Dijlah Journal of Engineering Science (DJES) Vol. 2, No. 4, December, 2025, pp. 1-13
ISSN: Printed: **3078-9656, Online: 3078-9664**, paper ID: 74

Using PTHREAD to Calculate the geometric mean of M numbers

Ehab Abdulrazak. Alasadi
Department of Islamic Science, University of Kerbala, Kerbala, Iraq

Article Info

Article history:

Received Aug., 17, 2025
Revised Sept., 20, 2025
Accepted Oct. 29, 2025

Keywords:

Parallel program,
PTHREAD,Mutex
Shared memory.

ABSTRACT

Analyze the possibilities of implementing a parallel algorithm for calculating the geometric mean of M numbers. Design and implement (in C/C++) a solution based on memory sharing between individual "threads" of the program. Consider using the "**PThread**" library. Use the file to dump auxiliary information about the execution of individual threads, with at most one writing to the file at any time. Determine the speedup when executing on N processors if consider the PRAM model. Document the design as well as the implementation in detail. If the solution requires it, use files for the input and output of matrices (and vectors) where the row of the matrix corresponds to the row of the file and the values of the columns are separated by spaces or tabs. Unless otherwise stated, work with real numbers.

Corresponding Author:

Ehab Abdulrazak . AL-ASADI
Department of Islamic Scnience , University of Kerbala
Kerbala, Iraq
Email: ehab.a@uokerbala.edu.iq

1. INTRODUCTION

Thread is a way of executing within a process. A process can also have more than one thread. The standard library for the creation of threads in your program is in POSIX layer of the operation system . The following system call will allow you to create new threads in your process. The first argument in pthread_create() is a variable that holds the address that reference to the thread want to create, and the second argument is which typed to pthread_attr_t is a variable that let us address the attributes of the thread such as priority. The third argument need to supply to this system call is a function pointer that let the kernel knows where the thread should begin from and the fourth argument is the parameter would like to pass to the function that we are going call from this thread.[1]. The determinant is a function dependent on the dimensions of a matrix that assigns a scalar value to each square matrix. This value generally expresses the size of the matrix. In the following text, we will understand the determinant to mean this scalar value.Using the determinant of a matrix, its singularity can be determined. The determinant is also used to solve systems of linear equations using Kramer's rule, determine the eigenvalues of a matrix, the orientation of the coordinate system, and solve other problems.Determining the determinant of a matrix, especially for matrices of larger dimensions, is a computationally demanding task for which it makes sense to design a parallel algorithm.

Program Parameters
The research consists of five files:

thread – compiled executable file **thread.c**

– program source code **list.txt** - output file

Compile the program:

To compile, enter:

```
gcc thread.c -o thread -l pthread –lm or

g++ thread.c -o thread -l pthread
```

Using the program:

The program can be started as follows

```
./thread [-h] this help

./thread <filename> <number of threads>
```

The total product and auxiliary thread listings are displayed on the screen

Program datasets

After starting, the program opens the input file and reads the input data from it, which are separated by spaces. according to the definition of the geometric mean, these should be real positive non-zero numbers. It determines the length of the array and writes the elements to the string array of dynamically allocated memory. After loading all the elements, the main thread of the program (manager) creates the necessary number of secondary (**worker**) threads and starts the calculation. Once created, the threads will get an allocated range in the shared memory of the process and will perform the multiplication process on them and store the intermediate result. In order to get the final result of the problem, it is necessary to perform the necessary operations on all data parts, which gives us all the intermediate results.[8],[9]

