Ca’ Foscari University of Venice

Department of Environmental Science, Informatics and Statistics

Master Degree Computer Science

Course of High Performance Computing

(A.A. 2013 - 2014)



dense matrix-matrix multiplication

Versione 1.0 del 21 May 2014

Prof. Orlando salvatore

*A cura del gruppo*:

|  |  |  |
| --- | --- | --- |
| Barbon Gianluca | e-mail: | *gbarbon@dsi.unive.it* |
| De Zotti Cristian | e-mail: | *cdezotti@dsi.unive.it* |

Summary

1. Introduction 4

2. Development Environment 5

3. Implementation 6

3.1 Header files 6

3.1.1 Stopwatch.h 6

3.1.2 Header.h 6

3.1.3 InOut.h 7

3.2 Common Operations, Behaviours, Data Structures 7

3.2.1 Launch parameters 7

3.2.2 Input / output 7

3.2.3 CSV file 7

3.2.4 Data Structures 8

3.2.5 Free functions 8

3.3 Cannon Algorithm 8

3.3.1 Matrix division 8

3.3.2 Matrix align 9

3.3.3 Communication (from master) 9

3.3.4 Slave process core loop 9

3.3.5 Communication (to master) 11

3.3.6 Result matrix computation 11

3.4 Processor Farm 11

3.4.1 Communication of B (from master): 11

3.4.2 First communication of A rows (from master): 12

3.4.3 Multiplication and communication loop (slave processes) 12

3.4.4 Communication of consecutive A rows and result matrix computation 13

3.5 Matrix Matrix multiplication 13

3.5.1 Communication (from master) 13

3.5.2 Communication (among slave processes) 14

3.5.3 Multiplication 14

3.5.4 Communication (from slave) 15

3.5.5 Result matrix computation 15

3.6 Matrix Matrix Fast multiplication 15

3.6.1 Communication (from master) 16

3.6.2 Multiplication 16

3.6.3 Communication (from slaves) 17

3.6.4 Result matrix collection 17

4. Testing Phase 18

4.1 Environment Description 18

4.2 Algorithms times 18

4.2.1 Heavy function 18

4.2.2 No-Input/Output Vs Input/Output 18

4.2.3 No-Optimization Vs Optimization 19

4.2.4 Number of process Vs Time 19

4.2.5 Matrix dimension Vs Time 23

4.2.6 Matrix Matrix multiplication 25

4.2.7 Cannon Algorithm 25

4.2.8 Processor Farm 25

4.2.9 Matrix Matrix Fast multiplications 25

5. Conclusion 25

6. Appendix I: MPI primitives 25

6.1 MPI\_Send 25

6.2 MPI\_Recv 26

6.3 MPI\_Sendrecv\_replace 26

7. Appendix II: User Manual 27

8. Bibliography 27

# Introduction

The project consists of the implementation of three parallel algorithms for matrix multiplication.

The implemented algorithms differ according to the process of parallelization achieved:

* **Matrix Matrix multiplication:** static assignment of job processes; communication mechanisms are needed to provide the sub-block possessed all the processes that have a block that is in the same row or column of the possessed.
* **Cannon Algorithm:** static assignment of job processes; before applying the algorithm must align the matrices. The exchange of sub-blocks between processes is via a rotation of the sub-blocks owned by the process.
* **Processor Farm:** dynamic mapping of job process; the allocation of the blocks is performed from the master and do not have any iteration between the worker.

The project was implemented using the standard **MPI** (Message Passing Interface). MPI is the de facto standard for communication between nodes on a cluster of computers running a parallel program developed for distributed-memory systems. MPI compared to previous libraries used for passing parameters between nodes, has the advantage of being very portable (MPI has been implemented for many parallel architectures) and fast (MPI is optimized for each architecture).

**Constraints of the project:**

* The input data (generate synthetic floating-point FP matrixes, stored row-major as ASCII files) are stored on the disk, and can be accessed by processor master only:
  + (don’t use the mounted HOME, but the local mounted disk).
  + data blocks must be transmitted/scattered from processor master to all the others (algorithm 1 and 2).
  + the master of the processor farm (algorithm 3), which is responsible for data distribution to all the slaves, must thus be mapped on processor 0 in order to permit data transmission / collection.
* Also the output matrix must be collected by processor 0 and stored on the disk.
* However, it must be possible to disable the input/output of data from/to the disk, to remove the influence of the input/output on the total execution cost.
* You can iterate the matrix-matrix multiplication several times.
* The simple computation of a matrix-matrix multiplication cannot be enough for taking advantage of current parallel platforms. Compute instead:
* C = f(A) x B. f() is function that can independently be applied to all A[i,j]. f() emulate the application of an expensive function by running a loop and this increases the minimum granularity of parallel tasks.

