Abstract

Processor affinity was defined as the means of attempting to transfer a thread to a different processor ("Processor affinity and binding," 2022). This type of implementation binds the thread to a specific core. In the article "Processor affinity and binding" (2022), binding was defined as the means of dispatching a thread to a specific processor, regardless of the status of the other processors. As a program gets more complex and dynamic, managing threads is a way to improve the program's performance, hence processor affinity can greatly impact a program's runtime speed.

Introduction

The algorithm for last week's exercise, with minor adjustments, was used for this week's exercise of processor affinity. The machine used to run the program runs on an Intel Core i7-8700 CPU @3.20GHz x 12 with a 16GiB RAM using an Ubuntu operating system and 1TB of disk capacity.

The C language was used to implement the program, for uniformity as it is the language used to implement the past two exercises. The objective is still arriving with the Pearson Correlation Coefficient (PCC) vector using threads, but with the addition of assigning them to specific processors or cores. Since the row-major implementation performed better in the previous exercise, the core affinity concept was applied to it to observe if the runtimes can perform even better. Additionally, 2d arrays and vectors of bigger sizes have been tested to see the program's behavior at even higher sizes.

Objectives

The PCC vector solves uses the multithreaded, row-wise implementation with the attempt of making it core affine. This utilizes the machine's resources even more, while observing if it can perform better. The specific objectives of the experiment have been listed below.

1. Randomize values inside matrix **X** of size *n by n* and vector **y** of size *1 by n*.
2. Identify the machine's number of processors and assign the created **t** threads that will solve for the PCC vector of **n/t** number of subroutines.
3. Partition the 2d array into *n by n/t* subarrays.
4. Store values in a vector **r** of size *1 by n*.
5. Implement core affinity to the row-wise implementation of the PCC solver.
6. Measure the time it takes to solve for the PCC vector as **t** gets higher after it gets assigned into different cores.

7. Observe the runtime values as **t** and **n** gets higher and compare it with the previous weeks' implementations.

8. Test the limit of the machine.

Methodology

The aforementioned specifications were used to run the threaded implementation of the PCC solver starting from thread value of 1 up to 64. The implementation had 3 major functions in total:

- The function `pearson_cor_threaded()` is concerned with the initialization of the processors and creation of threads. The total processor count available was determined using the `get_nprocs()` function. The function also manages the arguments that will be passed on to the helper function. The arguments identifies how the 2d array will be partitioned and which subarray will be sent to each instance of `pthread_create()`. The function is also concerned with binding the threads to specific cores after they have been spawned. Spawned threads are also joined in the function.

- The function `pearson_cor_threaded_helper()` is a helper function to the function mentioned above. The helper function is called in every instance of `pthread_create()`. It solves for the terms of the formula first, then computes for the actual PCC value given the terms computed. The function takes in **`void* arguments[i]`** from the fourth parameter of `pthread_create()`, and is then typecasted into an **`int*`** type. The **`int* temp`** variable holds the index where the subroutine should start and end, as well as the size **`n`** of the 2d array. The row-wise implementation of the PCC solver was retained given that it is the better-performing implementation. The column-wise implementation has been removed.

- The `main()` function contains the memory allocation for the 2d array **X** and vector **y**. The randomization of values in both **X** and **y** is executed in this function. To limit the value being stored, a range of 1 to 999,999 was used. Again, since the program follows the row-wise implementation, the transpose code snippet was also retained. The `gettimeofday()` function was used to get the wall time of the execution, printing out the recorded runtime.

- The `deallocate()` function handles the deallocation of the memory assigned to the pointers during execution.

Results and Discussion

The runtimes of the program given an n by n 2d array, 1 by n vector, and t number of threads assigned to specific cores are shown in the tables and illustrations below. The size of the 2d array and vector was set to 25000 initially. Previously, the partition was defined by dividing the rows by

the total number of threads. Again, looking back at the previous runtime values obtained using this approach, we have proved that the runtimes in this approach are much slower than the row-wise implementation. Instead of creating n/t by n sizes of subarrays, we opted to go with n by n/t sizes of partitions.

The total processor count outputted by the `get_nprocs()` function was used to utilize all available processors, distributing all threads equally among them. The runtime values below show the performance of the core-affine implementation of the multithreaded PCC solver.