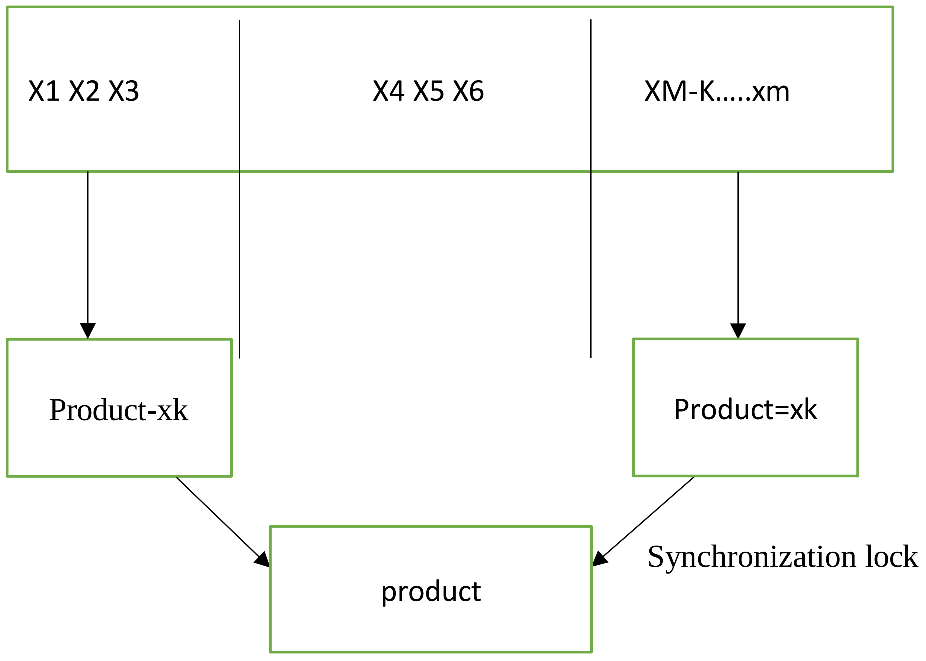

Fig (1) Writing to a shared memory location

We can choose two procedures for saving intermediate results.the first is to use the allocated memory field, in which individual intermediate results will be written. After all data blocks are executed, the intermediate results are multiplied. With this approach, there can be no contention between threads when reading and writing, so there is also no mutual slowdown. let M be the number of elements and N the number of threads. The main thread of the process waits until all threads have completed their tasks and then multiplies the intermediate results,for synchronization it will be necessary to use one of the functions of the pthread library, for example connecting threads with the manager thread after the activity has ended.[5].

Manager Thread

```
vector - a global field of size M
tmp - global array for intermediate results of
size N from , to - the scope of processing for the
thread
N = number of
threads product
= total product

CreateThreads(threadCount) // creates a defined number of threads
joinThreads()                // wait until the
threads end product = tmp[0];
for (int i=0;i<N;i++) sucin *=
tmp[i] root(product)
                                  /
/ restult
```

Worker Thread

```
for (int i=From;i<To;i++)
{
 Tmp[N]
*=vector[i];
Lock
allocation
Write the result to output file
unlocking the lock}
```

2. METHOD

The geometric mean is a useful measure of position for sets that have a log-normal distribution (that is, the logarithms of the values of a variable have a norm distribution). It is calculated only from non-negative values of the variable as the nth square root of the product of all values: **See Fig 1**

$$\bar{x}_G = \sqrt[n]{\prod_{i=1}^{n} x_i} \qquad \text{.................................. [1]}$$

a. PRAM Model

It is a computer model that consists of M synchronous processors with shared memory. An assignment should be designed based on this model. In this model, the problem of synchronization and communication is neglected and it mainly focuses on the task of parallelizing the problem. There are different variations of the model depending on how processors access shared memory at a given time.

i. **EREW** (exclusive read, exclusive write) – no two processors are allowed to simultaneously read and write to the same memory location.
ii. **CREW** (concurrent read, exclusive write) - allows simultaneous access to the memory location for reading, but only one processor can write.
iii. **CRCW** (concurrent read, concurrent write) - allows simultaneous access to the memory location for reading and writing. This algorithm is further divided into Priorities - the complete write is assigned to the processor with a higher priority Arbitrary - complete write is allowed to a randomly selected processor

Common – writing occurs only if both processors want to write the same value **[6],[7]**