**Goals:**

* Evaluate speedups for the different versions
* with different granularities (cost of function f()).
* with different problem sizes.
* note that for very large data size, sequential version can incur in trashing (superlinear speedup).
* Evaluate system imbalance or heterogeneity (optional)
* for example, by running synthetic workloads on some processors.
* the algorithm based on the processor farm (3) is naturally able to cope with these issues.
* the other two algorithms (1 and 2), based on static block-wise distribution of arrays, may suffer from this imbalance.

# Development Environment

# Implementation

## Header files

All the program version rely on three header files for common operations.

### Stopwatch.h

This header supply a stopwatch, in order to count the execution time of a program (or a program portion). It defines a new tipe (Stopwatch) to collect start and end time.

* ***Stopwatch StopwatchCreate()*** Allocates and initializes a stopwatch.
* ***void StopwatchStart(Stopwatch chrono)*** Starts the time of a stopwatch.
* ***void StopwatchStop(Stopwatch chrono)*** Stops the time of a stopwatch.
* ***float StopwatchGetElapsed(Stopwatch chrono)*** Gets the difference between the starting and the ending time.
* ***int StopwatchIsValid(Stopwatch chrono)*** Checks whether the time is a valid number or not.
* ***void StopwatchPrintResolution()*** Prints the time resolution.
* ***void StopwatchPrintWithComment(const char\* message, Stopwatch chrono)*** Prints the time with an additional comment.
* ***void StopwatchPrintToFile(const char\* text, Stopwatch chrono)*** Appends the time with an additional comment to an external file.

### Header.h

This header file includes all the C libraries needed by the programs. Moreover, it defines a list of useful functions for matrix manipulation.

* ***double\*\* matrix\_creator(int a, int b)*** Creates an axb matrix of doubles and returns a pointer to this matrix.
* ***void matrix\_init(double\*\* mat, int n)***Initializes a nxn matrix with random values.
* ***void simple\_matrix\_init(double\*\* mat, int n)*** Initializes a nxn matrix with a simple stream of integers.
* ***void freematrix(int n, double\*\* mat)*** Frees an nxn matrix.
* ***void printmatrix(int a, int b, double\*\* C)*** Prints an axb matrix.
* ***void printvector(int a, double\* C)*** Prints a vector of length "a".
* ***double \* matrix\_vectorizer(int a, int b, double \*\* mat)*** Transforms an axb matrix in a vector, by listing all the rows in sequence, and returns a pointer to this vector.
* ***double \*\* devectorizer(int a, int b, double \* vet)*** Transforms a vector in a axb matrix, by splitting the vector in rows, and returns a pointer to this matrix.
* ***void matrix\_transposer(int n, double \*\* A)***Transposes a square matrix.
* ***double heavy(double a, int h)*** Function that increase the computational cost, in order to increase the minimum granularity of parallel tasks. This function is parametrized, this means that it allows to set the additional load.

### InOut.h

This header supply all the input output operations to the programs. It offers only two methods, the first one for the matrix loading, while the second one is for the matrix writing. The input and the output are done only from and to csv ASCII files.

## Common Operations, Behaviours, Data Structures

In the following paragraphs we describe the common operations / behaviours to all the versions.

### Launch parameters

Each version of the program must be launched with mpiexec command and a given set of parameters. Let’s take a sample execution:

mpiexec -n 5 ./bin/hpc\_mpi\_mm-fast-op 16 0 0

The first two arguments are common for every MPI program launch and are respectively the number of processes involved (notice that, in example, number 5 will stand for 1 master + 4 slaves processes) and executable to launch. The other arguments are introduced in this project and stands for:

* Input and output matrices dimension
* Boolean value for input/output activation (‘0’ stands for no input/output, ‘1’ for the opposite)
* Last value specifies the type of load function: 0 stands for no load factor, 1 for basic load factor, 2 for heavy load factor.

### Input / output

Input output operations are granted by the inout.h header. They are not compulsory for the program execution, but they can be used to load external square matrix and to store the result into an output file. In case they are not chosen, system will create autonomously input matrices and won’t save the final matrix. System created matrix are initialized using the function simple\_matrix\_init from the header file header.h (please refer to header files paragraph in order to see functions behaviour).

