Assignment 1 - CS401V - Distributed Systems

Matrix Multiplication: An example of parallel computing

Conventional algorithm for multiplication of two matrices A(*m* x *n*) and B(*n* x *t*) has a complexity of O(*m* x *n* x *t*). Given that the two matrices are square matrix of size *m*, there exist the solutions that allow to marginally reduce the complexity of the algorithm. For instance, Strassen Algorithm reduces the complexity of the matrix multiplication algorithm to O((7/8) x m^3). Contradictory, by using parallel computing, without reducing the algorithm complexity, we still can reduce the execution time of the algorithm by parallelizing and distributing the computation of the algorithm. Unfortunately, we do not have a multi-core computer that allows us to realize this approach easily. Nevertheless, we at least can manually distribute the computation of the algorithm. The objective of the project is to implement such an algorithm in C/C++ language, by using process creation instruction **fork()**, the primitive synchronizations and a method of memory-mapped file I/O **mmap()**.

The project should be submitted before 23:59, 18/10/2025. A zip file named by yourname-givenname.zip should be sent to tram.truong-huu@ttu.edu.vn. It composes of the source code of the project well commented, a report justifying your choice, describing the used algorithm and analizing the obtained results.

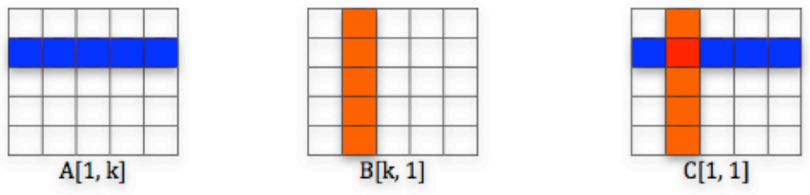
1 Matrix Multiplication Algorithm

Given two matrices A(*m* x *n*) and B(*n* x *t*) for multiplication, we describe in the following three approaches to obtain the resulting matrix C(*m* x *t*).

1.1 Sequential computation

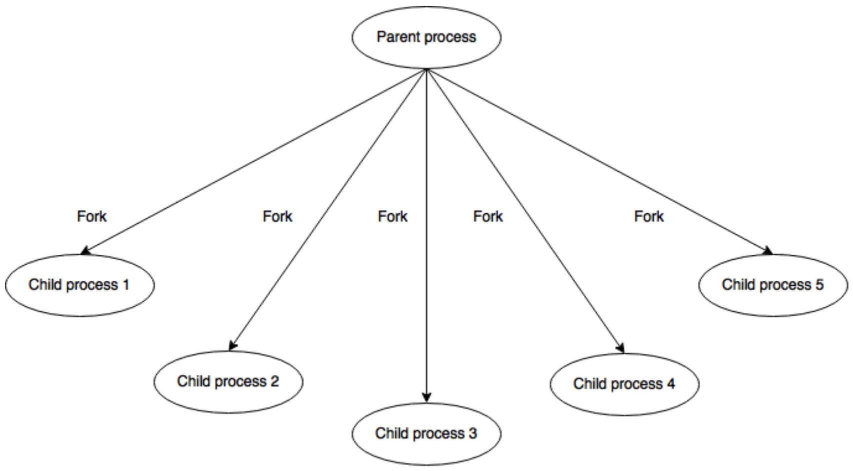
This is a naive solution in which a sole process sequentially computes the value of each element of matrix C. The pseudo code is as follows:

```
for (i=0; i<m; i++)
   for (j=0; j<t; j++) {
      C[i][j]=0;
      for (k=0; k<n; k++)
         C[i][j]+= A[i][k]*B[k][j];
   }
```



1.2 Parallel computation at row level

The conventional algorithm (i.e., the sequential algorithm) becomes time-consuming if the size of the input matrices is large. In order to accelerate the computation, we can parallelize the algorithm and run it by using the computational capacity of multiple computers in a distributed system. This is the principle of grid computing or cloud computing. In the framework of this project, assuming that each thread is considered as a computer in the distributed system, we will realize the computation of matrix multiplication by multiple child processes initialized by the same parent process (see in the following figure).