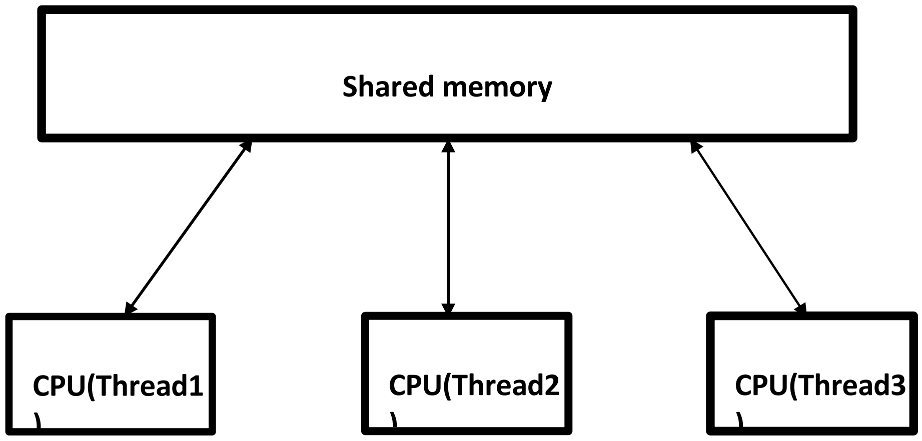

Fig (2) PRAM Model

The sought values of the solution according to this model will be acceleration and efficiency. Let us define TS(n) as the time consumption of the best-known sequential algorithm for a given problem with an input of size n. Let us define TP(n) as the time of a parallel algorithm with an input of size n that is executed on m processors.

Acceleration = S □ TS (n) [6]

TP (n)

Efficiency of the parallel algorithm= EP

EP – is in the range <0.1>

$T1$ (n), $m.T$ (n[6]

T1 – time requirement of a parallel algorithm executed on one processor

In our case, will use the (Manager – Worker) model. It means that the main thread (Manager) will be created while the program is running. This thread creates other threads as needed, the so-called worker threads (Worker). These threads are created to execute a sequence of instructions - in practice it is some part of the program. The Manager, i.e. the main thread, serves for the entire organization of the algorithm and has the task of collecting and evaluating data from working threads.

The program read the input file and reads the input data from it, which are separated by spaces. According to the definition of the geometric mean, these should be real positive non-zero numbers. It determines the length of the array and writes the elements to the string array of dynamically allocated memory. After loading all the elements, the main thread of the program (manager) creates the necessary number of secondary (worker) threads and starts the calculation. Once created, the threads will get an allocated range in the shared memory of the process and will perform the multiplication process on them and store the intermediate result. In order to get the final result of the problem, it is necessary to perform the necessary operations on all data parts, which gives us all the intermediate results. The algorithm calculation is indicated in the following section. Considering a more efficient approach that does not block individual threads by waiting for the write option, I chose the first option.

Manager Thread

```
vector - a global field of size M
tmp - global array for intermediate results of
size N from , to - the scope of processing for the
thread
N = number of
threads product
= total product
CreateThreads(threadCount) // creates a defined number of threads
joinThreads()                // wait until the
threads end product = tmp[0];
```

```
for (int i=0;i<N;i++) sucin *=
tmp[i] root(product)// restult
```

Worker Thread

```
for (int i=From;i<To;i++)
{
 Tmp[N]
*=vector[i];
Lock
allocation
Write the result to output file
unlocking the lock}
```

Time measurement

The measurement of the time spent on the calculation can be realized with the functions gettimeofday, getrusage, or clock_t(). In the case of the function gettimeofday and clock_t(), it will be the total time that was needed to calculate the given problem, with the difference that the function returns the number of ticks of the processor since the system was started, which must be converted to seconds, for example, by dividing by the constant CLOCKS_PER_SEC. The getrusage function provides information about the time that the task spent calculating in the processor, the time when the process was not executed at all (eg when it was waiting). see (Table1).

When modifying a matrix from matrix A to matrix B, there are several rules regarding its determinant:

If B was created from A

• by swapping two rows or columns, then det(B) = -det(A)

• by multiplying one row or column by a constant c, then det(B) = c.det(A)

• by adding the product of one row to another row, or by adding one column to another column, then det(B) = det(A)

The determinant calculated from the modified matrix must therefore be subjected to the appropriate operations, depending on the rules used in the modification.