### CSV file

A csv file, named testfile.csv, is used in order to store the behavior of the chosen configuration. An example line of this file will be:

1399974683,mm-op,4,192,io,func2,0.454699

* First value is the time in seconds since an arbitrary time in the past given from function MPI\_Wtime(). It is used as an id for each execution (because simultaneous executions are not expected).
* Second is the name of the chosen version.
* Third show the number of process involved in the matrix calculation (notice that one process is always used as master and does not perform calculations; this process is not counted in this number, so as for the example, in reality we have 4+1 total process).
* Fourth number shows the dimension of input and output matrices.
* Fifth field shows whether input/output option is active or not (respectively, it will shows “io” and “no\_io”).
* Sixth field shows whether the load function is active or not and, in case it’s working, the type of function (possible values: “low”, “func1”, “func2”).
* Last field shows the total execution time of the configuration described by the previous parameters.

Csv file generation is performed through the use of the StopwatchPrintToFile function from the MpiStopwatch header, that will allow to attach the result of the current execution at the end of the testfile.csv. This will allow to collect all the time results in a unique file.

Csv file type has been chosen because of its simple construction and also because this kind of file can be imported in excel in order to easily generate final charts.

### Data Structures

The only data structure implemented is in the MpiStopwatch.h header, in order to maintain in a unique structure starting and ending time.

Other data structure could have been implemented, for example for matrices, but we chose to not implement any because of possible performance deterioration.

### Free functions

At the end of each program we can find a list of free function in order to free all memory structures created with malloc in the heap. Moreover, various free are also present inside functions to free temporary data structures and we have implemented a dedicated free function for 2D matrix in the header.h file.

## Cannon Algorithm

### Matrix division

Matrices must be divided in a number of blocks equal to the number of process (set at the beginning). Each of these blocks must thus be sent to the respective processor. This work is done in the master process with the function matrix\_block :

double \*\* matrix\_block(double\*\* matrix, int n, int nblock) {

int i, j, k, x, y, offset = n / sqrt(nblock);

double \* tempM, \*\*block, \*\* Mblock;

block = matrix\_creator(offset, offset);

Mblock = (double \*\*) malloc(nblock \* sizeof (double\*));

k = 0;

for (i = 0; i < n; i += offset)

for (j = 0; j < n; j += offset) {

for (x = i; x < offset + i; x++)

for (y = j; y < offset + j; y++) {

block[x - i][y - j] = matrix[x][y];

}

/\*vectorize the two pieces of matrices\*/

tempM = matrix\_vectorizer(offset, offset, block);

Mblock[k] = tempM;

k++;

}

freematrix(offset, block);

return Mblock;

}

### Matrix align

After receiving, matrices must be aligned in the master processes with a skewing method. The alignment must be done basing on the order of blocks assigned to each processor. This work is done in the master process with the functions skewing\_row and skewing\_column. Below there’s the code of the skewing\_row function (skewing\_column function is not listed here, but has a similar behaviour):

void skewing\_row(double \*\* M, int n) {

int i, j, k, index;

double \*\*r\_swap = (double \*\*) malloc(sizeof (double\*) \* n);

for (i = 0; i < n; i++) {

r\_swap[i] = M[i];

}

k = 1;

for (i = sqrt(n); i < n; i = i + sqrt(n)) {

for (j = 0; j < sqrt(n); j++) {

index = (j + k) % (int) sqrt(n);

M[i + j] = r\_swap[i + index];

}

k++;

}

}

### Communication (from master)

First communication consists in telling passing to each process the data needed.

MPI\_Send(Ablock[i], numElements, MPI\_DOUBLE, i, TAG, MPI\_COMM\_WORLD);

MPI\_Send(Bblock[i], numElements, MPI\_DOUBLE, i, TAG, MPI\_COMM\_WORLD);

### Slave process core loop

After having received the first two blocks (row and column blocks), the slave internal loop performs the core operations of the cannon algorithm in each loop iteration: multiplication and block sending and receiving.

for (index = 0; index < lato; index++) {

A = devectorizer(dim, dim, Ablock[0]);

B = devectorizer(dim, dim, Bblock[0]);

matrix\_transposer(dim, B);

// Multiplication

matrix\_mult(A, B, C, dim, load\_bool);

row\_dest = getRankRowDest(myrank, nblock);

row\_mit = getRankRowMit(myrank, nblock);

// block A

MPI\_Sendrecv\_replace(Ablock[0], numElements, MPI\_DOUBLE, row\_dest, 1, row\_mit, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

col\_dest = getRankColDest(myrank, nblock);

col\_mit = getRankColMit(myrank, nblock);

// block B

MPI\_Sendrecv\_replace(Bblock[0], numElements, MPI\_DOUBLE, col\_dest, 1, col\_mit, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

Let’s see in the details this loop.

