Core-affine Threaded Computation of the Pearson Correlation Coefficient

Jefferson Louise Basilio[1]*

Abstract
The program from *Exercise02* was modified to associate each thread into the processor's *m* cores. The user is made to input *t* threads. The matrix was both divided column-wise and row-wise, specifically, it was divided into *n x n/t* and *n/t x n* sub-matrices - each assigned to the threads created. The threads containing the sub-matrices are then assigned to cores. The remainder sub-matrices are assigned to the *t*th thread. Likewise, the remainder threads are assigned to the *m*th core of the processor. The C Programming Language along with the POSIX Thread library were used for threading implementations. For scheduling, the library *sched.h* was used.

[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].

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 and assign the said threads into the processor's cores. By explicitly assigning the threads into the cores means the processor has more efficiency on utilizing its caching system.

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:

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

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 , endIndex, size , rowSize, cpuCore;
} 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 integer variable *cpuCore* is the core to assign the thread into. It has a range from *0* to *m - 1*, meaning if the processor has *8* logical cores, only *7* shall be explicitly utilized according to the exercise instructions.

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 row-wise and column-wise

The *n* x *n* matrix is divided column-wise into *n* x *n/t* sub-matrices, with the integer *t* coming from user input. Likewise, the matrix is also divided row-wise into *n/t x n*. 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.

```
// Divide each thread into number of cores
noOfCores = 7;      // Number of logical cores on the machine
threadPerCore = threadSize / noOfCores;

startIndex = 0;
endIndex = subMatrices;     // n/t
colSize = subMatrices;      // n x n/t
rowSize = subMatrices;      // n/t x n
currentCore = 0;        // Starting core
for (i = 0; i < threadSize; i++) {

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

  // Create 2D array for each thread and fill

  /* n x n/t */
  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;
    }
  }

  /* n/t x n */
  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].colSize = colSize;      // n x n/t
  argArray[i].rowSize = rowSize;      // n/t x n
  argArray[i].vectorY = vectorY;
  argArray[i].cpuCore = currentCore;   // Assign which core to the
      thread
  startIndex = endIndex;
  endIndex += subMatrices;

  // Increment the core; Last core gets the remainder threads
  // much like how the last thread gets the remaining submatrices
  if (((i+1) % threadPerCore == 0) && currentCore != noOfCores - 1)
    currentCore++;

}
```

1.3 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);
```

1.4 Setting the Core Affinity

The library *sched.h* is needed to be able to assign the threads to a specific core. On the code itself, only four (4) lines are needed to manipulate the CPU scheduling. The code is shown below.

```
#define _GNU_SOURCE
#include <sched.h>  // cpu_set_t , CPU_SET
...
void *pearson_cor(void *args) {
```

```
5
6   arg *argLocal = (arg *) args;
7   cpu_set_t cpuset;
8
9   // Clears the cpuset
10  CPU_ZERO(&cpuset);
11
12  // Set CPU core on cpuset
13  CPU_SET(argLocal->cpuCore, &cpuset);
14
15  // Schedule
16   sched_setaffinity (0, sizeof(cpuset), &cpuset);
17
18   ...
19  }
```

Where *CPU_ZERO()* takes the address of a variable with data type cpu_set_t, to clear the CPUs it contains[2]. *CPU_SET()* which takes two arguments, first is an integer, for the specific CPU core. 0 is first core, 1 is second, and so on. The second argument is the address of the *cpuset*. The *sched_setaffinity()* is the one responsible for the scheduling itself.

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.