Calculating the determinant from the modified matrix

The determinant can be calculated directly from the matrix in several ways:

Calculating the determinant by definition

Let the matrix A = (Ai,j), (where Ai,j is the matrix element in the i-th column and j-th row) be square. Then

• for matrix dimension A = 1, the determinant is det(A) = A1,1

• for matrix dimension A = 2, the determinant is det(A) = A1,1 A1,2 – A2,1 A1,2

• for matrices of dimension n, the Leibnitz formula can be used

det(A) = sum(sign(p).prod(Ai,p))

where

• sum is the function of the sum of the terms over all their permutations

• sign is a function whose output is 1 if an even permutation p follows and -1 if an odd permutation p follows (which was created by an even or odd number of exchanges of elements in it)

• prod is the function of the product of terms for i = 1 to n

Example of the method of calculating the determinant according to Leibnitz's formula for n = 3:

det(A) = A1,1 A2,2 A3,3 + A1,3 A2,1 A3,2 + A1,2 A2,3 A3,1 +

+ A1,3 A2,2 A3,1 + A1,1 A2,3 A3,2 + A1,2 A2,1 A3,3

This results in n! sum terms, so the method is not suitable for large matrices.

Laplace Decomposition

Using the Laplace decomposition, the determinant of a matrix can be calculated by decomposition along any row or column. The calculation by decomposition of row i of a matrix A of dimension n can be written as follows:
det(A) = sum(Ai,j (-1)i+j Mi,j)
where
• sum is the function of the sum over all j = 1 to n
• Mi,j is the determinant of the submatrix of matrix A, which was created by removing row i and column j from matrix A.

Additional rules for calculating determinants

• The definition of the determinant implies that the determinant of a triangular matrix is equal to the product of the members along the main diagonal, i.e. A1,1 to An,n for a matrix of dimension n
• For square matrices A and B,
det(AB) = det(A)det(B).
• The determinant for a matrix with a row or column of zeros is equal to 0.
• The determinant of a matrix with 2 identical rows or columns is equal to 0.

Similar methods for solving by modifying a matrix

Gaussian elimination method

Gaussian elimination method is an algorithm whose input is an arbitrary matrix and the output is a matrix containing rows defined as follows:
The first non-zero term from the left is equal to 1, the other terms are zero. This result is achieved by the operations of exchanging rows, adding a row and the multiple of another row, and dividing rows. The algorithm has a complexity of O(n3) and is numerically quite unstable when working with real numbers. To increase stability, so-called pivoting is used, which is the rearrangement of rows so that the largest term of the column is always on the main diagonal (this is partial pivoting). After completing the Gaussian elimination algorithm, it is easy to calculate the determinant from the matrix thus obtained as the product of the terms on the main diagonal.

3. RESULTS AND DISCUSSION

Description of the algorithm

The program was implemented using the C++ language in the LINUX environment and the POSIX thread library (pthread) and its following functions:

pthread_create – function to create a new thread pthread_exit – function to terminate a thread

pthread_join – the function allows you to wait for the given thread to complete pthread_mutex_lock - a function to lock a lock.

pthread_mutex_unlock – function to unlock locks pthread_mutex_init – lock initialization pthread_mutex_destroy – release lock resources.

To simulate parallel processing by multiple processors according to the shared memory PRAM model, I used functions to create, terminate and join threads. I us mutexes to maintain access shared memory (file-only in a particular implementation). These basic elements give the program the following execution structure:

Input and output requirements

create the required number of working threads (if the operating system allows it). Subsequently, the main thread calls join on all created threads and thus waits until the end of the last thread. As mentioned in the analysis, I used an array to store partial results, thus eliminating the need for synchronization for each multiplication operation, which increased the execution speed of the program at the expense of allocating more memory.

long double *tmp; long double *vector;