Multiplication is done for a partial block through the matrix\_mult function:

void matrix\_mult(double\*\* A, double\*\* B, double\*\* C, int dim, int load) {

int i, j, k, l;

for (i = 0; i < dim; i++) {

l = 0;

for (j = 0; j < dim; j++) {

for (k = 0; k < dim; k++) {

if (load == 0)

C[i][j] += A[i][k] \* B[l][k];

else

C[i][j] += heavy(A[i][k], load) \* B[l][k];

}

l++;

}

}

After a multiplication step a process can communicate its own block of A to the right neighbour and receive the new block of A from the left neighbour. Furthermore, we will have also the communication of B blocks in the vertical direction, by sending our B block to the upper neighbour and by receiving the new block from lower neighbour.

Anyway, each process doesn't know the rank of neighbour processes and of itself too. We thus must calculate the coordinates with these 4 functions:

* getRankRowDest
* getRankRowMit
* getRankColDest
* getRankColMit

We show the code of only the first one of these functions, because the behaviour is quite similar:

int getRankRowDest(int rank, int np) {

int rank\_dest;

int proc\_lato = (int) sqrt(np);

if ((rank % proc\_lato) == 0)

rank\_dest = rank + (proc\_lato - 1);

else

rank\_dest = rank - 1;

return rank\_dest;

}

Now the process can send/receive columns and rows to/from its neighbours.At the end we will have a matrix shift (see figure below).

This communication process will continue among with multiplication step until each process has received all the blocks of the same rows of A and of the same columns of B.

### Communication (to master)

Now we have the final blocks. These blocks are thus transformed in array and sent to the master, that will have a unique 1D array for the whole resulting matrix.

double \*C\_vett = matrix\_vectorizer(dim, dim, C);

MPI\_Send(C\_vett, numElements, MPI\_DOUBLE, master, TAG,MPI\_COMM\_WORLD);

### Result matrix computation

Finally the master can compute the resulting matrix by collecting all the pieces from every process:

for (i = 0; i < nblock; i++) {

MPI\_Recv(tempC[offset], numElements, MPI\_DOUBLE, i, TAG, …);

offset++;

}

double \*C\_vett = matrix\_vectorizer(nblock, numElements, tempC);

block\_matrix(C, C\_vett, nblock, N);

Notice that function block\_matrix is needed to transform the final matrix, stored in a 1D array, into a “real” 2D matrix.

## Processor Farm

### Communication of B (from master):

We first allocate in every slave the space needed to receive the B matrix. At this time master can communicate to every process the whole matrix.

Bvett = (double \*) malloc(sizeof (double) \* N \* N);

MPI\_Recv(Bvett, N\*N, MPI\_DOUBLE, 0, TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

B = devectorizer(N, N, Bvett);

### First communication of A rows (from master):

Master process starts to scan matrix A and to communicate at each process each row (in case we have more process than rows, it will communicate -1 as identifier, in order to state to the slave process the termination). Notice that the communication is split into two part, one for sending the index and the other for the row sending.

for (i = 0; i < numnodes - 1; i++) {

if (index < N && index != -1) {

MPI\_Send(&index, 1, MPI\_INT, i + 1, i + 1, MPI\_COMM\_WORLD);

MPI\_Send(A[index], N, MPI\_DOUBLE, i + 1, i + 1, …);

index = index + 1;

} else {

index = -1;

MPI\_Send(&index, 1, MPI\_INT, i + 1, i + 1, MPI\_COMM\_WORLD);

}

}

### Multiplication and communication loop (slave processes)

Now each slave process that possesses a rows of matrix A, can go on with the computation of the resulting rows. This process consists in a loop composed of three steps (A rows receiving, multiplication, result sending) and is last until reception of -1 index (condition for while loop).

