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 16 maggio 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 3](#_Toc387138256)

[2. Implementation 4](#_Toc387138257)

[2.1 Header files 4](#_Toc387138258)

[2.1.1 Stopwatch.h 4](#_Toc387138259)

[2.1.2 Header.h 4](#_Toc387138260)

[2.1.3 InOut.h 5](#_Toc387138261)

[2.2 Matrix Matrix multiplication 5](#_Toc387138262)

[2.3 Cannon Algorithm 5](#_Toc387138263)

[2.4 Processor Farm 5](#_Toc387138264)

[2.5 Matrix Matrix Fast multiplications 5](#_Toc387138265)

[3. Testing Phase 6](#_Toc387138266)

[3.1 Environment Description 6](#_Toc387138267)

[3.2 Algorithms times 6](#_Toc387138268)

[3.2.1 Matrix Matrix multiplication 9](#_Toc387138269)

[3.2.2 Cannon Algorithm 9](#_Toc387138270)

[3.2.3 Processor Farm 9](#_Toc387138271)

[3.2.4 Matrix Matrix Fast multiplications 9](#_Toc387138272)

[4. Conclusion 9](#_Toc387138273)

[5. Appendix I: MPI primitives 10](#_Toc387138274)

[5.1 MPI\_Send 10](#_Toc387138275)

[5.2 MPI\_Recv 10](#_Toc387138276)

[5.3 MPI\_Sendrecv\_replace 11](#_Toc387138277)

[6. Appendix II: User Manual 12](#_Toc387138278)

[7. Bibliography 12](#_Toc387138279)

# 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.

# 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.

## Matrix Matrix multiplication

## Cannon Algorithm

## Processor Farm

## Matrix Matrix Fast multiplications

# Testing Phase

## Environment Description

## Algorithms times

### Heavy function

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

### No-Optimization Vs Optimization

### Number of process Vs Time

### Matrix Matrix multiplication

### Cannon Algorithm

### Processor Farm

### Matrix Matrix Fast multiplications

# 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