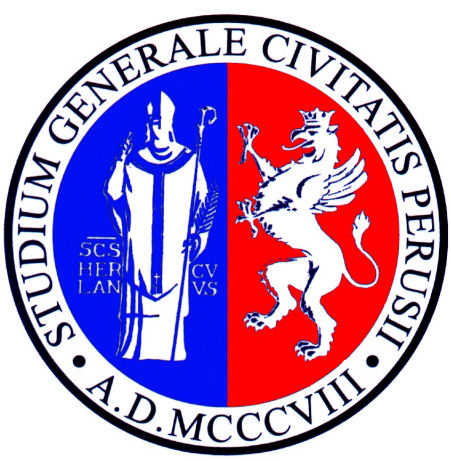
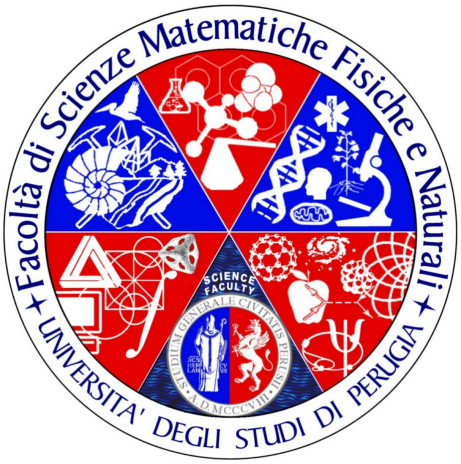
UNIVERSITÀ DEGLI STUDI DI PERUGIA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di laurea specialistica in Informatica





Corso di:
Applicazione e Calcolo in Rete Avanzato

Progetto di esame:
MM MPI
(Applicazione per la moltiplicazione di matrici su cluster usando MPI)

Studente:
Diego Russo - 231423
me@diegor.it

Professori:
Antonio Laganà
Leonardo Pacifici

Anno Accademico 2015/2016

Revision: 1f3dc84

Indice

Elenco delle figure

Sommario

La moltiplicazioni tra matrici è un problema che che si presenta spesso in varie applicazioni. Esempi pratici sono elaborazioni delle immagini, crittografia, applicazioni matematici ed ingegneristiche. Saper effettuare la moltiplicazioni tra matrici in maniera efficiente è dunque molto importante. Questo elaborato mostra un'approccio di come la moltiplicazione può essere parallelizzata utilizzando l'algoritormo di Cannon ed la libreria MPI.

1 Introduzione

1.1 La moltiplicazioni tra matrici

Per moltiplicazioni tra matrici si intende il prodotto tra righe e colonne tra due matrici. Questo prodotto, possibile sotto certe condizioni, da luogo ad un'altra matrice.

Esistono vari tipi di matrici: vettori, quadrate, rettangolari. Ovviamente si possono moltiplicare questi diversi tipi tra di loro a patto che sussistano delle condizioni sulle loro dimensioni.

Si può eseguire il prodotto tra A e B a patto che il numero delle colonne della matrice A (prima matrice) sia uguale al numero delle righe di B (seconda matrice).

Il prodotto AB avrà il numero di righe di A (prima matrice) ed il numero di colonne di B (seconda matrice).

Dunque il prodotto tra due matrici quadrate delle stesse dimensioni è sempre possibile.

Esempi:

A[4 x 5] x B[4 x 2] = C[4 x 2]

A[4 x 4] x B[4 x 6] = C[4 x 6]

A[4 x 4] x B[4 x 4] = C[4 x 4]

Il funzionamento del prodotto tra matrici è spiegato dalla figura 1.1.

Il diagramma mostra il caso in cui A è 4 x 2 e B è 2 x 3, e si voglia calcolare l'elemento $(C)_{12} = (AB)_{12}$ (figura 1.2) della matrice prodotto C = A x B, di dimensioni 4 x 3.