MPI\_Recv(&index, 1, MPI\_INT, 0, myrank, MPI\_COMM\_WORLD, …);

while (index != -1) {

MPI\_Recv(rigaA, N, MPI\_DOUBLE, 0, myrank, MPI\_COMM\_WORLD, …);

for (j = 0; j < N; j++)

ris[j] = 0.0;

for (j = 0; j < N; j++) {

for (i = 0; i < N; i++) {

if (load\_bool == 0)

ris[j] += rigaA[i] \* B[i][j];

else

ris[j] += heavy(rigaA[i], load\_bool)\*B[i][j];

}

}

MPI\_Send(&index, 1, MPI\_INT, 0, TAG, MPI\_COMM\_WORLD);

MPI\_Send(ris, N, MPI\_DOUBLE, 0, TAG, MPI\_COMM\_WORLD);

MPI\_Recv(&index, 1, MPI\_INT, 0, myrank, MPI\_COMM\_WORLD, …);

}

It’s easy to see that at the end of a single row computation, the slave process sends the results among with the index to the master process, that will collect the received row in the correct rows of the resulting matrix.

MPI\_Send(&index, 1, MPI\_INT, 0, TAG, MPI\_COMM\_WORLD);

MPI\_Send(ris, N, MPI\_DOUBLE, 0, TAG, MPI\_COMM\_WORLD);

Data needed for next iteration are collected through the two MPI\_Recv at the beginning and at the end of the loop. The last of the two receive is used for collecting the index and to catch the -1 index in order to stop the loop.

### Communication of consecutive A rows and result matrix computation

Master will receive each computed block from slave process and will continue to send them new rows to compute with the following loop:

while (recv < N) {

for (i = 0; i < numnodes - 1; i++) {

if (recv < N) {

MPI\_Recv(&indexR, 1, MPI\_INT, i + 1, TAG, …);

MPI\_Recv(C[indexR], N, MPI\_DOUBLE, i + 1, TAG, …);

recv++;

}

if (index < N && index != -1) {

MPI\_Send(&index, 1, MPI\_INT, i + 1, i + 1, …);

MPI\_Send(A[index], N, MPI\_DOUBLE, i + 1, i + 1, …);

index = index + 1;

} else {

index = -1;

MPI\_Send(&index, 1, MPI\_INT, i + 1, i + 1,…);

}

}

}

At the end of the loop we will obtain the complete result matrix.

## Matrix Matrix multiplication

Our matrix-matrix multiplication version differs from the standard implementation. Indeed, we use the cannon structure as base for the communication, but we changed it in the content of the communication and in the multiplication, in order to implement the matrix matrix algorithm.

### Communication (from master)

The two matrices are first divided into blocks through the matrix\_block function (see the cannon algorithm paragraph for function description), basing on the number of available processes.

After this first step master can send to slaves processes the matrices blocks.

Ablock = matrix\_block(A, N, nblock);

Bblock = matrix\_block(B, N, nblock);

numElements = dim\*dim;

for (i = 0; i < nblock; i++) {

MPI\_Send(Ablock[i], numElements, MPI\_DOUBLE, i, TAG, MPI\_COMM\_WORLD);

MPI\_Send(Bblock[i], numElements, MPI\_DOUBLE, i, TAG, MPI\_COMM\_WORLD);

}

### Communication (among slave processes)

Similarly to the cannon algorithm, we first receive the two blocks from the master. After that we perform a loop that allow the slave process to collect and send from and to the other processes the remaining parts of desired rows and columns. In order to do this we use the MPI\_Sendrecv\_replace function.

for (i = 1; i < lato; i++) {

row\_dest = getRankRowDest(myrank, nblock);

row\_mit = getRankRowMit(myrank, nblock);

// block A

MPI\_Sendrecv\_replace(Aswap[0], numElements, MPI\_DOUBLE, row\_dest, 1, row\_mit, 1, …);

col\_dest = getRankColDest(myrank, nblock);

col\_mit = getRankColMit(myrank, nblock);

// block B

MPI\_Sendrecv\_replace(Bswap[0], numElements, MPI\_DOUBLE, col\_dest, 1, col\_mit, 1, …);

...

}

As for the getRank functions please refer to the Cannon algorithm paragraph. Notice that this algorithm differs considerably from the Cannon algorithm because multiplication is not done step by step inside the loop, but it's only done one time per slave at the end of the data collection.

