HELLENIC REPUBLIC
National and Kapodistrian
University of Athens
EST. 1837

Department of Physics
COURSE: COMPUTER SYSTEMS

POSIX THREADS

Comparing Performance of POSIX Threads in Squaring Arrays in C

Author:
Christos Koromilas

Student Number:
201100079

1 Project Description

This project involves developing a program to create and manipulate arrays using POSIX threads. We constructed an array A with $A(1) = 1, A(2) = 2, \ldots, A(10^6) = 10^6$ and calculated $B[i] = A[i]^2$ using both 1 and 4 threads. The performance was compared using a time counter across 1000 executions to enhance accuracy.

1.1 Utilization of Threads

Using threads enables concurrent execution of multiple processes, thereby enhancing computational efficiency. Proper management is crucial to avoid race conditions.

1.1.1 Thread Management

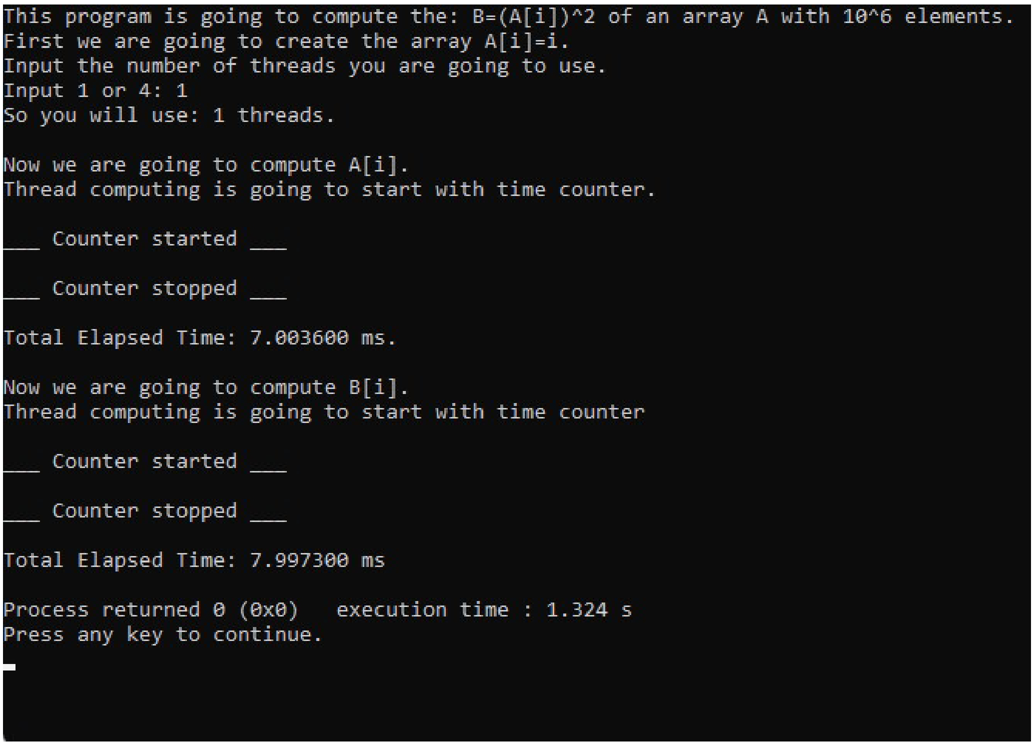
Threads are divided into four equal parts, with each thread starting computation from distinct points to prevent data overlap, enhancing the reliability of parallel processing.

1.1.2 Race Condition Management

We implemented synchronization mechanisms to manage dependencies and prevent race conditions, ensuring stable and reliable thread operations.