Più generalmente possiamo esprimere la moltiplicazioni di due matrici con la figura 1.3 dove ogni elemento della matrice C è calcolato utilizzando la formula in figura 1.4.

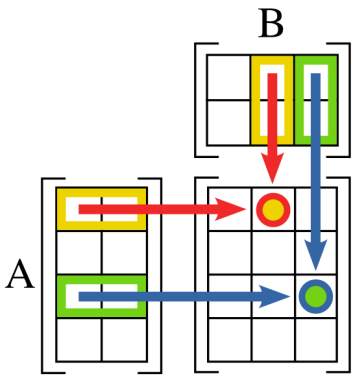


Figura 1.1: Diagramma della moltiplicazione

$$(C)_{12} = \sum_{r=1}^{2} a_{1r} b_{r2} = a_{11}b_{12} + a_{12}b_{22}$$

Figura 1.2: Formula per calcolare $(C)_{12}$

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\lambda + c\rho & a\beta + b\mu + c\sigma & a\gamma + b\nu + c\tau \\ p\alpha + q\lambda + r\rho & p\beta + q\mu + r\sigma & p\gamma + q\nu + r\tau \\ u\alpha + v\lambda + w\rho & u\beta + v\mu + w\sigma & u\gamma + v\nu + w\tau \end{pmatrix},$$

Figura 1.3: Esempio di prodotto tra matrici

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

Figura 1.4: Formula generica per calcolare c_{ij}

1.2 Parallelizzazione

La moltiplicazioni tra matrici appartiene a quella classe di problemi che possono essere parallelizzati o ottimizzati. Esistono molte ottimizzazioni come ad esempio: Strassen, Coppersmith-Winograd, etc. . . (vedi figura 1.5)

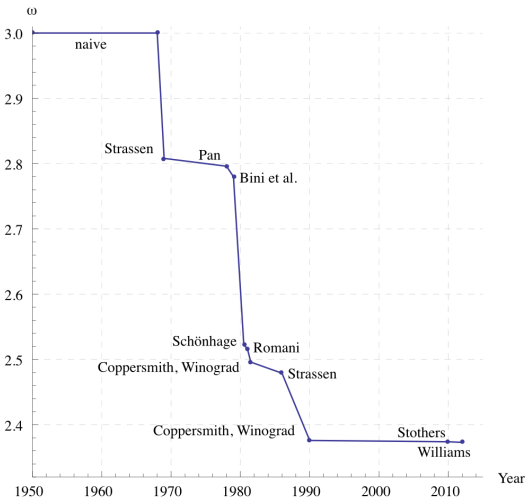


Figura 1.5: Ottimizzazioni di algoritmi di moltiplicazioni tra matrici nel tempo

Per quanto riguarda la parallelizzazione ci sono algoritmi che sfruttano memoria condivisa mentre altri che sono distribuiti decomponendo il problema in sottoproblemi da eseguire localmente. Questo tipo di approccio porta con se diverse problematiche come overhead di comunicazione, decomposizione del problema e scambio di informazioni tra i vari nodi. Senza scendere troppo nei dettagli dei vari algoritmi distribuiti (Johnson's algorithm, Ballard and Demmel), possiamo trovare anche l'algoritmo di Cannon spiegato nel prossimo capitolo ed utilizzato per lo sviluppo dell'applicazione.

2 Cannon

2.1 Algoritmo

Si consideri la moltiplicazione tra due matrici quadrate di dimensione n, C = AxB.

Le matrici vengono partizionate in p blocchi quadrati, dove p è il numero di processi da utilizzare per eseguire l'algoritmo. In questo modo ogni processi ha $n/\sqrt{p}$ x $n/\sqrt{p}$ pezzo di blocco.

I processi vanno da $P_{(0,0)}$ a $P_{(\sqrt{p}-1,\sqrt{p}-1)}$ ed inizialmente ogni sottoblocco A_{ij} e B_{ij} vengono associati a P_{ij} e sono organizzati in righe e colonne. I dati poi sono inviati in maniera incrementale in $\sqrt{p}$ passi al blocco superiore o sinitro utilizzando un sistema ad anello.