### Multiplication

At the end of the previous loop the slave process possesses the entire rows and columns needed for its multiplication part. Anyway, data must first be ordered.

for (j = 0; j < lato \* dim; j += dim)

for (x = 0; x < dim; x++)

for (y = j; y < dim + j; y++) {

A[x][y] = Ablock[0][el];

el++;

}

for (x = 0; x < dim; x++) {

el = 0;

for (j = 0; j < lato \* dim; j++) {

B[x][el] = Bblock[0][(j \* dim) + x];

el++;

}

}

Now we can multiply by calling the matrix\_mult function.

void matrix\_mult(double\*\* A, double\*\* B, double\*\* C, int r, int c, int load) {

int i, j, k, l;

for (i = 0; i < r; i++) {

l = 0;

for (j = 0; j < r; j++) {

for (k = 0; k < c; k++) {

if (load == 0)

C[i][j] += A[i][k] \* B[l][k];

else

C[i][j] += heavy(A[i][k], load) \* B[l][k];

}

l++;

}

}

}

### Communication (from slave)

Block is now complete, so slave process can send it to the master.

double \*C\_vett = matrix\_vectorizer(dim, dim, C);

MPI\_Send(C\_vett, numElements, MPI\_DOUBLE, master, TAG, MPI\_COMM\_WORLD);

### Result matrix computation

Similarly to the cannon algorithm, the master can compute the resulting matrix by collecting all the pieces from every process.

for (i = 0; i < nblock; i++) {

MPI\_Recv(tempC[offset], numElements, MPI\_DOUBLE, i, TAG, MPI\_COMM\_WORLD, …);

offset++;

}

double \*C\_vett = matrix\_vectorizer(nblock, numElements, tempC);

block\_matrix(C, C\_vett, nblock, N);

Notice that function block\_matrix is needed to transform the final matrix, stored in a 1D array, into a “real” 2D matrix.

## Matrix Matrix Fast multiplication

This is a simplified version of the classical matrix matrix multiplication algorithm. We decided to introduce it in order to have a reference version similar to the classical multiplication among matrices. It's erroneously named "fast" because in the first version it performed better than the original Matrix-Matrix multiplication algorithm.

### Communication (from master)

Data are sent from the master with the function matrix\_sender. This function behaviour is the real difference with the Matrix Matrix multiplication algorithm: instead of demanding communications of blocks to slaves processes, it's the task of the master to send directly to slaves the complete rows and columns they need. Matrix\_sender function is thus the following:

void master\_sender(double\*\* A, double\*\* B, int offset, int n) {

int i, j, worker = 0;

double \* tempA, \* tempB;

for (j = 0; j < n; j += offset)

for (i = 0; i < n; i += offset) {

worker++; /\*increment worker number\*/

tempA = matrix\_vectorizer(offset, n, &A[j]);

tempB = matrix\_vectorizer(offset, n, &B[i]);

/\*MPI send of rows and cols. Tag 0 rows, tag 1 cols\*/

MPI\_Send(tempA, offset \* n, MPI\_DOUBLE, worker, 0, …);

MPI\_Send(tempB, offset \* n, MPI\_DOUBLE, worker, 1, …);

free(tempA);

free(tempB);

}

}

The two for loops allows to cover all the processes and to send them the needed rows and columns in the form of 1D arrays.

### Multiplication

After having received all the data needed and having transformed again vectors into matrices, a slave can compute the multiplication.

double\*\* mult(double\*\* rows, double\*\* cols, int n, int offset, int load) {

int i, j, k;

double\*\* res; /\*data structure for resulting matrix\*/

res = matrix\_creator(offset, offset);

for (i = 0; i < offset; i++) {

for (j = 0; j < offset; j++) {

res[i][j] = 0;

for (k = 0; k < n; k++) {

res[i][j] += rows[i][k] \* cols[j][k];

if (load == 0)

res[i][j] += rows[i][k] \* cols[j][k];

else

res[i][j] += heavy(rows[i][k], load) \* cols[j][k];

}

}

}

return res;

}

### Communication (from slaves)

At the end of the multiplication process slave send to the master their block packed in a 1D vector.

res\_vect = matrix\_vectorizer(offset, offset, res);

MPI\_Send(res\_vect, offset \* offset, MPI\_DOUBLE, 0, 2, MPI\_COMM\_WORLD);

### Result matrix collection

The master collect blocks incoming from slaves by using function master\_receiver.

double\*\* master\_receiver(int n, int offset) {

int i, j, x, y, worker = 0;

double\*\* res, \*\* res\_temp;

res = matrix\_creator(n, n);

for (j = 0; j < n; j += offset)

for (i = 0; i < n; i += offset) {

worker++;

double \* res\_vect = (double \*) malloc(offset \* offset \* sizeof (double));

MPI\_Recv(res\_vect, offset \* offset, …, worker, 2, …);

/\*transform the received vector in a matrix\*/

res\_temp = devectorizer(offset, offset, res\_vect);

free(res\_vect);

/\*computes the corresponding piece of the final matrix\*/

for (x = 0; x < offset; x++)

for (y = 0; y < offset; y++)

res[i + x][y + j] = res\_temp[x][y];

freematrix(offset, res\_temp);

}

return res;

}

# Testing Phase

## Environment Description

## Algorithms times

### Heavy function

Note that with a number of processes equal to 9, and a matrix size of 768x768, increasing the load given by the function, it grows in a linear way the time taken by various algorithms.