N	t	Time Elapsed (in seconds)			Average Runtime
		Run 1	Run 2	Run 3	
25,000	1	23.9776 seconds	23.6160 seconds	24.6265 seconds	24.0734 seconds
25,000	2	12.4182 seconds	12.2926 seconds	38.5726 seconds	38.3745 seconds
25,000	4	6.2529 seconds	6.5092 seconds	6.3983 seconds	6.3868 seconds
25,000	8	4.0934 seconds	4.1163 seconds	4.1227 seconds	4.1108 seconds
25,000	16	3.2197 seconds	3.0637 seconds	3.2622 seconds	3.1819 seconds
25,000	32	3.0241 seconds	3.1322 seconds	3.0796 seconds	3.0786 seconds
25,000	64	3.0395 seconds	3.0842 seconds	3.0449 seconds	3.0562 seconds

Table 1. Runtimes of the row-wise, core-affine threaded implementation of the PCC solver on a 25000 by 25000 2d array.

Looking at the table, the starting average runtime took only 24.0734 seconds to complete. Compared to the previous week's 74.9428 seconds runtime using the same size, and Week 1's 44.710 seconds runtime at 2d array size of 20000 by 20000, we can observe that the core-affine implementation performed the best even when using a single thread. We can also see that at 64 threads, the core-affine implementation only took 3.0562 seconds on average to execute, while the previous week took 8.5816 seconds to execute. Starting from thread count of 1 to 64, the core-affine implementation performed much better by a huge margin.

Refer to the illustration below to see the trend of the runtimes.

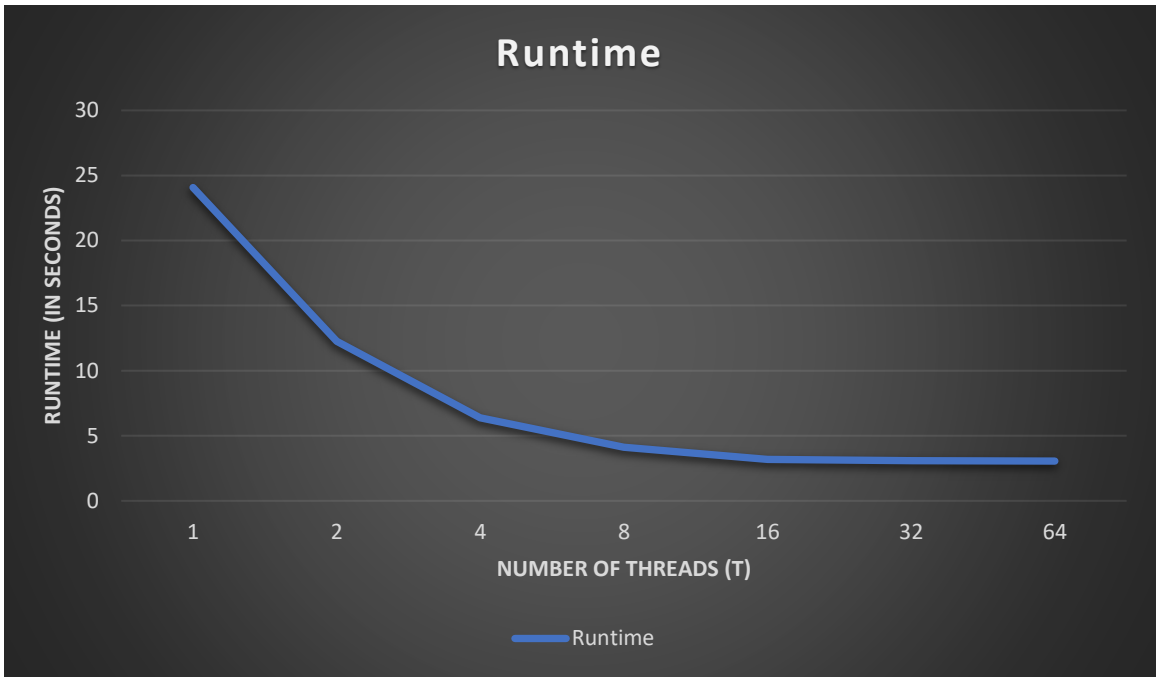

Chart 1. Line graph representation of the runtimes of the row-wise, core-affine implementation on a 25000 by 25000 2d array as t number of threads increases.