Let *p* be the number of processes initialized by the parent process, each process *q* (*q*=1..*p*) will compute the values of one row of the resulting matrix C. The pseudo code for the computation of each row is as follows:

```
compute_row (i): // compute the value of each element of row  i of matrix C
    for (j=0; j<t ;j++){
         C[i][j]=0;
         for (k=0;k<n;k++)
             C[i][j]+=A[i][k]*B[k][j];
    }
```

If matrix C has an important number of rows, the number of child processes *p* is usually smaller than *m* (*p*<<*m*). In this case, the *p* processes successively compute multiple rows of matrix C until all the rows are computed. In order to decide which row a child process has to compute, the following strategy can be used: a common variable *l* (initially assigned to 0), keeps the index of the next row will be computed. Each child process increases the value of *l* by 1 after receiving the index of the row to be computed. Consequently, the access to the common variable *l* has to be synchronized.

The data structure of the program is as follows:

- A common variable allowing each child process to determine the next row to be computed (concurrent access to be protected)
- Two input matrice A and B that have to be shared among child processes in read-only mode (for example by memory segmentation function **mmap()**)
- The resulting matrix C that is shared among the child processes in write-mode (do we need to share it mutually?)

1.3 Parallel computation at element level

We discuss in this section the third approach for parallelizing the matrix multiplication algorithm. This approach has a grainularity much finer than that presented in the previous section.

Similarly, this approach also uses *p* child processes created by the same parent process. These child processes successively compute multiple elements of matrix C until all elements are computed. Consequently, two common variables *(l,c)* are used to keep track of the indice of the next element to be computed. The pseudo code to be run by each child process is presented in the following:

```
compute_element (l, c):
    C[l][c]=0;
    for (k=0;k<n;k++)
       C[l][c]+=A[l][k]*B[k][c];
```

2 Implementation procedure

To simplify the number of parameters, we will consider here and after only the square matrice of size *m*.

2.1 Step 1

Before implementing the algorithms presented above, you will first implement a function named **populate** (int size, double* A, double* B) that will generate the two input matrice randomly. The function takes three input parameters as the size of the matrice and the memory address of the two matrice. The function uses **rand()** to randomly generate a value.

```
for (i=0; i<m; i++)
   for (j=0; j<m; j++) {
     A[i][j] = rand();
     B[i][j] = rand();
   }
```

2.2 Step 2

Implement the sequential algorithm in a file named **sequentialMult.c**. The program has to:

- take an input parameter as the size of the input matrice
- utilize the function **populate(...)** implemented in Step 1 to randomly generate the values of matrice A and B
- measure the execution time in µsecond to perform the multiplication. To do so, you can the command **gettimeofday()** in providing a variable of type **struct timeval**.

  ```
  struct timeval myts;
  ...
  gettimeofday(&myts, NULL);
  ```

- print out matrix C after finishing the execution by using the function **void printm(int size, double* C)** that you also have to implement.

In your report, provide the execution time of the algorithm for the matrix of size: 10, 100, 1000, 10000, 100000 et 1000000[1].

2.3 Step 3

Implement the second algorithm (parallel computation at row level) in a file named **parallelRowMult.c**. It also has to compose of the same functions as implemented in **sequentialMult.c**. In addition, it has to:

- take into account a parameter *p* as the number of child processes to be created.
- guarantee the synchronization when accessing to the common variable *l*.
- ensure that all the child processes terminate when matrix is entirely computed. The parent process waits until the completion of all the child processes, print out matrix C and terminate itself.

In your report, provide the execution time of the algorithm for the matrice of size: 10, 100, 1000, 10000, 100000 and 1000000, and for a number of child processes: 10, 100 et 1000. You are encouraged to write a shell script to autmate the execution invocation of these different computations.

2.4 Step 4

Similarly, implement the third approach in a file namely **parallelElementMult.c**. This program takes into account a parameter *p* as the number of child processes to be created and realizes the same computations as shown in Step 3.

2.5 Step 5

In your report, analyze the results obtained in the form of graph (If the execution time is too large, you may you logarithmic scale for y-axis). Compare the execution time of all three approaches, explain the behavior of the curves and provide your comments on the graphs.

References

The primitives of synchronization are described in the following website: http://www.cm.cf.ac.uk/Dave/C/

---

[1] It is possible that several runs might not be possible. Return to the top