Assignment 1

Nuno Neto
up201703898

December 9, 2020

Parallel Computing
Masters in Computer Science

1 Algorithm Implementation

The algorithm implemented was meant to solve the All pairs Shortest path problem which consists of finding the shortest path from each vertex to every other vertex in a directed graph. To solve this problem we first represent the graph as the matrix size N ($N \rightarrow$ Number of vertices in the graph) where each position $(row, column)$ of the matrix represents the cost of going directly from the vertex row to the vertex $column$. If there is no direct path between these two nodes we represent that position as ∞. The solution to the problem is a matrix with size N where each position $(row, column)$ corresponds to the cost of the shortest path between the vertex row and $column$.

1.1 Distance Product Matrix Multiplication

This algorithm is an adaptation of the normal matrix multiplication algorithm, where we replace the multiplications by sums and the sum by the minimum of the sums.
The algorithm is given the matrices D_f and D_k as arguments (Where f and k

are the depth of paths that can be formed to give the distance between two vertices) we get a resulting matrix D_{f+k} where the paths formed have the added depth of the two input matrices. When we refer to a path, we mean a path to get from a vertex to another either directly or by intermediary nodes. The depth of a given path is given by how many intermediary nodes you have to travel to get from the origin to the destination.
To solve the All Pairs Shortest Path problem we need to get a matrix that takes into account paths with depth N, D_N. We also know that if we have a matrix that takes paths of size larger than N we still maintain that it will contain the shortest path, as there were no new paths introduced that hadn't already been considered and because we always take the *min* of the sums we always keep the shortest path.
To obtain this matrix D_N we could start with the matrix D_k where k starts at 1 and then we multiply it repeatedly by D_1 util we get the matrix D_N. This method of calculating the result yields a complexity of $O(N^4)$ because we repeat the Distance Product Matrix Multiplication N times and matrix multiplication has a complexity of $O(N^3)$.

1.2 Repeated Squaring

To reduce the complexity of this algorithm we exploit a mathematical property of the Distance Product matrix multiplication. We know that given two matrices D_f and D_k when we perform the algorithm the resulting matrix will be D_{f+k}. So instead of multiplying one matrix N times by D_1 we multiply the matrices by them selves until we get a matrix that has a factor larger than N.
That is, we start with the weight matrix D_1 and we multiply it by itself to get D_2 so, generically, we multiply $D_k * D_k$ to get D_{2k} and then repeat this procedure until $2k >= N$. This yields a complexity of $O(\sqrt{N}) * O(N^3)$ that is $O(N^{3+\frac{1}{2}})$ which is significantly less than the original $O(N^4)$ from the previous method.

1.3 Fox's Algorithm

Fox's algorithm implements the Matrix Multiplication in a parallel manner, distributing the workload across multiple processors (To solve the All Pairs Shortest path problem we still need to apply the Repeated Squaring algorithm). To do this the algorithm requires a number of processes p that is a perfect square and whose square root divides N. So $p = q^2$ and $nbar = N/q$. Then we divide the factor matrices among the processes in a *block checkerboard* fashion. So we imagine our processes as virtual 2D array with size $q*q$ and each of the processes is given a factor of the matrix sized $nbar * nbar$ and the processes are assigned by $\phi(p) = (p/3, p \ mod \ 3)$. A visual representation can be found in Figure 1.

We will then do q steps and in each step we do the following (In our case, the input matrices A and B are the same):

- Choose a submatrix from the input matrix A each row in the process matrix. The matrix chosen in a given row is given by $u = (r + step)\ mod\ q$, so we get the process $A_{r,u}$ for each row r.
- In each row broadcast the chosen processes' matrix to the other processes in the same row.
- On each process, multiply the received submatrix of A by the matrix B stored in the process.
- On each process, send the send stored submatrix of B to the process directly above in a circular fashion, that is, the process in the first row sends the submatrix to the last row).