The trend is still downwards, same as the previous weeks' implementation but the core affine implementation starts at a much lower runtime while also ending in a lower runtime value. Additionally, the trend of the line graph from thread count of one to eight has a steep shape, while thread count of 16 onwards starts to straighten out, meaning that the change gets lower as the thread count increases. This can also be observed in the previous weeks' implementation.

The implementation was also applied on a 30000 by 30000 2d array and 1 by 30000 vector. The runtime values are as follows:

N	t	Time Elapsed (in seconds)			Average Runtime
		Run 1	Run 2	Run 3	
30,000	1	33.7077 seconds	33.8215 seconds	33.6847 seconds	33.7380 seconds
30,000	2	17.3353 seconds	17.3258 seconds	17.3387 seconds	17.3333 seconds
30,000	4	8.9900 seconds	8.9631 seconds	8.9839 seconds	8.9790 seconds
30,000	8	5.9402 seconds	5.9033 seconds	5.9078 seconds	5.9171 seconds

30,000	16	4.7996 seconds	4.8382 seconds	4.7603 seconds	4.7994 seconds
30,000	32	4.6145 seconds	4.6961 seconds	4.6565 seconds	4.6557 seconds
30,000	64	4.6150 seconds	4.4368 seconds	4.3661 seconds	4.4726 seconds

Table 2. Runtimes of the row-wise, core-affine threaded implementation of the PCC solver on a 30000 by 30000 2d array.

The observation with the 30000 by 30000 2d array is the same as the 25000 by 25000, whereas the core-affine implementation performs two to three times better than the simple multithreaded program. Comparing the runtimes, we can see that using a single thread in the core-affine implementation only takes 33.7380 seconds on average to complete while the simple multithreaded implementation takes 105.1970 seconds. Meanwhile, using 64 threads give us 4.4726 seconds in the core-affine implementation, while the simple multithreaded gives us 14.4996 seconds.

To compare it further, the newly-gathered runtimes are put against the previous week's runtimes. The chart below illustrates the shift in the graph.

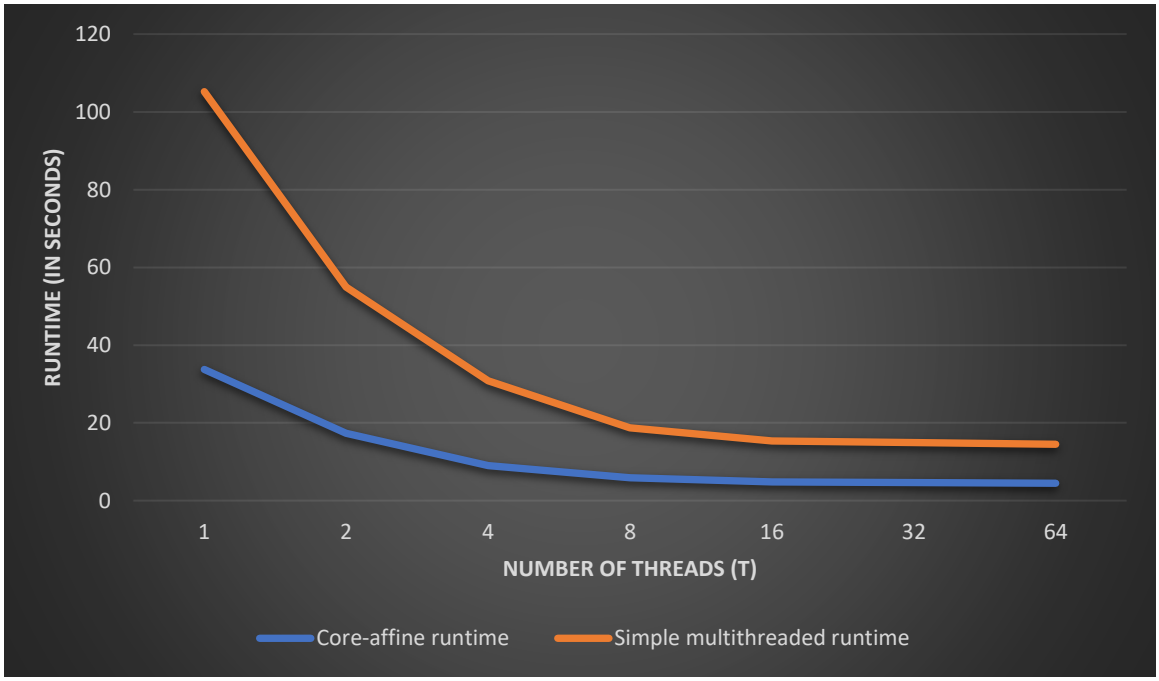

Chart 2. Line graph representation of the runtimes of the row-wise, core-affine implementation on a 30000 by 30000 2d array against the simple multithreaded implementation as t number of threads increases.