Prima di iniziare la moltiplicazione vera e propria, c'è bisogno di un allineamento iniziale delle matrici A e B in modo che ogni processo possa moltiplicare i propri blocchi in maniera indipendente. Dunque si applica uno shift a sinistra con wraparound di i step a tutti i sottoblocchi A_{ij} della matrice A. Sulla matrice B invece si applica uno shift in alto di j step su tutti i sottoblocchi B_{ij}.

Dopo la fase di allineamento iniziale, l'algoritmo può eseguire la prima moltiplicazione tra i vari sottoblocchi per poi muovere ancora i vari blocchi della matrice A a sinistra e della matrice B in alto seguendo lo stesso meccanismo di prima. Si continua così finchè non si sono eseguite tutte le $\sqrt{p}$ moltiplicazioni.

Per quanto riguarda il costo dell'algoritmo possiamo suddividerlo nei seguenti blocchi:

- nello step di allineamento iniziale, un blocco shifta al massimo di $\sqrt{p}-1$. L'operazione di shift per ogni colonna e colonna richiede:

  $t_{comm} = 2(t_s + \frac{t_w n^2}{p})$

- Ogni $\sqrt{p}$ shift richiede:

  $t_s + \frac{t_w n^2}{p}$

- La moltiplicazione di $\sqrt{p}$ matrici di grandezza $(\frac{n}{\sqrt{p}})$ x $(\frac{n}{\sqrt{p}})$ richiede $\frac{n^3}{p}$

Sommando i tempi precedenti si ha che il tempo parallelo per l'algoritmo di Cannon è:
$T_p = \frac{n^3}{p} + 2\sqrt{p}(t_s + \frac{t_w n^2}{p}) + 2(t_s + \frac{t_w n^2}{p})$
dove indichiamo con t_s = startup-time e con t_w = pre-word transfer time.

2.2 Esempio

Per capire meglio il funzionamento dell'algoritmo, è necessario un esempio che mostri i vari passi. Si voglia moltiplicare due matrici A[3x3] x B[3x3] mostrate nella figura 2.1. Si consideri l'iterazione i=1, j=2. Dunque $C_{1,2}$ viene calcolato con la seguente formula:
C[1,2] = A[1,0] * B[0,2] + A[1,1] * B[1,2] + A[1,2] * B[2,2]

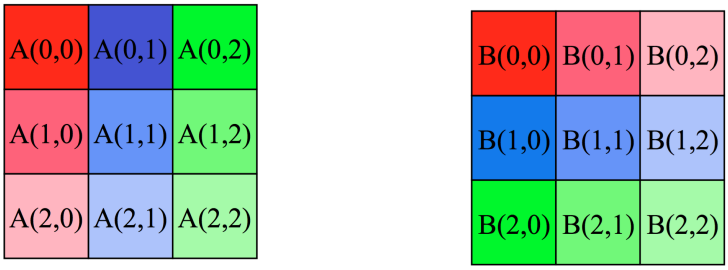


Figura 2.1: Cannon: matrici iniziali

Vogliamo che A[1,0] e B[0,2] siano sullo stesso processo. Dunque sulla matrice A si fa uno shift delle righe (figura 2.2) e sulla matrice B si fa uno shift delle colonne (figura 2.3).

In questo modo si nota che la prossima coppia A[1,1] e B[1,2] sono pronti per essere spostati ed andare a finire dove A[1,0] e B[0,2] sono ora. La stessa cose si ripete per A[1,2] e B[2,2].

Appena effettuata la moltiplicazione possiamo effettuare un altro shift su tutte e due le matrici avendo le matrici come in figura 2.4. Come su può vedere, A[1,1] e B[1,2] sono allineati per essere moltiplicati.

Si continua a shiftare in modo da avere A[1,2] e B[2,2] in posizione per effettuare la moltiplicazione.