```
1  void multiOperation(int **ownMatrixX, int *vectorY, int size, int
       colSize, int *Xjy, int *xSummation, int *xSquaredInside) {
2
3    int i, j, XjyElement, summationElement, squaredInsideElement;
4    i = 0;
5    while ( i < colSize ) {
6      XjyElement = 0;
7      summationElement = 0;
8      squaredInsideElement = 0;
9
10     /*  n x n/t  */
11     for (j = 0; j < size; j++) {
12       XjyElement += ownMatrixX[j][i] * vectorY[j];
13       summationElement += ownMatrixX[j][i];
14       squaredInsideElement += ownMatrixX[j][i] * ownMatrixX[j][i];
15     }
16
17     /*  n/t x n  */
18     for (j = 0; j < size; j++) {
19       XjyElement += ownMatrixX[i][j] * vectorY[j];
20       summationElement += ownMatrixX[i][j];
21       squaredInsideElement += ownMatrixX[i][j] * ownMatrixX[i][j];
22     }
23
24     Xjy[i] = XjyElement;
25     xSummation[i] = summationElement;
26     xSquaredInside[i] = squaredInsideElement;
27     i++;
28   }
29 }
```

1.6 Checking CPU Utilization

The Linux command *top* is used to view the processes and utilization of resources. The *top* is inputted after running the program. This is used to test the validity of the code, if it really overrides the operating system in assigning which core to use to run the program. .

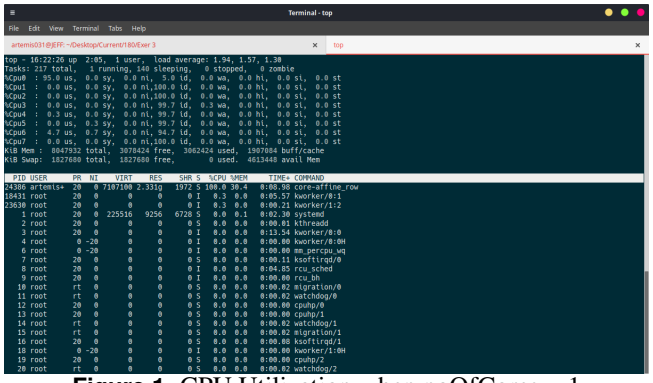

Figure 1. CPU Utilization when noOfCores = 1

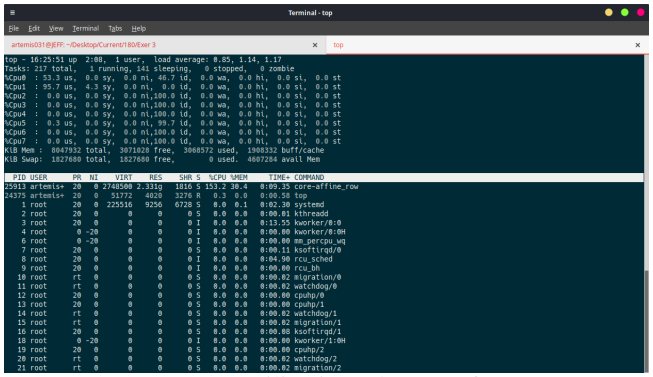

Figure 2. CPU Utilization when noOfCores = 2

Looking at the results of *top*, only the *%Cpu0* was used in *Figure 1*. In *Figure 2*, only *%Cpu0* and *%Cpu1* was used. This means that the code is working and valid.

2. Results and Discussion

2.1 Column-wise Division

The results of *n x n/t* is shown in *Figure 3*. 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.

2.2 Row-wise Division

Shown in *Figure 5* are the results of dividing the matrix in an *n/t x n* manner. The results are significantly - more than *700%* improvement when *t = 64*. However, the runtime stops improving starting from *t = 8*. These results effectively proves that storing and accessing values row-wise induces *cache-hits*. The C programming language is designed such as it stores

C (Threaded) with core-affinity n x n/t					
		Time Elapsed (seconds)			
n	t	Run 1	Run 2	Run 3	ore-affine averag
25000	1	16.925972	17.018621	17.168565	17.03771933
25000	2	12.698874	12.606614	12.620695	12.642061
25000	4	10.804455	10.807434	10.827442	10.81311033
25000	8	9.592308	9.597181	9.579596	9.589695
25000	16	9.325922	9.273414	9.335108	9.311481333
25000	32	9.399963	9.381204	9.429192	9.403453
25000	64	8.36272	8.390736	8.32236	8.358605333

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

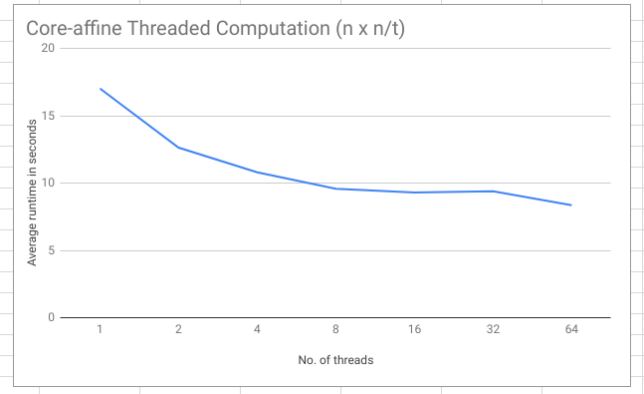

Figure 4. Line graph of *Figure 3*

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 6*.

C (Threaded) with core-affinity n/t x n					
		Time Elapsed (seconds)			
n	t	Run 1	Run 2	Run 3	Core-affine
25000	1	3.345809	3.344565	3.345099	3.345157667
25000	2	1.677757	1.67374	1.678641	1.676712667
