Assignment I: All-Pairs Shortest Paths

Hugo Oliveira[1]

November 7, 2016

[1]University of Porto, Portugal.

0.1 Algorithm Implementation

The function processArguments() process input arguments and checks it the are valid. If all valid returns an OK situation putting the file name of input and output in the received pointed vars.

```
int processArguments(int nArgs, char *argv[], char * inName, char *outName){
...
}
```

The function allocate2DMatrix() creates an 2D matrix with the specified dimensions and return the pointer to its heap location.

```
int ** allocate2DMatrix (int nRows, int nCols){
...
matrix = (int **) malloc (nRows*sizeof(int *));
    if (!matrix) return (NULL);

    for (i=0; i < nRows; i++) {
        matrix[i] = (int *) malloc (nCols*sizeof(int));
        if (!matrix[i]) return (NULL);
    }
}
```

The function allocate1DMatrix() creates an 1D array to be used as chunk in rows and columns communications of MPI.

```
int ** allocate2DMatrix (int nRows, int nCols){
...
int * allocate1D(int nRows, int nCols)
}
```

The function readFileToMatrix() read the number of columns in file name and their contents parsing to numbers to the matrix, returning an OK situation or error if something went wrong to be later handled. The number of rows and the 2D matrix pointer are returned in the arguments.

```
int readFileToMatrixTest(char* fileName, int *nRows, int *matAdd){
...
    FILE *fp = fopen(fileName,"r");
    if(fp ==NULL){
        printf("Cannot open file\n\r");
        return -1;
    }
    int ** mat = allocate2DMatrix(n,n); // allocate the matrix
    ....// parse things
    fclose(fp);
    // read input file
    *nRows =n; // Fill n
    // return matrix
    printf("All read\n");
    *matAdd = mat;
    return 1; // ok condition
}
```

A function checkConditions() check if in the current setup the number or process is an perfect square regarding the number of rows of the matrix (assum-

ing square matrix)

```
int checkConditions(int P, int Q, int nRows){
...
}
```

The function replaceZerosByInf() replaces zeros out of the diagonal matrix by -1 and the replaceInfByZeros() does the inverse process.

The floydWahshal() receives the A, B and C vector chunks to perform the algorithm. If the value of matrices are negative, nothing is done on vector C, else the correspondent chunk is replaced by index plus sub matrix size according to the sqrt of number o available process and number of rows/columns.

```
void floydWahshal(int nSize, int *aMatrix, int *bMatrix, int *cMatrix) {
    ...
    for (i = 0; i < nSize; i++) {
        for (j = 0; j < nSize; j++) {
            for (k = 0; k < nSize; k++) {
                // we waste three cicles on this
                a = aMatrix[k + i * nSize]; // grab the value in the vector
                b = bMatrix[j + k * nSize];
                c = cMatrix[j + i * nSize];

                if (a != INF && b != INF) { // Only of not inf
                    if (c == INF || c > a + b)
                        cMatrix[j + i * nSize] = a + b;
                }
            }
        }
    }
}
```

The algorithm start to setup grid conditions and the ROOT process start to parse the arguments, read the file and allocated matrix. The root process check if the conditions are meet and broadcast the sub matrix size if conditions are met or -1 case in any of the of the his tasks fails. This enables that all process are informed of the sub matrix size, and in any error they can terminate safely calling the MPI Finalize().

After determining the sub-size of matrices, the ROOT process starts to slice the 2D matrix into vector chunks in order to serialize the matrix chunks. However there are two genre of vector chunks that have the size of the sub matrix, the localMatrix, and the tempMatrix, the localMatrix is used by the ROOT and all remainder process to capture the chunks while the tempMatrix is only used by the ROOT to build the remainder process matrix chunks and send then using the for I indices. This procedure avoids the ROOT process to send to and receive an matrix chunk that he already posses. This saves one communication to the same process. THE MPI Send() takes the sub matrix vector and the process that are different from ROOT and sends the data.