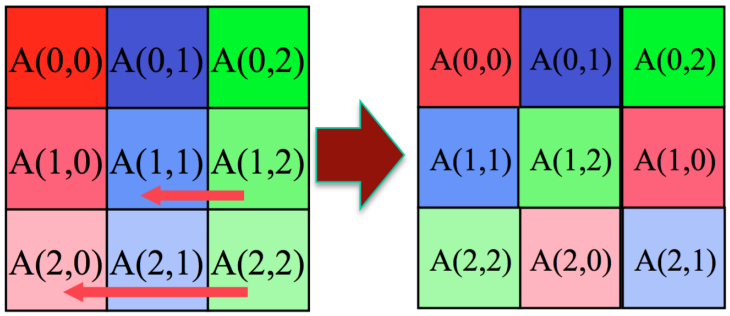


Figura 2.2: Cannon: shift iniziale sulla matrice A

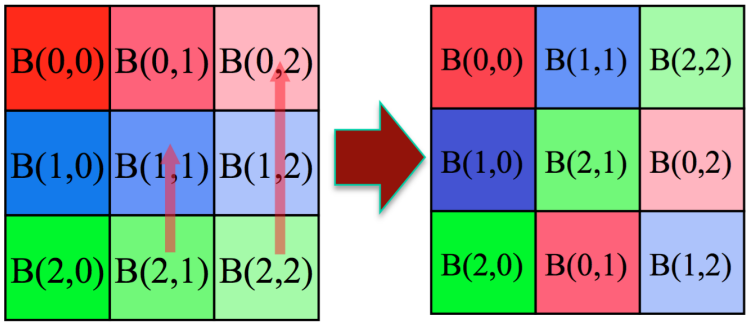


Figura 2.3: Cannon: shift iniziale sulla matrice B

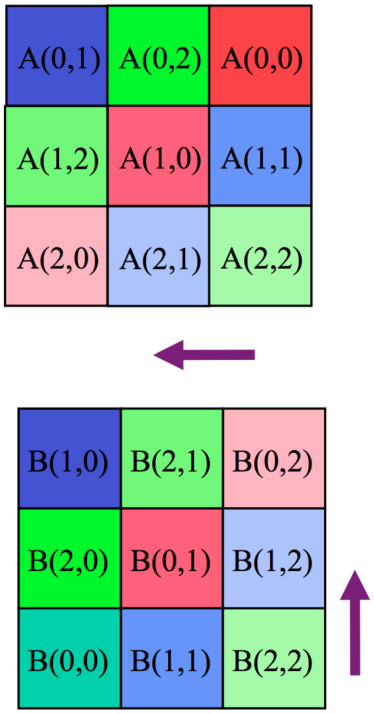


Figura 2.4: Cannon: shift sulle matrici A e B

3 MM MPI

In questo capitolo viene analizzato tutto quello che riguarda l'implementazione di MM MPI. La struttura del codice, le primitive MPI utilizzate, compilazione, esecuzioni, output dell'applicazione ed infine delle ottimizzazione per migliorare le prestazioni di MM MPI.

3.1 Il codice dell'applicazione

L'applicazione è scritta interamente in C ed utilizza la libreria mvapich2. Di seguito la lista dei file.

```
contributors.txt
data
doc
logs
Makefile
Makefile.mm.dev
Makefile.mm.inc -> Makefile.mm.pg
Makefile.mm.pg
README
run_mm_mpi_dev.sh
run_mm_mpi_pg.sh
src
test.sh
```

La directory *data* contiene i file di input da passare all'applicazione.

La directory *doc* contiene i sorgenti di questa relazione

La directory *src* contiene il codice dell'applicazione con i vari Makefile. Sia il codice che i Makefile verrano spiegati successivamente.

Nella root dell'applicazione ci sono poi vari script per l'esecuzione ed cosa molto importante il README con istruzioni su come compilare ed eseguire.