The results of the graph are obtained by keeping the communication time unchanged, and increasing in a linear manner the computation time in each process through the load function.

### No-Input/Output Vs Input/Output

The results of the following graphs are obtained by keeping unchanged the normal size of the matrix to 768x768. With these graphs we want to see how they change the running times of the various algorithms in a situation equal for all, using the input output from the disk, or by generating the matrices on the fly.

As you can see below, the execution time using the disk load increases, except in the case of the algorithm Processor Farm where it remains almost unchanged.

### No-Optimization Vs Optimization

### Number of process Vs Time

Matrix 192x192

As can be seen from the first graph, with matrices of small size and therefore with very limited computational load, it is possible to note that passing from 4 to 9 to 16 processes the execution time tends to increase, especially in the algorithm Processor Farm.

This anomalous behavior is due to the greater number of messages that the algorithms must perform as the number of processes; This is most noticeable in the Processor Farm because of the deployment of the entire matrix B to each slave process, thus creating a process whereby the master resulting in performance degradation.

The situation improves with the activation of the load, in fact, the computational cost increases with respect to communication costs, and then we take advantage of running with a greater number of processes.

Matrix 960x960

In this situation we can note that even if the load function is not activated, thanks to the increased computational cost due to the increase of the size of the matrix, the execution time decreases as the number of processes.

The performance improvement is due to the smaller amount of data processed by each process as they are available to multiple processes.

The greater improvement Processor Farm can be justified by the greater number of rows that are processed in parallel.

The Processor Farm still cannot reach the performance of other algorithms because communications are still more expensive than the computational costs.

We note in fact that by enabling the load to the maximum, we have a situation in which the Processor Farm aligns with the performance of other algorithms.

### Matrix dimension Vs Time

With 4 processes and with a function of load off, as the size of the array the execution time of the four algorithms increases.

This increase is due to the size of the sub-blocks sent, in the algorithms Matrix Matrix, MM-fast and Cannon in the Processor Farm is increased while the size of data sent (since the size of the rows is greater), and the number of communications necessary (increase number of rows).

In Farm Processor performance is worse due to the high number of communications executed.

Matrix Matrix algorithms and Cannon have performance, in this case, very similar because the number of communications carried out, and the size of the data sent are equivalent.

The performance difference between the algorithm Processor Farm and the other two increases with the growth of the size of the matrix; this happens because the cost of computation is low and therefore the communication time tends to "drown out" the latter.

Note that with the processes 16, the gap between the execution time of the Processor Farm and decreases that of the other algorithms given that the algorithms are executed on a larger number of processes and thus increase the number of communications to be performed in the algorithms Matrix Matrix and Cannon.

With load function activated (funct2), is increased the time required for computation, and this means that the performance of the Processor Farm is aligned to those of other algorithms.

By the transition from a situation without load function to a situation with load function active, the cost of communications is unchanged, but the calculation time increases considerably. In this situation, the algorithm Processor Farm takes advantage, because the product vector to matrix has lower complexity compared to the product matrix to matrix.

### SpeedUp

In this section we will analyze the results concerning the speedup; This value is useful to have a measure for the improvement in performance that is achieved using the parallel algorithm with multiple processes, compared to the algorithm running on a single process.   
The following graphs will be compared curves speedup of the four algorithms, while keeping constant the size of the matrix and the load function.

In the case of relatively simple problems, as we see in the graph (192x192 matrix), all algorithms have a negative trend in the value of speedup.

It appears evident that the communication time affects significantly compared to the computation time. In fact, as the number of processes, and then the communication times, the speedup decreases.

This test, however, is not indicative of the general trend of the value of speedup, because the problem is very simple, so as to render unnecessary the parallelization.

With this large matrix and load function (funct2) activated we can have a high computational, therefore benefiting all parallel algorithms. In fact, as we can see in the figure in the case where the processes vary from 4 to 9, we can have a linear increase of speedup, while in step 9 to 16 processes the speedup tends to decrease and to remain almost unchanged, except in the case of the algorithm processor Farm which still retains some growth.