2 Execution Results

The application of multi-threading demonstrated significant performance improvements, quantitatively assessed through repeated timing measurements. The results highlight the benefits of using multiple threads in terms of execution speed and efficiency.



```
This program is going to compute the: B=(A[i])^2 of an array A with 10^6 elements.
First we are going to create the array A[i]=i.
Input the number of threads you are going to use.
Input 1 or 4: 1
So you will use: 1 threads.

Now we are going to compute A[i].
Thread computing is going to start with time counter.

___ Counter started ___

___ Counter stopped ___

Total Elapsed Time: 7.003600 ms.

Now we are going to compute B[i].
Thread computing is going to start with time counter

___ Counter started ___

___ Counter stopped ___

Total Elapsed Time: 7.997300 ms

Process returned 0 (0x0)   execution time : 1.324 s
Press any key to continue.
```

Figure 1: Parallelism observed with 1 thread vs 4 threads for 10^6 elements. The graph indicates that multi-threading can significantly reduce computational time, showcasing nearly a fourfold increase in speed.

```
This program is going to compute the: B=(A[i])^2 of an array A with 10^6 element
s.
First we are going to create the array A[i]=i.
Input the number of threads you are going to use.
Input 1 or 4: 4
So you will use: 4 threads.

Now we are going to compute A[i].
Thread computing is going to start with time counter.

___ Counter started ___
))))))))))))))))))))))))))))))))))))))))++++++++++++++++++++++++++++++++++++++++
****************************************,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

___ Counter stopped ___

Total Elapsed Time: 9.000300 ms.

Now we are going to compute B[i].
Thread computing is going to start with time counter

___ Counter started ___
))))))))))))))))))))))))))))))))))),,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,+++++
+++++++++++++++++++++++++++++++++++)))))****************************************

___ Counter stopped ___

Total Elapsed Time: 9.999000 ms

Process returned 0 (0x0)   execution time : 0.880 s
Press any key to continue.
```

Figure 2: Parallelism observed with 1 thread vs 4 threads for 10^7 elements. This figure reinforces the scalability of multi-threaded processing with even larger data sets, where the performance gains become more pronounced.