3.1.1 Makefiles

I Makefiles giocano un ruolo molto importante per la compilazione dell'applicazione. Di seguito il Makefile principale che è responsabile della compilazione delle varie versioni dell'applicazione.

```
default: all

## Serial version
serial:
  cd src/serial && make && cp x.mm ../../x.mm_serial;

## Cannon - blocking version
2D-cannon:
  cd src/cannon && make -f Makefile && cp x.mm ../../x.mm_2D_cannon;

# Optimizations
2D-cannon-openmp-outer:
  cd src/cannon && make -f Makefile.openmp.outer && cp x.mm ../../x.
    mm_2D_cannon_openmp_outer;

2D-cannon-openmp-middle:
  cd src/cannon && make -f Makefile.openmp.middle && cp x.mm ../../x.
    mm_2D_cannon_openmp_middle;

2D-cannon-openmp-inner:
  cd src/cannon && make -f Makefile.openmp.inner && cp x.mm ../../x.
    mm_2D_cannon_openmp_inner;

2D-cannon-openmp-nested:
  cd src/cannon && make -f Makefile.openmp.nested && cp x.mm ../../x.
    mm_2D_cannon_openmp_nested;

2D-cannon-cblas:
  cd src/cannon && make -f Makefile.cblas && cp x.mm ../../x.
    mm_2D_cannon_cblas;

## Cannon - non blocking version
2D-cannon-nonblock:
  cd src/cannon && make -f Makefile.nonblock && cp x.mm ../../x.
    mm_2D_cannon_nonblock;

# Optimizations
2D-cannon-nonblock-openmp-outer:
  cd src/cannon && make -f Makefile.nonblock.openmp.outer && cp x.mm ../../x
    .mm_2D_cannon_nonblock_openmp_outer;

2D-cannon-nonblock-openmp-middle:
  cd src/cannon && make -f Makefile.nonblock.openmp.middle && cp x.mm ../../
    x.mm_2D_cannon_nonblock_openmp_middle;

2D-cannon-nonblock-openmp-inner:
  cd src/cannon && make -f Makefile.nonblock.openmp.inner && cp x.mm ../../x
    .mm_2D_cannon_nonblock_openmp_inner;

2D-cannon-nonblock-openmp-nested:
  cd src/cannon && make -f Makefile.nonblock.openmp.nested && cp x.mm ../../
    x.mm_2D_cannon_nonblock_openmp_nested;

2D-cannon-nonblock-cblas:
```

```
45   cd src/cannon && make -f Makefile.nonblock.cblas && cp x.mm ../../x.
       mm_2D_cannon_nonblock_cblas;
46
47 all: serial \
48    2D-cannon 2D-cannon-openmp-outer 2D-cannon-openmp-middle 2D-cannon-openmp
       -inner 2D-cannon-openmp-nested 2D-cannon-cblas \
49    2D-cannon-nonblock 2D-cannon-nonblock-openmp-outer 2D-cannon-nonblock-
       openmp-middle 2D-cannon-nonblock-openmp-inner \
50     2D-cannon-nonblock-openmp-nested 2D-cannon-nonblock-cblas
51
52 clean:
53   cd src/serial && make clean;
54   cd src/cannon && make -f Makefile clean;
55   cd src/cannon && make -f Makefile.nonblock clean;
56   rm ./x.mm_*
```

Ci sono poi dei Makefile secondari, uno per ogni versione dell'applicazione da compilare. Un esempio è il seguente:

```
1  include ../../Makefile.mm.inc
2
3  VPATH = ../shared
4
5  OBJECTS = mm.o check.o gendat.o mxm.o mxm-local.o \
6            format.o reset.o utils.o
7
8  EXEC    = x.mm
9
10 ${EXEC}: clean ${OBJECTS}
11   @echo
12   ${LD_MPI} ${LDFLAGS} ${OBJECTS} ${LIBS} -o ${EXEC}
13
14 .SUFFIXES:  .c .o
15 .c.o    :
16   @echo
17   ${CC_MPI} ${CFLAGS} -DCBLAS ${INCS} -c -o $@ $<
18
19 clean:
20   /bin/rm -f ${OBJECTS} ${EXEC}
```