As observed in the graph above, both implementations follow the downward trend, but the simple multithreaded runtime shows a steeper change from thread count one to eight. Additionally, we can also observe the new runtimes shifting downwards, which gives us the impression that the current implementation has a significant performance boost.

Given the improvement with the runtime values, we were able to test bigger 2d array sizes. The table below shows the runtime values in a 40000 by 40000 2d array.

N	t	Time Elapsed (in seconds)			Average Runtime
		Run 1	Run 2	Run 3	
40,000	1	60.1526 seconds	59.7985 seconds	59.899 seconds	59.9500 seconds
40,000	2	30.8124 seconds	30.753 seconds	30.759 seconds	30.7748 seconds
40,000	4	15.9466 seconds	16.2062 seconds	16.2499 seconds	16.1342 seconds
40,000	8	11.5025 seconds	11.5548 seconds	11.7726 seconds	11.6100 seconds
40,000	16	8.3033 seconds	8.6785 seconds	8.9710 seconds	8.6509 seconds
40,000	32	7.8801 seconds	7.9328 seconds	8.0497 seconds	7.9542 seconds
40,000	64	7.7173 seconds	7.9303 seconds	7.9849 seconds	7.8775 seconds

Table 2. Runtimes of the row-wise, core-affine threaded implementation of the PCC solver on a 40000 by 40000 2d array.

Comparing the runtimes above with the previous week's results, we can see that though the 2d array size was doubled, the current implementation still performed better. The massive performance boost can be observed better in this example, as the current runtimes are still faster even at a higher size. In fact, the same can also be observed in a 50000 by 50000. The table below shows the runtime.

N	t	Time Elapsed (in seconds)			Average Runtime
		Run 1	Run 2	Run 3	
50,000	1	93.6793 seconds	93.7307 seconds	93.8238 seconds	93.7446 seconds
50,000	2	48.1926 seconds	48.7372 seconds	49.0511 seconds	48.6603 seconds
50,000	4	25.0281 seconds	24.9601 seconds	25.7321 seconds	25.2401 seconds

50,000	8	17.0132 seconds	16.9899 seconds	17.1127 seconds	17.0386 seconds
50,000	16	13.3754 seconds	13.2314 seconds	13.5125 seconds	13.3731 seconds
50,000	32	12.9863 seconds	13.1446 seconds	13.0721 seconds	13.0677 seconds
50,000	64	13.1651 seconds	12.2041 seconds	12.6989 seconds	12.6894 seconds

Table 3. Runtimes of the row-wise, core-affine threaded implementation of the PCC solver on a 50000 by 50000 2d array.

When compared to the simple threaded implementation, the core-affine implementation is still faster even though the 2d array and vector size were doubled. The current implementation ran almost 11 seconds faster than the simple threaded implementation. A clearer comparison can be inferred from the illustration shown below.