So I used two tmp arrays - for partial results for each worker thread and a vector - to load the processed data. For most cycles, the variables number_of_fibers and number_of_elements of the vector play an important role, kt. must be greater than the first one. I solved the problem of data distribution between individual threads with this algorithm, which first calculates the size of the part that will be processed by one thread and then recalculates the boundary values for each thread, allocating any remaining to the last thread:

```
long offset;
offset = number_of_elements / number_of_threads; for(i=0;i<number_of_threads-1;i++){ arg_field[i].id = i;
arg_field[i].from =i*offset; arg_field[i].to = i*offset+offset;
}
if (number_of_elements>i*offset){ arg_array[number_of_threads-1].id = i; arg_field[number_of_threads-1].from
=i*offset;
```

a.Program description

Program files:

thread – compiled executable file thread.c –

program source code output.txt - output file

Compile the program:

To compile, enter:

```
gcc thread.c -o thread -l pthread –lm or
g++ thread.c -o thread -l pthread
```

Using the program:

The program can be started as follows

```
./thread [-h] this help
```

```
./thread <filename> <number of threads>
```

The total product and thread listings are displayed on the screen.

b.PRAM Description

I summarized the basic theory of the PRAM model briefly in the analysis, and now I will try to outline how to determine the speedup when executing on N processors if you consider the PRAM model. To determine the acceleration, it is also necessary to model the sequential algorithm as it follows from its calculation, presented later. The calculation time of a problem with complexity n is the number of time units (steps) required to complete the calculation T(n). The algorithm for a given size of the task needs P(n) processors. The cost of the calculation is defined as C(n) = T(n).P(n). Work W(n) is defined as the total number of operations on all processors required to perform the task. At the same time, Brent's theorem applies, that if the algorithm has a time complexity of T(n) steps and requires the work of W(n) operations, it can be simulated on PRAM with p processors in time T(n,p)<=|W(n)/p|+ T(n).[7]

The elements are stored in the vector field, the result will be in the gp variable, n is the number of elements:

Sequential algorithm:

```
for i:= 1 to n do gp *= vector[i]; gp =
pow(gp, 1/n);
```

If consider square rooting and multiplication as one step, the following applies:

number of processors: P(n, 1) = 1 number of

steps: T(n, 1) = n + 1 = O(n) work: W(n, 1) =

O(n)

price: C(n, 1) = P(n, 1).T(n, 1) = O(1).O(n) = O(n)

Assume that n =pk Parallel

algorithm:

```
for h:= 1 to k do
for i:= 1 to n/ph pardo tmp[p] *= vector[i*p+p]; gp=tmp[0];
for i:= 1 to p to gp *= tmp[i]; gp =
pow(gp, 1/n);
```

number of processors: P(n, p) = p = n

number of steps: T(n, p) = O(logp n) work:

W(n, n) = O(n)

price: C(n, n) = P(n,p).T(n,p) = O(n).O(logp n) = O(n.logpn) so, calculate the acceleration:

S(n,n) = Ts(n,1)/T(n,p) = O(n) / O(logpn) =(if p=n) O(n)

Funding This research received no external funding and no funding was involved .

Evaluation of the results, the program had to be tested. Therefore, the network of 1 thread was created. This was used during the calculation and therefore the results are distorted. especially when using multiple threads that generate multiple messages. in these cases, it is rather a slowdown, since the calculation time on one node is very low due to the mentioned circumstances. See fig (3) fig (4).

Table 1: Speedup at N process measurements

No of Threads	**1**	**5**	**10**	**20**	**50**
1	Real time1	Real time2	Real time3	Real time4	Real time5
2	4.1644	2.1377	8.2770	4.9990	3.720
3	2.40442	3.540	6.2930	2.2410	8.9730
4	8.7097	4.8350	2.6290	8.6250	1.7490
5	4.0749	6.6760	2.50	4.8960	1.9590
Avg	4.8376	4.2970	4.9240	5.4400	4.1000
Speed up= T1/Tn		1.1250	0.8720	0.9050	1.3260

