x n)			
		Time Elapsed (seconds)			
n	t	Run 1	Run 2	Run 3	Average
25000	1	3.341198	3.341166	3.34182	3.341394667
25000	2	1.670442	1.671951	1.677379	1.673257333
25000	4	0.889401	0.898412	0.894184	0.893999
25000	8	0.885947	0.867876	0.8686	0.874141
25000	16	0.878883	0.866433	0.855991	0.8671023333
25000	32	0.880214	0.879331	0.861484	0.8736763333
25000	64	0.861983	0.868531	0.870947	0.8671536667

Figure 3. The runtime results of *n x n/t* computation

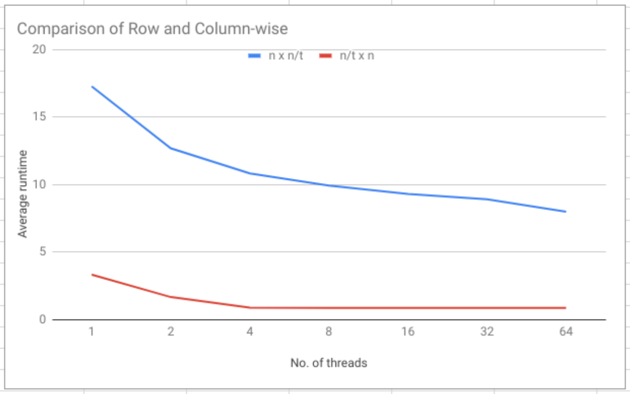
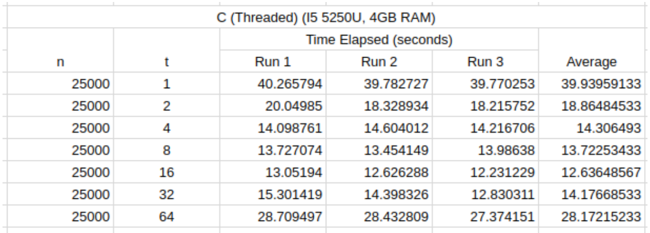


Figure 4. Line graph comparison of Row and Column-wise

2.3 Better Hardware

Better hardware merits better runtime. For additional information, the program was ran on a laptop with different hardware composition. Specifically a Macbook Air which has an *Intel Core i5-5250U* and a *4GB DDR3 1600MHz* memory. Results are shown below.



C (Threaded) (I5 5250U, 4GB RAM)					
		Time Elapsed (seconds)			
n	t	Run 1	Run 2	Run 3	Average
25000	1	40.265794	39.782727	39.770253	39.93959133
25000	2	20.04985	18.328934	18.215752	18.86484533
25000	4	14.098761	14.604012	14.216706	14.306493
25000	8	13.727074	13.454149	13.98638	13.72253433
25000	16	13.05194	12.626288	12.231229	12.63648567
25000	32	15.301419	14.398326	12.830311	14.17668533
25000	64	28.709497	28.432809	27.374151	28.17215233

Figure 5. Program runtime on a *Macbook Air (Early 2015) Model*

2.4 Amdahl's Law

As seen in both results, the number of threads and the running time is inversely proportional. One might be wondering why the running time is not halved when the resources are doubled, which can be covered by *Amdahl's law*. Amdahl's law calculates the maximum theoretical speedup of a program given additional resources[2]. Having twice the resources does not halve the execution time of a program.

3. Conclusion

From the previous exercise, Python was used with the library *numpy*. The runtime results are significantly less even in serial computation with C. This is due to the difference of compiled and interpreted language. GCC has a built-in optimizers[3] and allows the programmer's code to be compiled in its barest form. Python on the other hand is slower due to being an interpreted language, but makes it up for being easy to read, understand and write. Going back, results showed that row-wise implementations when working with arrays are empirically better than column-wise ones. C stores the values in memory contiguously, so accessing them next to each other is faster - while considering the utilization of the processor's cache. To add, the machine stops responding when $n \geq 35000$. Theoretically the machine should be able to handle such size, because *malloc()* takes memory from the heap. However in this instance, the machine slowly hangs and completely stops after 5-10 seconds

4. List of Collaborators

1. Jefferson Louise Basilio
   Nothing follows

5. References

1. Pearson Product-Moment Correlation (2018). Retrieved from https://statistics.laerd.com/statistical-guides/pearson-correlation-coefficient-statistical-guide.php
2. Amdahl's Law(n.d.). Retrieved from https://www.techopedia.com/definition/17035/amdahls-law
3. Glen (2009). How many GCC optimization levels are there?. Retrieved from https://www.techopedia.com/definition/17035/amdahls-law

6. Appendices

6.1 *pearson_cor()* function

```c
void *pearson_cor(void *args) {

  arg *argLocal = (arg *) args;
  int size = argLocal->size;
  int colSize = argLocal->colSize;
  int startIndex = argLocal->startIndex;
  int **ownMatrixX = argLocal->ownMatrixX;
  int *vectorY = argLocal->vectorY;

  int *Xjy = (int *) malloc(size * sizeof(int));
  int *xSummation = (int *) malloc(size * sizeof(int));
  int *xSquaredInside = (int *) malloc(size * sizeof(int));
  int i, mxjy, xjy, y, ySquaredInside, ySquared, numerator;
  float denominator1, denominator2;


  // Perform matrix to vector multiplication , also get the sum of
      columns, and the sum of squared elements of columns
  multiOperation(ownMatrixX, vectorY, size, colSize, Xjy,
      xSummation, xSquaredInside);
  y = sumOfVector(vectorY, size);
  ySquaredInside = sumOfSquaredVector(vectorY, size) * size;
  ySquared = y * y;

  i = 0;
  while( i < colSize ) {
    mxjy = size * Xjy[i];
    xjy = xSummation[i] * y;
    numerator = mxjy - xjy;
    denominator1 = size * xSquaredInside[i] - xSummation[i] *
      xSummation[i];
    denominator2 = ySquaredInside - ySquared;
    pearsonValues[ startIndex ] = numerator / sqrt(denominator1 *
      denominator2);

    startIndex ++;
    i++;
  }
  free(Xjy);
  free(xSummation);
  free(xSquaredInside);
  pthread_exit(NULL);
}
```