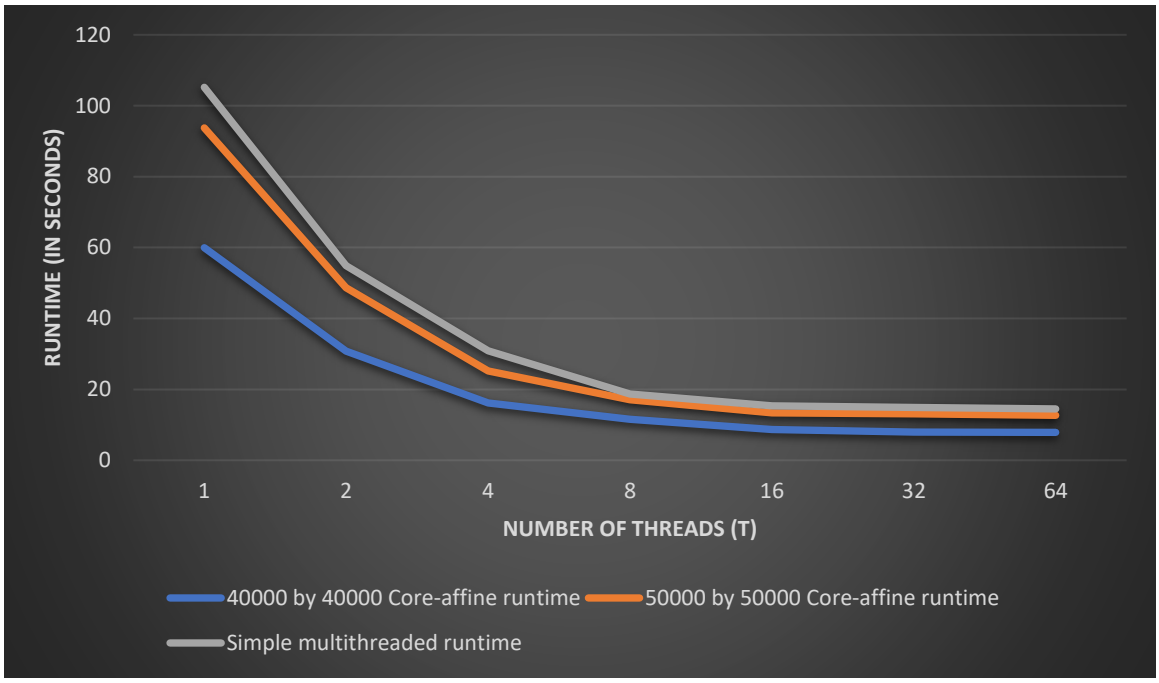

Chart 3. Line graph representation of the runtimes of the row-wise, core-affine implementation on a 50000 by 50000 and 40000 by 40000 2d arrays against the simple multithreaded implementation as t number of threads increases.

Given that the implementations had similar observations, looking at a line graph would better give a clear indication on the performance of the algorithm. Again, same as the previous line graphs, the trend of the runtimes is always downwards, starting to straighten out at thread count 8.

However, due to time constraints, the 2d array and vector sizes were limited to size 50000 by 50000. While it looks like the runtimes are low, the time it takes to allocate values and memory to

the 2d array and vector is quite long. Furthermore, spawning threads, assigning them to cores, and joining them also require quite some time. Nevertheless, the possibility of running the algorithm on an even higher size of 2d array and vector is very high, unlike the previous implementations since the resources are managed well and the load is equally distributed this time around.

Conclusion

In conclusion, the core-affine implementation dramatically reduced the runtime of the program. This is a very important tool to have, especially when concerned with the performance of the application. For memory-intensive programs and experiments like these, speed matters a lot especially when dealing with large input sizes.

Finding the right balance of the concepts discussed throughout the weeks should not be dismissed given that there are instances when there are more resources allocated than necessary. Threads must be used with some reservations – that is, the thread count must be directly proportional to the size of the input data.

Applying this in a real-world scenario, end users have the power when it comes to selecting which application suits them the most. Naturally, performance is valued highly when selecting an application since it saves the time of the use. Applying this concept in every application being developed must be a priority because it utilizes the full potential of the machine. However, this implementation really pushed the machine to the edge given that it used all the machine's available processors. Practicing mastery in this technique will allow the developer to manage the resources properly by giving the application a good performance without deeming the machine unusable.

References

Processor affinity and binding. (2022, May 16). IBM. Retrieved November 11, 2022, from https://www.ibm.com/docs/en/aix/7.2?topic=architecture-processor-affinity-binding