25000	4	0.880793	0.880528	0.881291	0.8808706667
25000	8	1.280991	1.298859	1.286708	1.288852667
25000	16	1.326494	1.327319	1.301183	1.318332
25000	32	1.359547	1.339057	1.328145	1.342249667
25000	64	1.134848	1.171386	1.070461	1.125565

Figure 5. The runtime results of *n x n/t* computation

2.3 Previous Exercise

Figure 7 shows the line graph comparison of the previous row-wise Threaded vs Core affine threaded. The running time of the two were the same until $n > 4$, in which the core-affine became slower.

3. Conclusion

Recurring results from previous exercises verifies that row-wise implementations of tasks involving 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. In this exercise, results proved that there is a slight difference when explicitly assigning the threads to specific cores, so the said cores will have no trouble locating the cache stored when the program is ran. However, when $t > 4$, it becomes slower

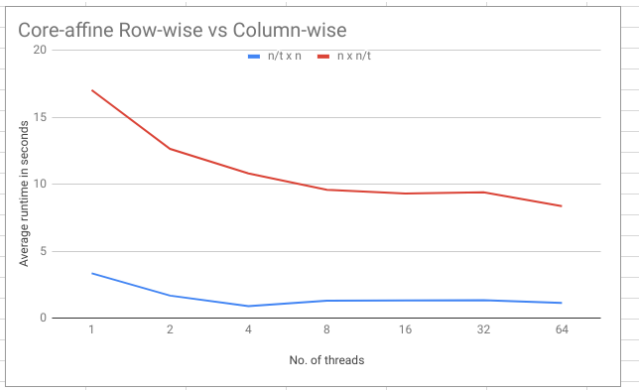

Figure 6. Comparison of n x n/t vs n/t x n

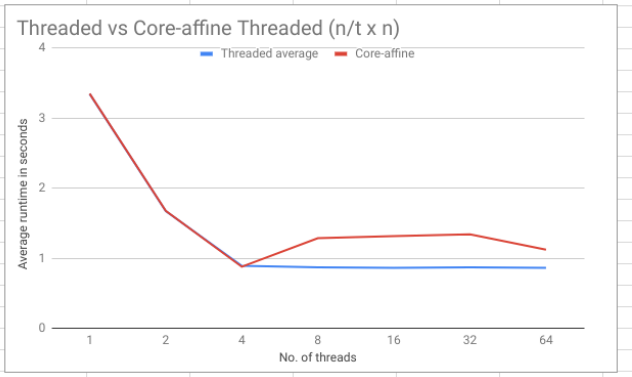

Figure 7. Comparison of present and previous exercise

than the non-core-affine one. Lots of factors can attest into this result. One is the CPU resources. Unavoidably, the machine runs other background programs when running the program. Another would be the scheduling and queuing protocols of the CPU - underneath the abstractions are the implementations of how the CPU itself manages its tasks per clock ticks, which cannot be manipulated for this level of exercise. Next would be *serial bottlenecks* - a factor that can hinder performance when there is a part of the program requiring serial accesses to resources[3]. To add, the machine stops responding when $n \geq 35000$. The threads that the machine can instantiate on the other hand is up to $t \leq 4500$. 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. The best performance time out of all the three exercises was that of the previous one (*Exercise 02*).

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. cpu_zero(3) - Linux man page (n.d.). Retrieved from https://linux.die.net/man/3/cpu_zero
3. Fisher, M. (2016). Why setting CPU affinity make threads run slower?. Retrieved from https://stackoverflow.com/questions/39495136/why-settting-cpu-affinity-make-threads-run-slower

6. Appendices

6.1 *pearson_cor()* function

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

  arg *argLocal = (arg *) args;

   cpu_set_t  cpuset;
  CPU_ZERO(&cpuset);          // clears the cpuset
  CPU_SET(argLocal->cpuCore, &cpuset); //set CPU core on cpuset
   sched_setaffinity (0, sizeof(cpuset), &cpuset);

   int  size = argLocal->size;
   int rowSize = argLocal->rowSize;
   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, rowSize, Xjy, xSummation, xSquaredInside);
  y = sumOfVector(vectorY, size);
  ySquaredInside = sumOfSquaredVector(vectorY, size) * size;
  ySquared = y * y;

  i = 0;
  while( i < rowSize ) {

     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);
}
```