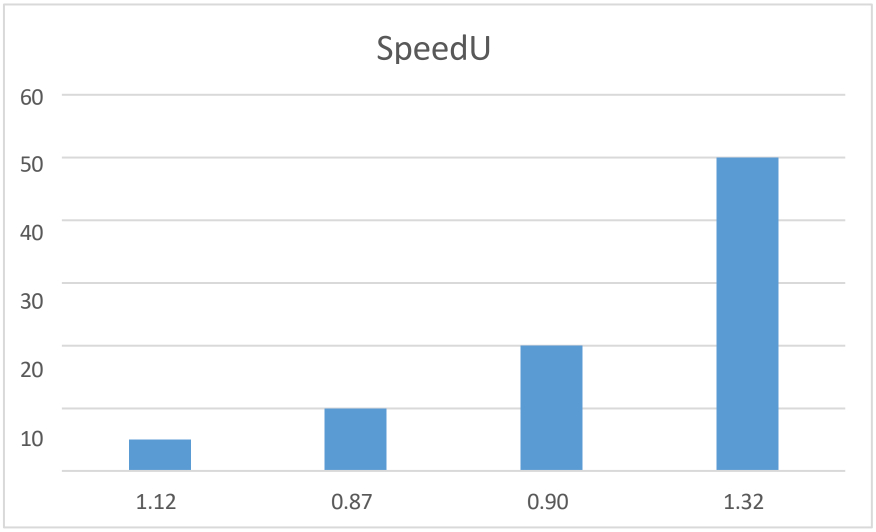

Fig (3) Speedup vs No. of Threads

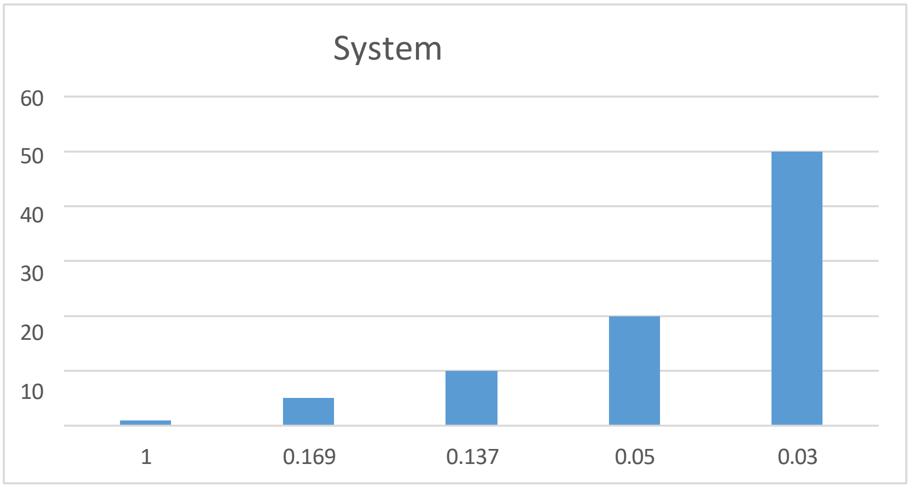

Fig (4) System efficiency

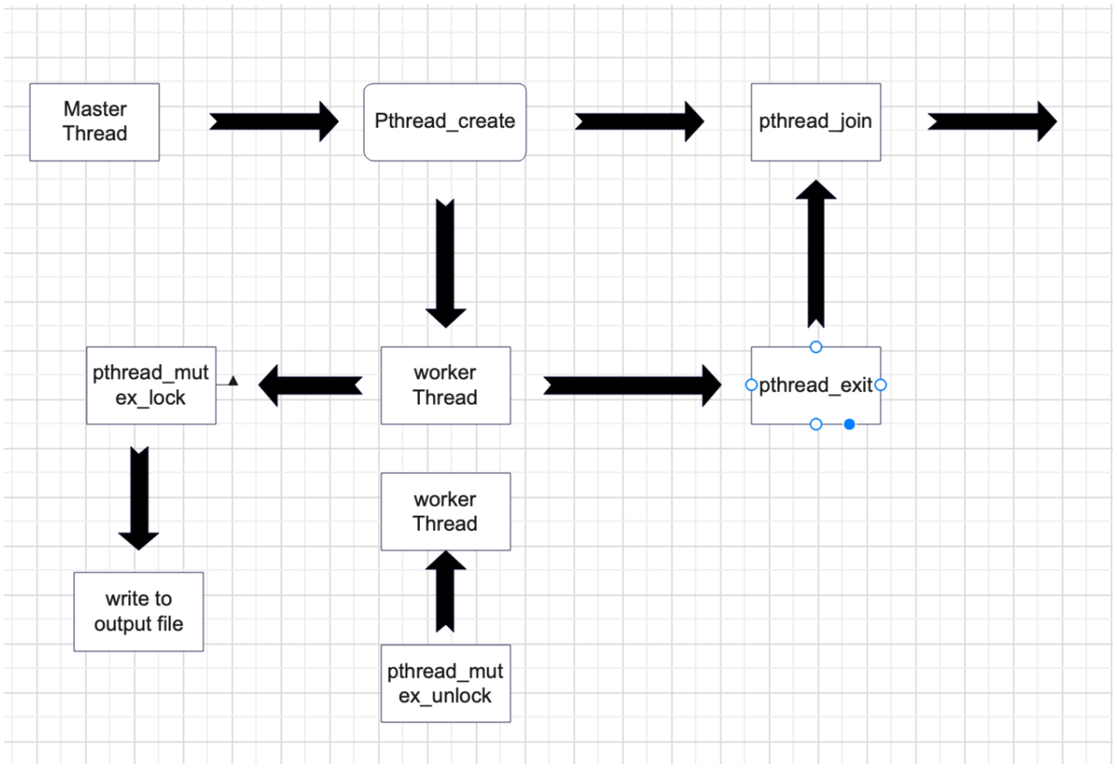

Fig (5) Program running structure

```
arg_array[number_of_threads-1].to = number_of_elements;
}
```

In order to sell information to the thread about which part of the field to process and where to write the partial results, I used an argument when creating the thread in which send the following structure:

```
struct argument
{
long id; long from; long to;
};
```

Other parts of the implementation are in correspondence with the proposal and can be consulted in the program

4. CONCLUSION

Testing took place on the Linux operating system. With an input vector with the number of coordinates 9, 128, 4096 and 1000000 elements. The calculation itself took a relatively short time since the specified problem is not computationally demanding. The program is rather numerically demanding. Even with the number of 4096 elements, the double data type was not able to process the resulting product and even the intermediate results of the products of individual threads, so I changed the data type to long double, which had no problems with the resulting product. However, the pow() function works with variables of type double, so the result loses precision and the geometric mean cannot be calculated for a product that is not of type double. This could be solved using equivalence.

REFERENCES