APPENDIX A

Source code for the threaded implementation of the PCC vector solver

```
#define _GNU_SOURCE
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <pthread.h>
#include <sched.h>
#include <sys/sysinfo.h>


float *pccvector;
int **X, *y;

/*
    Helper function for solving pearson correlation coefficient vector.
    Solves for the vector values by computing for each term.
     and stores it inside the pccvector array.
*/
void * pearson_cor_threaded_helper(void *position)
{
    int i, j, temp1, temp2, temp3, temp4, temp5, *temp;
    temp = (int *) position;
    temp1=0, temp2=0, temp3=0, temp4=0, temp5=0;

    //printf("startcol{%d}: endcol:{%d} rows:{%d}\n", temp[0], temp[1], temp[2]);
    for(i=temp[0]; i<temp[1]; i++){
        for(j=0; j<temp[2]; j++){
            temp1 += X[i][j];
            temp2 += pow(X[i][j], 2);
            temp3 += y[j];
            temp4 += pow(y[j], 2);
            temp5 += X[i][j]*y[j];
        }

        //printf("\n%d, %d, %d, %d, %d", temp1,temp2,temp3,temp4,temp5);
        pccvector[i] = (((temp[1])*temp5) - (temp1*temp3)) /
pow(((((temp[1])*temp2) - pow(temp1,2))*(((temp[1])*temp4) - pow(temp3,2))),
0.5);
    }

    printf("\n");
```

```
}

/*
    Main function for pearson correlation coefficient vector.
    Manages threading and execution of the helper function.
*/
void * pearson_cor_threaded(int m, int n, int t)
{
    int flag=0, i, j, subroutine_count=n/t, **arguments, temp;
    float *v;

    // //PRINTING 2D ARRAY VALUES
    // printf("\n\nRANDOMIZED %d by %d MATRIX\n", m,n);
    // for(i=0; i<n; i++){
    //     for(j=0; j<m; j++){
    //         printf("%d\t", X[i][j]);
    //     }
    //     printf("\n");
    // }


    // //PRINTING VECTOR VALUES
    // printf("\n\nRANDOMIZED 1 by %d VECTOR\n", m);
    // for(i=0; i<n; i++){
    //     printf("%d\t", y[i]);
    // }
    // printf("\n");

    // PCC VECTOR
    v = (float*)malloc(sizeof(float)*n);

    // CPU INITIALIZATION
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    // THREAD INITIALIZATION
    pthread_t tid[t];
    arguments = (int **)malloc(sizeof(int*)*t);

    temp = 0;
    for(i=0; i<t; i++){
        arguments[i] = (int*)malloc(sizeof(int)*3);
        arguments[i][0] = temp;
        arguments[i][1] = temp+subroutine_count;
        arguments[i][2] = n;
```

```
        temp += subroutine_count;
    }

    int proc_count = get_nprocs();

    // LOOP FOR CREATING THREADS
    for(i=0; i<t; i++){

        printf("thread# %d executing... \n", i+1);
        temp = i%t;
        CPU_SET(temp, &cpuset);
        pthread_create(&tid[i], NULL, pearson_cor_threaded_helper, (void *)
arguments[i]);
        pthread_setaffinity_np(tid[i], sizeof(cpu_set_t), &cpuset);

    }

    // JOINS THREADS
    for(i=0; i<t; i++){
        pthread_join(tid[i], NULL);
    }
}

// FREES MEMORY ALLOCATION
void deallocate(int size)
{
    for(int i=0; i<size; i++){
        free(X[i]);
    }
    free(X);
    free(y);
}


int main()
{
    FILE *fptr;
    int i, j, size, num_of_threads, temp;
    int *v;

    printf("Enter matrix size: ");
    scanf("%d", &size);
    printf("Enter number of threads: ");
    // scanf("%d", &num_of_threads);
```

```
    //allocates memory for matrix and vectors
    X = (int**)malloc(sizeof(int*)*size);
    y = (int*)malloc(sizeof(int)*size);
    pccvector = (float*)malloc(sizeof(float)*size);

    for(i=0; i<size; i++){
      X[i] = (int*)malloc(sizeof(int*)*size);
    }

    //assigns random values to matrix and vector
    for(i=0; i<size; i++){
        y[i] = 1+rand()%1000000;
        for(j=0; j<size; j++){
            X[i][j] = 1+rand()%1000000;
        }
    }

    // TRANSPOSES THE MATRIX
    for(i=0; i<size; i++){
        for(j=0; j<i; j++){
            temp = X[i][j];
            X[i][j] = X[j][i];
            X[j][i] = temp;
        }
    }

    // STARTS CLOCK
    struct timeval start, end;
    gettimeofday(&start, 0);

    pearson_cor_threaded(size, size, num_of_threads);

    gettimeofday(&end, 0);
    long seconds = end.tv_sec - start.tv_sec;
    long microseconds = end.tv_usec - start.tv_usec;
    double elapsed = seconds + microseconds*1e-6;

    //COMPUTES TIME TAKEN TO RUN
    printf("Execution time: %.4f\n", elapsed);
    deallocate(size);
}

/*
    References:
        Measuring execution time (wall time) :
```

```
            https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-
c-c-48634458d0f9
        C/C++: Set Affinity to process thread - Example Code:
            https://bytefreaks.net/programming-2/c/cc-set-affinity-to-process-
thread-example-code
        22.3.5 Limiting execution to certain CPUs:
            https://www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html
*/
```