1.4 Data Structures Used

To represent the matrices, we used a 1 dimensional array mapped to 2 dimensions so we can have all of the needed data in a contiguous piece of memory.

As utility methods to create, access, destroy, among others we have the following:

- int matrix_alloc(unsigned int size, unsigned int **matrix); - Allocate a new matrix with size $size * size$ into $*matrix$.
- int matrix_copy(unsigned int size, unsigned int *dest, unsigned int *source); - Copy a matrix with size $size * size$ from *dest* into *source*.
- int matrix_fill (unsigned int size, unsigned int *matrix, unsigned int value); - Fills the matrix *matrix* of size $size * size$ with the value *value*.
- int matrix_free(unsigned int **matrix); - Frees the matrix in $*matrix$.

1.5 Utility methods

The utility methods were defined in the files *matrix_multiplication.h* and *foxsalgorithm.h*. The methods used are as follows:

- *int prepareMatrixForAllPairs(unsigned int matrixSize, unsigned int *matrix);* - Prepares the input matrix to be used by the Distance Product Matrix Multiplication. One the requisites for this algorithm is that when we have a path between two nodes i, j such that $i \neq j$ with weight 0, this weight must be set to ∞. This is because if there is no path between i and j the weight must be set to ∞ and not 0.
- *int buildScatterMatrix(unsigned int matrixSize, unsigned int Q, unsigned int processes, unsigned int *originalMatrix, unsigned int *destinationMatrix);* - Builds the matrix with the matrices that are going to be assigned to each process in their correct order, to then be used by MPI's Scatter.

- *int assembleMatrix(unsigned int matrixSize, unsigned int Q, unsigned int processes, unsigned int *originalMatrix, unsigned int *destination);* - Takes the original matrix that contains all the sub matrices returned by the processes, obtained by MPI's Gather, in their respective order and reorders them in the correct order to be displayed to the user.

2 MPI

To be able to have these processes perform the calculation in parallel we must have communication between them. To implement this communication we used MPI (*Message Passing Interface*) which is a framework that allows easy communication between processes.
Using MPI we can expand our program into not only using various processes in the same machine but also communicating with processes that are in other machines, allowing us to scale the program with great ease as we have to perform no code changes to our programs for it to be able to scale from a single computer to the largest supercomputer on the planet.

2.1 Communicators

MPI groups processes into what are called Communicators. The default global communicator is MPI_COMM_WORLD, that contains all of the processes that are running under MPI. As we saw from the algorithm description found in Section 1.3 the communication between processes very rarely includes all the processes so it would be very inneficient to send all the information to all the processes every time. To avoid this we create custom communicators in each process for it's row and another one for it's column which are the communications used the most during the execution of the Fox algorithm.

2.2 Data Types

We setup a custom datatype that is a vector of size $\frac{N}{Q} * \frac{N}{Q}$ using *MPI_Type_vector* to create the type and then we let MPI know about this new datatype with *MPI_Type_commit*.

2.3 Data Structures

We created a data structure, *struct MpiInfo* that contains all the relevant information needed for MPI communications. This data structure stores the process count, communicators for it's row, column and a global communicator. It also stores the processes position in the grid (It's row and column) and corresponding rank and also stores the data type defined in Section 2.2.

2.4 Used communications

In this assignment we used:

- MPI_Bcast to send information from one process to all others in a communication group.
- MPI_Scatter to send each process it's sub matrix
- MPI_Gather to receive the result matrix from each sub process
- MPI_Sendrecv_replace to send the B matrix to the process below and receive the B matrix from the process above it.

Amongst other methods that were used to setup all the data types and gather all the necessary information for the program to run.

3 Performance Analysis

Table 1: Performance analysis table. Measured using Linux's time command.