che compila la versione cblas di MM MPI. Nella riga 1 c'è l'inclusione di un altro file. Questo file contiene tutti i parametri per la corretta compilazione e differisce a seconda dell'ambiete di sviluppo. Per compilare ed eseguire MM MPI su un MBP i seguenti parametri sono stati utilizzati:

```
1  SELF_DIR := $(realpath .)
2
3  CC      = gcc-5
4  CC_MPI  = mpicc
5  INCS    = -I "$(SELF_DIR)/../shared/" -I /System/Library/Frameworks/
       Accelerate.framework/Versions/Current/Frameworks/vecLib.framework/
       Versions/Current/Headers/
6  CFLAGS  = -O3 -mcmodel=medium -fopenmp
7  LD      = $(CC)
8  LD_MPI  = $(CC_MPI)
9  LDFLAGS = $(CFLAGS) -lcblas
10 LIBS    = -L /System/Library/Frameworks/Accelerate.framework/Versions/
       Current/Frameworks/vecLib.framework/Versions/Current/ -pthread
```

Prima della compilazione dunque si deve creare il Makefile con i giusti parametri e poi creare un link simbolico *Makefile.mm.inc*

3.1.2 La directory src/

Questa directory contiene tre sotto directory dove si possono trovare i file .c e .h che implementano MM MPI. Il codice è ben documentato con commenti (in inglese), dunque lascio all'utente l'esercizio di aprire i file sorgenti mentre legge la seguente relazione.

src/shared

In *shared* si trovano funzioni che sono indipendenti dall'implementazione (seriale o parallela). Il contenuto è i seguente:

```
1 format.c
2 format.h
3 reset.c
4 reset.h
5 shared.h
6 utils.c
7 utils.h
```

format.c contiene funzioni accessorie per stampare i risultati sullo stdout e debug sullo stderr.

reset.c contiene una funzione per fare il reset di una matrice: tutte le celle della matrice verranno impostate a 0.0.

L'unico contenuto di *shared.h* è una variabile globale per controllare il debug dell'applicazione.

Infine *utils.c* ha un parser degli argomenti passati da console: help, debug e file di input. C'è anche funzione per prendere il timestamp, utile per calcolare poi il tempo trascorso per la computazione (utilizzato solo per l'implementazione seriale).

src/serial

serial contiene l'implementazione seriale della moltiplicazione. La lista dei file è:

```
1 check.c
2 gendat.c
3 Makefile
4 mm.c
5 mxm.c
```

mm.c contiene il main, dove tutto il flusso di lavoro viene guidato: dal parsing dei parametri passati da console, alla generazione dei dati per poi

passare alla moltiplicazione vera e propria per poi concludere al controllo dei risultati, calcolo del tempo passato e la stampa allo stdout dei risultati.

mxm.c contiente la vera moltiplicazione tra matrice: questa implementazione è la versione naive dell'algoritmo. Giusto per completezza, si riporta di seguito il codice:

```
void mxm(int m, int l, int n, double a[][l], double b[][n],
         double c[][n]) {
    int     i, j, k;

    for (i = 0; i < m; i++) {
        for (j = 0; j < l; j++) {
            for (k = 0; k < n; k++) {
                c[i][k] += a[i][j] * b[j][k];
            }
        }
    }
}
```

Si ricorda che la sua complessità è di $O(n^3)$.

Oltre al Makefile per guidare la compilazione, *check.c* contiene una funzione per verificare se la moltiplicazione è avvenuta con successo e che abbia prodotto la C con i giusti risultati. Infine *gendat.c* implementa una funzione per generare i dati delle matrici A e B.