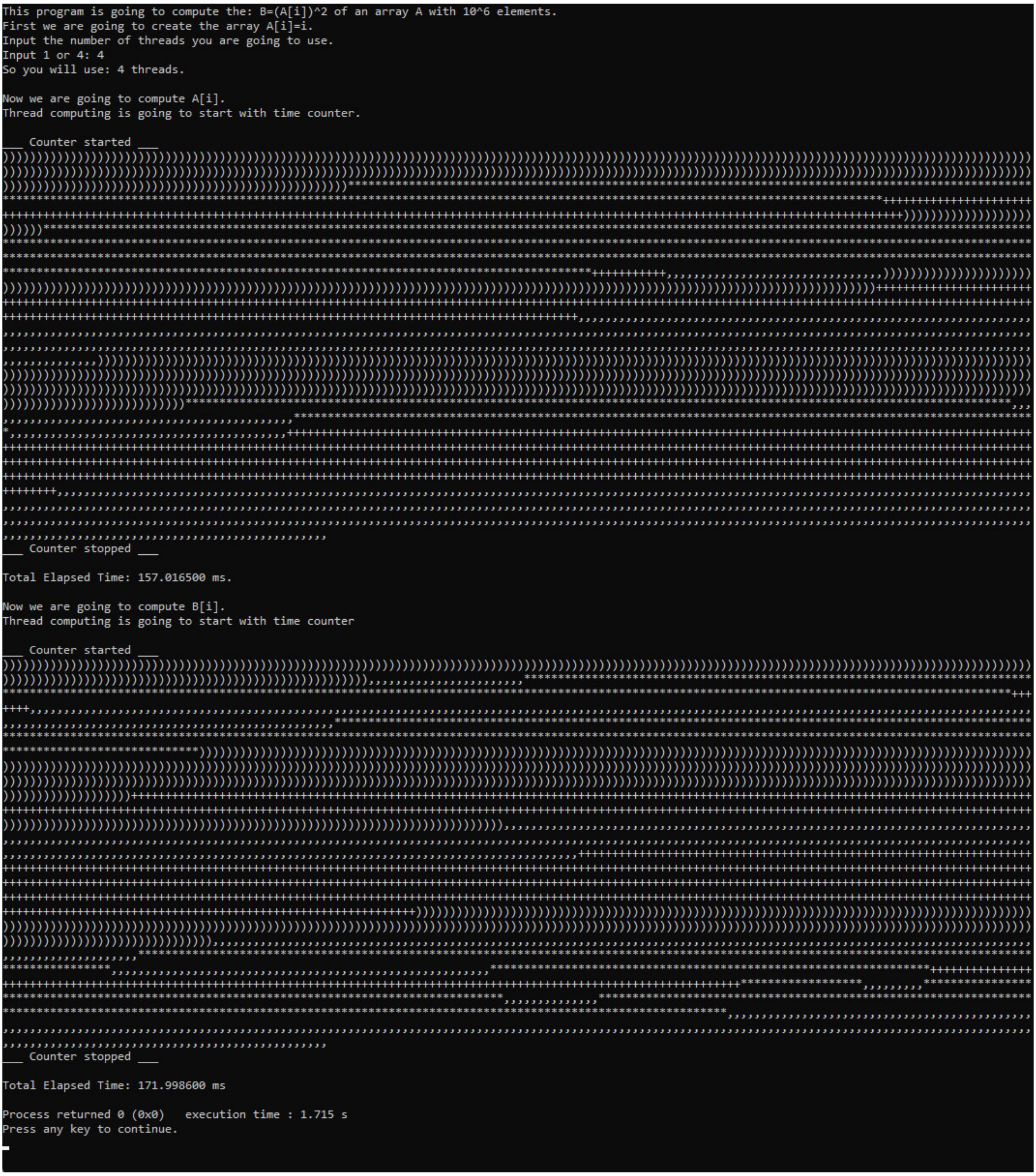


Figure 3: Improved parallelism visualization with larger dataset. The visualization aids in understanding the distribution of workload across threads and the effectiveness of concurrent execution.

Table 1: Execution Time Comparison

	Elements: 10^6		Elements: 10^7	
	A	B	A	B
1 thread	6,115,698	6,048,686	63,066,99	63,538,06
4 thread	2,159,298	2,026,7	17,954	17,476
1 thread/4 thread	2,832,262	2,984,499	3,512,699	3,635,733

3 Conclusion

The experiments conducted on a Ryzen R9 3900x processor clearly demonstrate the significant advantages of employing multi-threading in computational tasks. Using 4 threads instead of a single thread, we observed a nearly fourfold increase in processing speed, especially pronounced with larger datasets. This efficiency gain underscores the potential of parallel processing in enhancing performance and reducing execution times.

Moreover, the careful management of threads to prevent race conditions and the division of work among threads effectively minimized potential computational conflicts, further proving the robustness of a multi-threaded approach in practical applications. The results not only support the use of multi-threading in similar computational scenarios but also encourage further exploration into optimizing thread management techniques to harness even greater efficiencies in future projects.

