Project Assignment of Parallel Computing:

Project Assignment I: All-Pairs Shortest Path Problem

Determine the shortest path between nodes of a given graph using the Min-plus matrix multiplication algorithm combined with the Fox's algorithm for matrix multiplication using MPI

By: Yosbi Antonio Alves Saenz (201801653)

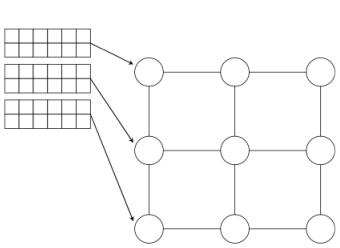
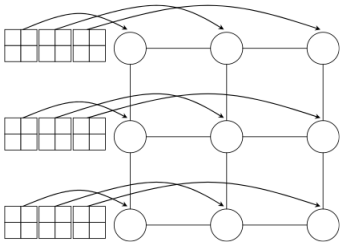
Faculdade de Ciencias
Universidade do Porto
Portugal
December 4, 2020

The algorithm implementation

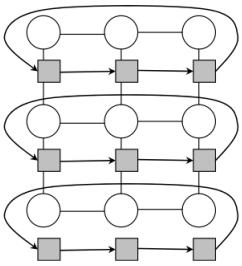
The main idea of the program is to speed up the min-plus matrix multiplication algorithm by partitioning the matrices in a chessboard block scheme and perform the multiplications (or the special matrix multiplications of the min-plus algorithm) using the Fox's algorithm taking advantage of parallelism using the MPI library. As input we have a matrix who is loaded into three matrices: The matrix A, the matrix B and the matrix C. The A and B matrix are to be multiplied and the result is stored in the C matrix.

In summary the program does the following:

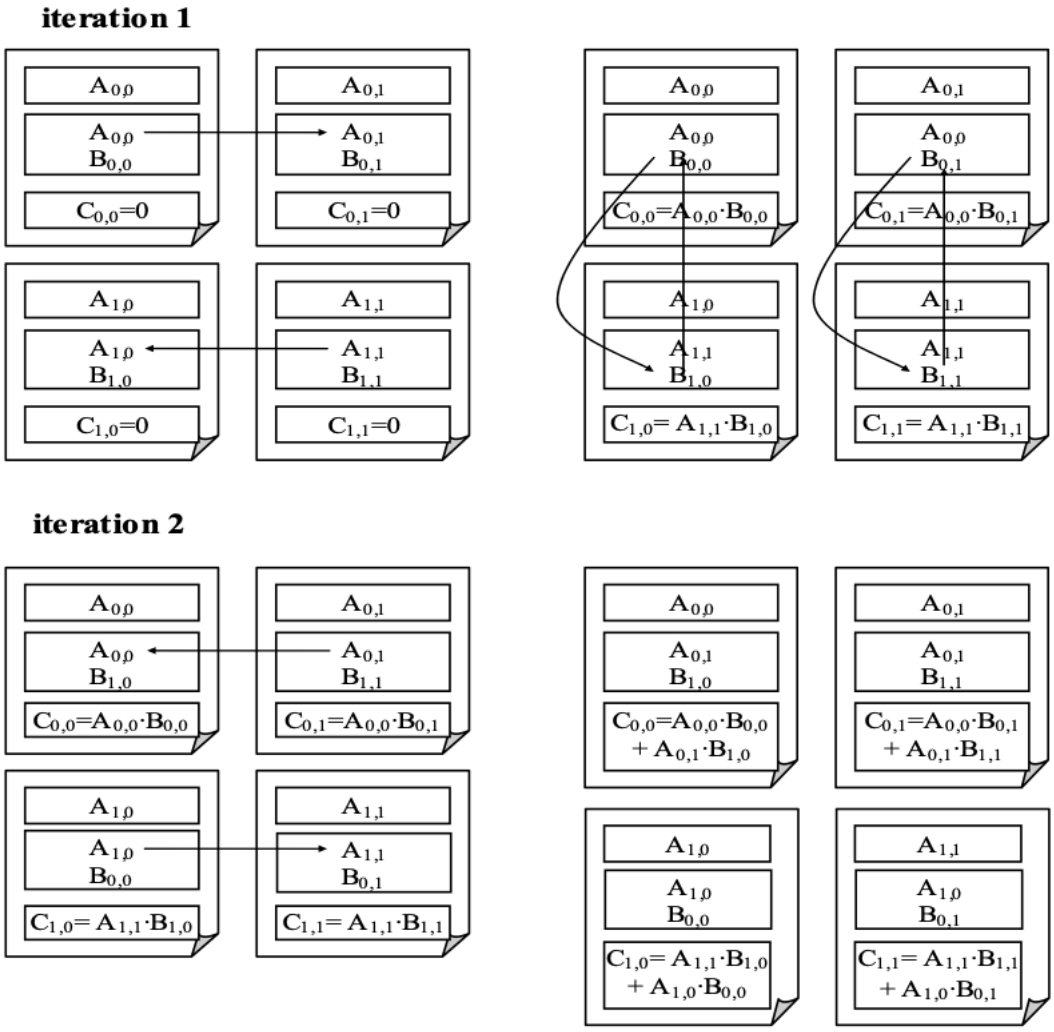
1. Validates the number of process to be a perfect square (so we can later divide the matrix).
2. Read the first line of input file (the size of the input matrix) and validate that the size of the matrix can be divided by the the size of the grid of processes. Then allocate the necessary memory of each process. Each process has four blocks of memory for the matrix: The initial block o f matrix A, the current block of the matrix A, the current block of the matrix B and the corresponding block of the result matrix C
3. Read the rest of the input file (the input matrix) and initialise the A, B and C matrices of the process 0. Because the characteristic of the min-plus algorithm we have to initialise each i,j elements when $i \neq j$, and they are in 0, to the value MAX_MATRIX_VALUE, that is simply the maximal value of integers in the programming language.
4. Create the grid of communicators. First we create one 2d cartesian communicator of all processes with periodicity in all ways, and from it we then create two communicators, one for the rows and one for columns of each process.
5. At this point we create a barrier to make sure that all the processes have all initialised and have all the communicators created to start the distribution of the data and the processing.
6. Distribute the matrices A, B and C to the corresponding blocks of each process, we do this by using the scatter function of the MPI library. First we scatter on each element of the first column the matrix, and then, each element of the first column will scatter to the rest of its own row the matrix, like this:

1-  2- 

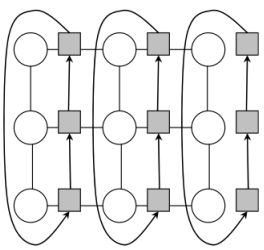
7. Then we do the Fox's algorithm for parallel matrix multiplication. In where:
   (a) For each row i, $0 \leq i \leq q$, the block Aij of $subtask(i, j)$ is transmitted to all the subtasks of the same grid row; index j, which defines the position of the subtask in the row, is computed according to the following expression: $j = (i + l) mod\, q$. We do this by using a broadcast on the row communicator we created in step 4.



(b) Blocks A'ij, B'ij obtained by each $subtask(i, j)$ as a result of block transmission are multiplied using the "special matrix multiplication", that it's like a normal multiplication of matrices but instead of using multiplication and sum operations, we use sum and minimum, respectively. So $Cij = min(Cij,\ A'ij + B'ij)$.



(c) Blocks B'ij of each $subtask(i, j)$ are transmitted to the subtasks, which are upper neighbours in the grid columns (the first row blocks are transmitted to the last row of the grid). We achieve this by using the MPI's function $MPI\_Sendrecv\_replace$, using the column communicator we created in step 4.



We iterate this steps until the num of iterations will be less than equal than the grid size.