Come ultima nota, nella sezione relativa all'esecuzione verrà spiegato il meccanismo che sta dietro alla generazione dei dati ed al relativo controllo del giusto risultato (che è uguale sia per l'implementazione seriale, sia per quella parallela).

src/cannon

La directory *cannon* ha molti più file rispetto alla corrispettiva seriale. Questo perchè sono implementate varie versione ed ottimizzazioni che verrano spiegate dopo.

```
check.c
gendat.c
Makefile
Makefile.cblas
Makefile.nonblock
Makefile.nonblock.cblas
Makefile.nonblock.openmp.inner
Makefile.nonblock.openmp.middle
Makefile.nonblock.openmp.nested
Makefile.nonblock.openmp.outer
Makefile.openmp.inner
Makefile.openmp.middle
Makefile.openmp.nested
Makefile.openmp.outer
mm.c
mxm.c
mxm-local.c
mxm-local.h
```

La lunga lista di Makefile serve a compilare queste diverse versioni dell'applicazione.

Come nella versione seriale troviamo sia *check.c*, sia *gendat.c* però la loro implementazione è leggermente diversa. Il controllo viene fatto su un array invece che su un array di array mentre la generazione dei dati avviene utilizzando MPI: ogni nodo è responsabile di generare il proprio blocco di dati.

mm.c ancora è il principale file dove il main guida tutto il flusso di lavoro: ci sono dei controlli preliminari da effettuare prima di prendere i dati dalla riga di comando. Il principale è controllare se il numero di processori che ho a disposizione sia un quadrato perfetto: se non lo fosse non potrei partizionare la matrice in blocchi per l'esecuzione parallela. Fatto ciò, si possono leggere i dati da console e iniziare a generare i dati delle matrici: qui cé un ulteriore controllo da fare. Si deve essere sicuri che le matrici siano "coperte" totalmente e senza scarti dal numero dei processori disponibili. Una volta passato questo controllo si prosegue alla generazione dei dati, moltiplicazione e controllo dei risultati: da notare che tutti questi passaggi sono effettuati tramite MPI. Se tutto dovesse andare bene, i risultato vengono diretti sullo stdout.

mxm.c ha un'unica funzione che é responsabile di effettuare la moltiplicazione parallela sfruttando MPI e l'algoritmo di Cannon. Infatti in questa funzione è possibile distinguere le varie fasi dello stesso algoritmo. Altra cosa da notare é la "doppia" implementazione: con la macro *#NONBLOCKING* si implementa un approccio non bloccante utilizzando primitive MPI non bloccanti ed un doppio buffer.

Infine *mxm-local.c* implementa la moltiplicazione (seriale) tra sottoblocchi di matrice: il codice infatti viene eseguito su ogni processo e si occupa di moltiplicare solo i dati locali, indipendentemente dal contenuto degli altri nodi. Questo file inoltre contiene alcune ottimizzazioni per migliorare le prestazioni dell'intero algoritmo: infatti grazie ad alcune macro, si hanno ottimizziazioni OpenMP e CBlas.

3.1.3 Primitive MPI utilizzate

Per comprendere meglio l'implementazione parallela, si ha qui una lista di primitive utilizzate nell'applicazione. Per ogni primitiva si descrive l'interfaccia e l'utilizzo all'interno del codice.

MPI_Init

```
1 int MPI_Init( int *argc, char ***argv )
```

- argc: puntatore al numero di argomenti

- argv: puntatore al vettore di argomenti

MPI_Init deve essere la prima istruzione ad essere chiamata prima di qualsiasi altra istruzione MPI. Si trova nel file *mm.c* subito dopo le dichiarazioni di variabili. l'API è:

MPI_Comm_size

```
1 int MPI_Comm_size( MPI_Comm comm, int *size )
```

- comm: comunicatore MPI
- size: numero di processi nel gruppo del comunicatore (output)

MPI_Comm_size determina la grandezza del gruppo associato al comunicatore. Si trova su 3 file: gendat.c, mm.c e mxm.c. Questo perchè tutti e tre i file hanno a che fare con operazioni MPI.