A Appendix

A.1 Source Code

Below is the source code used in the project, which demonstrates the setup and management of POSIX threads for array processing.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
//Compute the B(i)=square(A(i)) in: a) single thread b) four threads.
//Check the difference in the execution time.
#define ARRAY_SIZE 1000000 //the size of your array that you are going to use
#define BILLION 1000000000.0 // billion is using for time counter
int number_of_threads; // global threads
double A[ARRAY_SIZE], B[ARRAY_SIZE]; // global arrays
void* creatingarray(void* arg) //function that creates array A
{
int index = *(int*)arg;
for (int k=index; k<index+ARRAY_SIZE/number_of_threads; k++)
{
A[k]=(k+1);
if (number_of_threads!=1 && k<index+40) //for printing the first 40 characters
printf("%c",41+(index*number_of_threads/ARRAY_SIZE));
//printf("%d\t",index);
//printf("%13.0f\t",A[k]);
//printf("%d\n",k+1);
}
}
void* squaring(void* arg) //function that computes array B[i]=(A[i])^2
{
int index = *(int*)arg;
for (int k=index; k<index+ARRAY_SIZE/number_of_threads; k++)
{
B[k]=(A[k])*(A[k]);
if (number_of_threads!=1 && k<index+40) //for printing the first 40 characters
printf("%c",41+(index*number_of_threads/ARRAY_SIZE));
//printf("%d\t",index);
//printf("%13.0f\t",B[k]);
//printf("%d\n",k+1);
}
}
int main(int argc, char* argv[])
{
printf("This program is going to compute the: B=(A[i])^2 of an array A with 10^6 elements.\n");
printf("First we are going to create the array A[i]=i.\n");
printf("Input the number of threads you are going to use.");
do
{
printf("\nInput 1 or 4: ");
scanf("%d",&number_of_threads);
} while(!((number_of_threads==1) || (number_of_threads==4)));
//Input number 1 for 1 thread or 4 for 4 threads
int i;
struct timespec start, start2, end,end2; // for time counter
printf("So you will use: %d threads.\n",number_of_threads);
printf("\nNow we are going to compute A[i].\n");
pthread_t th[number_of_threads]; //Creating an array of threads
```

```
printf("Thread computing is going to start with time counter.\n");
printf("\n--- Counter started ---\n");
clock_gettime(CLOCK_REALTIME, &start); //Counter here is starting
for (i=0; i<number_of_threads; i++)
{
int* a=malloc(sizeof(int)); // a helps to divide the work of computing into 4
    equal
computational pieces
*a = i*(ARRAY_SIZE/number_of_threads);
//checking if something is going wrong when pthread create
if(pthread_create(th+i, NULL, &creatingarray, a) !=0)
{
printf("\n%d",i);
perror("Failed to create thread!");
return i+1;
}//else printf("\nStarted Succesfully!\n");
}
for (i=0; i<number_of_threads; i++)
{
//checking if something is going wrong when pthread join
if(pthread_join(th[i], NULL) !=0)
{
perror("Failed to join thread!");
return i+1;
}//else printf("\nFinished Succesfully!\n");
}
clock_gettime(CLOCK_REALTIME, &end); //Counter here stops
printf("\n--- Counter stopped ---\n");
double time_spent = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)/
    BILLION;
printf("\nTotal Elapsed Time: %.6f ms.\n", time_spent*1000.0);
// next we are using the same logic for computing B[i]=(A[i])^2
// we could use 4 threads for all computing more faster
// with half code, but for educational use weI did it with 2 times of using
    threads
printf("\nNow we are going to compute B[i].\n");
pthread_t th2[number_of_threads];
printf("Thread computing is going to start with time counter\n");
printf("\n--- Counter started ---\n");
clock_gettime(CLOCK_REALTIME, &start2);
for (i=0; i<number_of_threads; i++)
{
int* a=malloc(sizeof(int));
*a = i*(ARRAY_SIZE/number_of_threads);
if(pthread_create(th2+i, NULL, &squaring, a) !=0)
{
perror("Failed to create thread!");
return i+1;
}//else printf("Started Succesfully!\n");
}
for (i=0; i<number_of_threads; i++)
{
if(pthread_join(th2[i], NULL) !=0)
{
perror("Failed to join thread!");
return i+4;
}//else printf("Finished Succesfully!\n");
}
clock_gettime(CLOCK_REALTIME, &end2);
printf("\n--- Counter stopped ---\n");
```

```
time_spent = (end2.tv_sec - start2.tv_sec) + (end2.tv_nsec - start2.tv_nsec)/
    BILLION;
printf("\nTotal Elapsed Time: %.6f ms\n", time_spent*1000.0);
return 0;
}
```

Listing 1: POSIX Threads Implementation