[1] YoLinux: YoLinux Tutorial“POSIX thread (pthread) libraries“, http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html,accessed 15.Dec.2024

[2] Eric W. Weisstein“Geometric Mean“ http://mathworld.wolfram.com/GeometricMean.html, accessed 15.Dec.2024 .

[3] Dick. B,et al,” PThreads Programming: A POSIX Standard for Better Multiprocessing , O'Reilly Media,1996.

[4] David.B,“ Programming with POSIX Threads, Addison-Wesley Professional,1997.

[5] W. S, Stephen .R,” Advanced Programming in the UNIX Environment, Addison-

Wesley Professional,2013.

[6] Hesham El-Rewini , “Advanced Computer Architecture and Parallel Processing”,

Wiley-Interscience,2008.

[7] Alasadi, E. (2024).” Parallel algorithm for testing the singularity of an N-th order matrix”, Al-Bahir Journal for Engineering and Pure Sciences, 4(2), 1.

[8] RAl-Asadi, E. A. (2015). “Finding N prime numbers using distrusted computing PVM (parallel virtual machine)”. Int. J. Eng. Technol, 5(11), 578-588.

BIOGRAPHIES OF AUTHORS

The recommended number of authors is at least 2. One of them as a corresponding author.
Please attach clear photo (3x4 cm) and vita. Example of biographies of authors:

Author 1 picture	**Lecturer. Ehab A. Alasadi**, Received His MSc. degree in the computer engineering from University of Technology In Slovakia –in 2009 and Doctor of. He has been a full-time lecturer in Islamic Science collage -, Kerbala, Iraq, since March 2010. He, ongoing He can be contacted at email: ehab.a@uokerbala.edu.iq
2 Author 2 picture	Mini cv
Author 3picture	Mini cv
Author 4 picture	**Mini cv**
Author 5picture	**Mini cv**