8. We copy the result matrix C to the matrix A and B, to be able to do another iteration of the process if it is necessary

9. By iteratively repeating the steps 6 to 8, will allow us to obtain the matrix Df from matrix D1 by building successively matrices D2 = D1xD1, D4 = D2xD2, D8 = D4xD4, ... So we will repeat these steps until Df = DgxDg, with f $\geq$ N and g $<$ N.

10. Finally we iterate towards the resulting matrix C checking for any position Ci,j in where the value is equal to MAX_MATRIX_VALUE, in which case we change the value for 0

11. Present the resulting matrix C

12. Clean resources used by each process and exit

- **Assing:** The semantics of the rule is the following:

$$(x := n, s) \Rightarrow (s[x \rightarrow A[n]s)$$

In which we assign the value of a variable in the state s, if the variable does not exist inside the state, we insert the variable with the value into the state.

- **Skip:** The semantics of the rule is this:

$$(skip, s) \Rightarrow s$$

In which we abort the execution of a sequence of commands and return the state.

- **Arithmetic:** The semantics of this rule are the following:

$$\frac{(S_1, s) \Rightarrow n_1}{(S_1 + S_2, s) \Rightarrow (n_1 + S_2, s)}; \frac{(S_2, s) \Rightarrow n_2}{(n_1 + S_2, s) \Rightarrow (n_1 + n_2, s)}; (n_1 + n_2, s) \Rightarrow (n, s)$$

In which we first evaluate the variables doing a lookup in the state, we also evaluate the constants integers, and then we perform the operation giving a new state. We can do addition, subtraction and multiplication.

- **Sequence:** The semantics of the rule are the following:

$$\frac{(S_1, s) \Rightarrow (S_1', s')}{(S_1, S_2, s) \Rightarrow (S_1', S_2, s')}; \frac{(S_1, s) \Rightarrow s'}{(S_1, S_2, s) \Rightarrow (S_2, s')}$$

In which we can have a sequence or a sequence of sequences that, for example:

a. $Sequence[S_1, S_2]s$. In which we execute S_1, and we now have $Sequence[S_2]s'$.

b. $Sequence[S_1[S_a, S_b, S_c], S_2[S_d, S_e, S_f]]s$. In here we will execute the statement S_a of S_1 in witch we now have: $Sequence[S_1[S_b, S_c], S_2[S_d, S_e, S_f]]s'$ for execute.

- **Par:** The semantics of this rule are the following:

$$\frac{(S_1, s) \Rightarrow (S_1', s')}{(S_1 par S_2, s) \Rightarrow (S_1' par S_2, s')}; \frac{(S_1, s) \Rightarrow s'}{(S_1 par S_2, s) \Rightarrow (S_2, s')}; \frac{(S_2, s) \Rightarrow (S_2', s')}{(S_1 par S_2, s) \Rightarrow (S_1 par S_2', s')};$$
$$\frac{(S_2, s) \Rightarrow s'}{(S_1 par S_2, s) \Rightarrow (S_1, s')}$$

Care must be taken because while doing parallelism, because this semantic can be non-deterministic as we can have, for example, three results for the sequence of commands $(x := 1 \; par \; [x := 2; x := x + 2], s)$:

1. $\Rightarrow (x := 2; x := x + 2; s[x \Rightarrow 1]) \Rightarrow (x := x + 2; s[x \Rightarrow 2]) \Rightarrow s[x \Rightarrow 4]$
2. $\Rightarrow (x := 1 \; par \; x := x + 2; s[x \Rightarrow 2]) \Rightarrow (x := 1; s[x \Rightarrow 4]) \Rightarrow s[x \Rightarrow 1]$
3. $\Rightarrow (x := 1 \; par \; x := x + 2; s[x \Rightarrow 2]) \Rightarrow (x := x + 2; s[x \Rightarrow 1]) \Rightarrow [x \Rightarrow 3]$