	Execution Style								
	Sequencial	Parallel							
		p = 1		p=4		p=9		p=16	
Matrix Size	Time	Time	Speedup vs Sequencial	Time	Speedup vs Sequencial	Time	Speedup vs Sequencial	Time	Speedup vs Sequencial
N = 6	0.025s	0.598s	-2292%	0.637s	-2448%	1.7s	-6700%	-	-
N=300	2.9s	3.44s	-18%	1.32s	54.5%	3.1s	-7%	2.3s	20%
N=600	28.1s	28.14s	0%	7.89s	72%	8s	72%	5.17s	82%
N=900	1m47.8s	1m50s	-3%	26.7s	75%	22.18s	79%	10.8s	90%
N=1200	5m37.7s	5m44.2s	-2%	1m14s	80%	52.3s	85%	24.8s	93%

As we can see from the data in Table 1, the sequencial speed get's exponentially bigger as we would have expected.
Compared to the sequencial running times, running the Fox algorithm with only 1 process yields generally worse performance, which is also to be expected since Fox will still try to share messages, even when there's only one process. As we can see in the case with the input size 6, we have a tremendous increase in running time due to MPI's overhead. Still in the input size 6 case, we can see that adding more processes degrades the performance even more. This is because adding more processes adds greater overhead since we have to communicate between them. We see this especially when we go from 4 processes running in a single machine to running 9 processes which requires at least 3 machines to run when using mapping by slot to assign processes. This adds a very large overhead that will not be compensated by having a lot of processing power since the input size is so small.

When we start to increase the input size we can start to see the benefits of having several processing cores, since the work load starts to benefit from having more processing power. However we can still see the significant performance

degradation when we go from using a single machine to execute our program to when we use several machines. Again, this is due to the overhead that is introduced when communicating across machines and our input size is not big enough yet to compensate for it. When using 16 processes we see an increase compared to Sequential and 9 processes but it is still slower than 4 processes in a single machine, showing that the communication overhead is still a problem.

In $N = 600$, we start seeing the overhead of running several different machines getting compensated by the processing power that we have access to. We can still see that going from 4 processes locally to 9 on more than one machine does not yield any extra performance, but it also doesn't lose a lot of performance either. When moving to 16 processes we see another slight increase but no where near linear performance scaling.

In the two largest input cases, $N = 900$ and $N = 1200$ we reinforce the learnings from the previous examples as we see a great increase in performance from going to 1 process to 4 processes (Locally) and we see small performance benefits from adding more processes when they're spread out across various machines, reinforcing that the overhead of using different machines requires a very large input to actually be worth it.

In conclusion, we see that the performance does not scale linearly even when the processes are in the local machine, showing the overhead that exists when adding more processes to solve a problem. It also shows that choosing the right amount of processes for the input size makes a great difference as we can actually lose performance when adding too many processes to an input size that will simply not benefit from them.

4 Difficulties encountered

When developing this assignment, we encountered a problem when sending the B matrix to the process below and receiving from the process above. First we used separate MPI_Send and then MPI_Recv to and from the corresponding processes, however when fox was almost finished executing, some processes would die without any sort of explanation and the program would hang until terminated by the user. This was solved by switching to the MPI_Sendrecv_replace method. This problem was probably due to some synchronization problem that was leading to either a deadlock between some processes or some other problem that was causing the process to not terminate correctly.
Appart from that bug, there were no other major difficulties when developing this assignment.

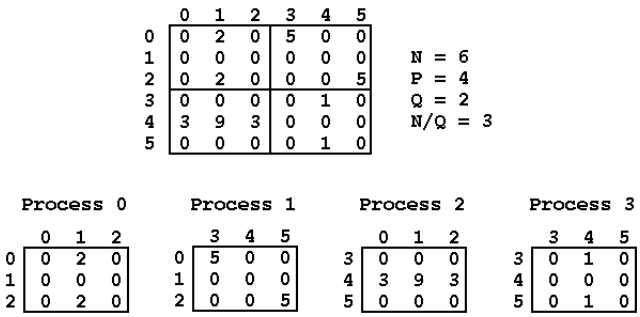

Figure 1: A visual representation of the division of the matrix by the processes