This decrease was caused by the fact that always increases the number of processes increases the cost of communication and reduces the cost of the calculation and therefore the overhead of communication become the largest fraction of the total time.

The algorithm maintains a Processor Farm growth speedup because the number of communications with increasing number of processes, remains unchanged, but the single process performs an overall number of computations lower.

### Efficiency

The efficiency as a measure of the fraction of the total time during which the various processors are effectively used to perform the computation.

An efficiency equal to 1 (Sp(n) = n) means that all the processors participating in a profitable way the execution of the algorithm, an efficiency much lower than 1 testifies several factors such as:

* Unbalanced load;
* Idle processors;
* Extra work in the parallel version of the algorithm;
* Overhead related to communication / synchronization.

In this case we see how the Cannon Algorithm and the MM have a good efficiency value in the case where the processes involved are few, but then these values ​​decrease rapidly with the increase of processes and this leads to more efficiently evaluate the algorithm processor Farm, because it tends to decrease more slowly the efficiency, especially in costly problems and the increase of the processes involved.

This is due to the "natural" load balancing performed by that algorithm, and also to the fact that the number of communications, on the contrary to what happens in Matrix-Matrix and Cannon, is not influenced by the number of processes involved, but only depends on the size of the array.

We note that in the optimized version will reach a value of efficiency even greater than one; this is due to the fact that the sequential algorithm used, the algorithm is not optimal, but it is the simple multiplication of matrices . The sequential algorithm great fact is the algorithm of Strassen , with which it would not have been possible to obtain efficiency values ​​greater than 1.

### Overhead

As you can see from the graphs below, the increase of the size of the problem and the number of the processes involved, the overhead grows very quickly, to the detriment of efficiency as we have seen from the graphs above.

### Load balance

# Conclusion

# Appendix I: MPI primitives

The following is a brief overview of the MPI functions used in the implementation phase, to allow communication between the various nodes involved in the parallel computation. The overview consists of the signature of the function and description of the parameters in input and output, to make it easier to understand the implementation of the algorithms.

## MPI\_Send

int MPI\_Send(const void \*buf, int count, MPI\_Datatype datatype, int dest,

   int tag, MPI\_Comm comm)

MPI\_Send performs a standard-mode, blocking send. This routine will block until the message is sent to the destination.

Input Parameters

**buf:** Initial address of send buffer (choice).

**count:** Number of elements send (nonnegative integer).

**datatype:** Datatype of each send buffer element (handle).

**dest:** Rank of destination (integer).

**tag:** Message tag (integer).

**comm:** Communicator (handle).

## MPI\_Recv

int MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype,

   int source, int tag, MPI\_Comm comm, MPI\_Status \*status)

MPI\_Recv performs a standard-mode blocking receive. This basic receive operation, MPI\_Recv, is blocking: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

The receive buffer contains a number (defined by the value of count) of consecutive elements. The first element in the set of elements is located at address\_buf. The type of each of these elements is specified by datatype. The length of the received message must be less than or equal to the length of the receive buffer.

Input Parameters

**count:** Maximum number of elements to receive (integer).

**datatype:** Datatype of each receive buffer entry (handle).

**source:** Rank of source (integer).

**tag:** Message tag (integer).

**comm:** Communicator (handle).

Output Parameters

**buf:** Initial address of receive buffer (choice).

**status:** Status object (status).

## MPI\_Sendrecv\_replace

int MPI\_Sendrecv\_replace(void \*buf, int count, MPI\_Datatype datatype,

   int dest, int sendtag, int source, int recvtag, MPI\_Comm comm,

   MPI\_Status \*status)

MPI\_Sendrecv\_replace sends and receives a message using a single buffer. The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive; odd processes receive first, then send) in order to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

MPI\_Sendrecv\_replace executes a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.

Input/Output Parameter

**buf:** Initial address of send and receive buffer (choice).

Input Parameters

**count:** Number of elements in send and receive buffer (integer).

**datatype:** Type of elements to send and receive (handle).

**dest:** Rank of destination (integer).

**sendtag:** Send message tag (integer).

**source:** Rank of source (integer).

**recvtag:** Receive message tag (integer).

**comm:** Communicator (handle).

Output Parameters

**status:** Status object (status).

# Appendix II: User Manual

# Bibliography

Orlando, S. (2013). HPC Course slides

Open MPI Project (2004-2014). Open MPI v1.4.5 documentation