The implementation of the SMALL language

The SMALL language has an interpreter who is programmed in Haskell. The implementation of the interpreter is divided into the following way:

- **Evaluator:** Who's work is solving the arithmetic expressions like $(x+2)$ or $x = (5*2)$, all the arithmetic expressions must be surrounded by parentheses '(' ')'. It also evaluates the value of a variable inside a State and the value of a constant integer
- **Executor:** The executor is in charge of executing the statements of the programming language, the statements can be:

  a. The assignation of a variable, for example: "$x = 2$",

  b. A sequence of statements, for example $[x = 2; y = 3; z = (y * x)]$,

  c. The par command for parallel execution of sequences of statements, for example $[x = 2; y = 3; z = (x * y)]par[x = 5; y = 6; z = (x * y)]$. The behaviour of this command can be non-deterministic as we saw on the rules of the semantics. We accomplish this by implementing a command planner. The command planner does a pseudorandom selection of what command on top of each list is going to be executed by using a random number generator, in which, we randomly choose a number between 0 to 100, and if it is less than 50 we choose to execute the command from the first list, and if the number is greater than equal than 50 we choose to execute the command from the second list, and so on.

  d. A skip command that stops the execution of a sequence of commands and return the current state, for example $[x = 1; x = 2; skip; x = 3]$ returns the state $[("x", 2)]$
- **Parser and Lexer:** who's work is to take the input string of the user and parse it into a form that the executor and the evaluator can understand, for example, the command "$[x = 7]par[x = 2; x = (5 * x)]$" will be translated to the form "Par [Assing "x" (Num 7)] [Assing "x" (Num 2), Assing "x" (Mult (Num 5) (Var "x"))]". The Parser and the Lexer were made using the tool "happy" (`https://www.haskell.org/happy/`). The file Parser.y can be compiled into Parser.hs by using the command "*happy Parser.y*". **Note:** you must install happy first with the command "cabal install happy", but it won't be necessary as I am providing the Parser.hs already compiled by the happy tool.

The usage of SMALL is very simple, you just have to call the interpreter and provide a sequence of commands in string format and a state:

$$> interpreter\ "commands\_sequence"\ state$$

The commands sequence can be a single statement, an array of statements (see the examples below)

The estate is an array of tuples, the tuples are of type (String, Int). The state can be empty "[ ]" or have some values already defined "$[("y", 1), ("x", 2)]$", which means $y = 1$ $x = 2$.

Some examples of usage:

- $*Main > interpreter$ "$x = 2$" [ ] with result $[("x", 2)]$
- $*Main > interpreter$ "$x = (2 + 2)$" [ ] with result $[("x", 4)]$
- $*Main > interpreter$ "$x = (2 + y)$" $[("y", 5)]$ with result $[("x", 7), ("y", 5)]$
- $*Main > interpreter$ "$[x = 1]par[x = 2; x = (2 + x)]$" [ ] with result or $[("x", 4)]$ or $[("x", 3)]$ or $[("x", 1)]$
- $*Main > interpreter$ "$[x = 1; x = 2; skip; x = 3]$" [ ] with result $[("x", 2)]$

References

- H.R. Nielson, F. Nielson. Semantics With Applications A Formal Introduction. Revised edition (1999).
- Lindsey Kuper. Write an interpreter (2013). Accessed on: November 1st 2020, in: `http://composition.al/blog/2013/06/23/write-an-interpreter/`
- Lindsey Kuper. Write an interpreter: variables (2013). Accessed on: November 1st 2020, in: `http://composition.al/blog/2013/06/29/write-an-interpreter-variables/`
- Madhavan Mukind, National Programme on Technology Enhanced Learning (NPTEL). Functional Programming in Haskell (2015). Accessed on: October 14 2020, in: `https://nptel.ac.in/courses/106/106/106106137/`
- Happy. Acessed on: November 1st 2020, in `https://www.haskell.org/happy/`