With all sub matrices announced to the correspondent process, all remainder process start to gather the current sub matrix size that encapsulates also condition errors that may occur during initialization. If any error condition was broadcast, all process invoke MPI Finilize() and terminate safely. In normal conditions they obtain their chunk size and receive their correspondent vector matrix invoking MPI Recv().

After this stage the grid is created that create the rows and col communicators and setup the Cartesian matrix for the process. All process allocate their auxiliary chunks, rowMatrix, colMatrix and resMatrix and if the rank of the process correspond to one diagonal, it makes an local copy of localMatrix to rowMatrix to be used in FOX algorithm. All process make local copies of their localMatrix to colMatrix and resMatrix to be used. resMatrix attains the result of floydWhasrshal and the initial step contains the values that will be updated in the FOX stages iterations. For robustness, all allocation are tested to see if they are all succeeded, otherwise the MPI Finalize is called.

Now the FOX process takes place and if the process is the diagonal itself broadcasts the rowLines of its local matrix otherwise receives the broadcasted line. When FOX algorithm computes the floydWharshal and updates the resMatrix with partial computations. MPI Sendrecv replace make the column matrix shift on each stage. For each pairD_f with $f = 2^n$, localMatrix and colMatrix are updated with the partials results for that are contained are in resMatrix. The number of runs of FOX algorithm is $Log_2(N)$ but to avoid sending N original we multiply subMatrixSize with the sqrt of the number of process and multiply by two to resemble the log operation and the for cycle starts at 2 and goes in powers of two (2, 4, 8, etc)

After all done, all the process take the resMatrix with the partial computed matrix and mapped into the final vector solution solMatrix. Then the ROOT process starts to make the inverse process of converting the array vector solution vector with the results back to the 2D original matrix format. After all finished, the ROOT process replaces the original zeros that were outside of the matrix diagonal and gathers the time results and free the main Matrix itself while all process free their matrix vectors chunks and MPI is terminated.

The test trials were performed with the test case with 300 rows and for each configuration five runs were made.

0.2 Performance Evaluation

Setup	Max Time	Min Time	Average
1 Core	2.507	2.451	avg 2.49
4 Core	0.623	0.701	avg 0.648
9 Core	0.63	0.64	avg 0.667
4 Machines Core 4	0.489	0.483	avg 0.486
16 Machines Core 1	0.4661	0.41	avg 0.429

The setup with 1 core only has substantially more computing time due to the need o perform an sequential multiplication. Using 4 cores with one machine decreases the time significantly, however using an an more broader configuration that uses 9 cores, ate least two more machine involved, doesn't improve significantly the computation time due to the fact that the trade between communication cost and computing time was almost equal. However using more cores, we drastically decrease the time involved in partial computations with almost the same amount of communications cost.

The using 16 cores 4X4 or 16*1 should present significant time changes due to the act that more communications need to be done between different machines,

however we cannot verify completely this situation due to the fact that all 16 computers were not always connected.

0.3 Difficulties and improvements

The improvements can be made regarding the use of MPI Isend and MPI Irecv instead of using the normal API in order to circumvent the buffer limits that may occur using large matrices. Also the reading process was slightly altered in order to the temporally line string can hold any length. The readFileToMatrix() function allocates an$lineString = malloc(sizeof(char) * (n * 2 + 1))$; with n being the number of rows and assuming that its separated by spaces, this number is doubled to attain all. After used, the function releases the memory chunk.

Bibliography

[1] Acetatos Abo Akademi University - Matrix multiplication

[2] Thomas Anastasio, Example of Matrix Multiplication by Fox Method, November 23, 2003

[3] Ned Nedialkov, Communicators and Topologies: Matrix Multiplication Example, Dept. of Computing and Software McMaster University, Canada, January 2012

[4] Graph Theory: Finding all-pairs shortest paths and counting walks.