MPI_Comm_rank

```
1 int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

- comm: comunicatore MPI
- rank: rank del processo chiamante nel gruppo del comunicatore (output)

Determina il rank del processo chiamante nel comunicatore. Come nel caso precedente, si trova su 3 file: gendat.c, mxm.c e mm.c. Sui primi due file è replicato due volte perchè vengono utilizzati due diversi comunicatori: quello generico (MPI_COMM_WORLD) ed un altro comunicatore con una topologia cartesiana, utilizzato per implementare l'algoritmo di Cannon.

MPI_Barrier

```
1 int MPI_Barrier( MPI_Comm comm )
```

- comm: comunicatore

Blocca finchè tutti i processi nel comunicatore hanno raggiunto questa routine. Si trova solo sul file mm ed è utilizzato per sincronizzare i vari step dell'applicazione.

MPI_Wtime

```
double MPI_Wtime( void )
```

Ritorna il tempo trascorso del processo chiamante da un punto arbitrario nel passato. È utilizzato in mm.c per prendere il tempo prima e dopo la moltiplicazione parallela per poi calcolare il tempo trascorso.

MPI_Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, int root, MPI_Comm comm)
```

- sendbuf: indirizzo del buffer da inviare
- recvbuf: indirizzo del buffer di ricezione, rilevante solo al processo root (output)
- count: numero di elementi in sendbuffer
- datatype: tipo di dati in sendbuf
- op: operazione di riduzione
- root: rank del processo root
- comm: comunicatore

La funzione è presente in mm.c e serve a ridurre gli output della funzione check() distribuiti tra i nodi ad un solo valore. La funzione è così chiamata:

```
MPI_Reduce(&local_check,   // Indirizzo del buffer da inviare
           &ok,            // Indirizzo del buffer ricevente
           1,              // Numero di elementi
           MPI_INTEGER,    // Tipologia di elementi
           MPI_SUM,        // L'operazione da effettuare (somma)
           0,              // Rank del processo root
           MPI_COMM_WORLD);// Comunicatore
```

Si ricorda che se check() ritorna 0, il check è andato a buon fine. Dunque si sommano tutti i check distribuiti e si memorizzano in "ok". Questa variabile verrà controllata in seguito per verificare se la moltiplicazione ha avuto successo.

MPI_Cart_create

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const
    int periods[], int reorder, MPI_Comm *comm_cart)
```

- comm_old: comunicatore in input

- ndims: numero di dimensioni della griglia cartesiana
- dims: array di ndims interi per specificare il numero di processi in ogni dimensione
- periods: array di ndims valori per specificare se la griglia è peiodica (true) o meno (false) in ogni dimensione
- reorder: il ranking può essere riordinato o meno
- comm_cart: comunicatore con la nuova topologia cartesiana

La funzione è utilizzata per creare un comunicatore con topologia cartesiana ed è presente sia in gendat.c che in mxm.c. L'unica differenza tra le due chiamate è che in gendat.c non si ha bisogno della periodicità della griglia mentre in mxm.c si: questo perchè in mxm.c la griglia deve essere shiftata con wraparound, proprio come l'algoritmo di Cannon.

MPI_Cart_coords

```
1 int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

- comm: comunicatore
- rank: rank del processo nel gruppo di comm
- maxdims: lunghezza del vettore coords
- coords: array di interi (lunghezza di ndims) contenente le coordinate cartesiane del processo specificato (output)

Determina le coordinate del process nella topolgia cartesiana dato uno specifico rank nel gruppo. È utilizzata sia in gendat.c, sia in mxm.c: le coordinate sono fondamentali per capire la posizione virtuale del blocco di dati nello spazio cartesiano.

MPI_Cart_shift

```
1 int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
      int *rank_dest)
```

- comm: comunicatore con struttura cartesiana
- direction: la coordinata della dimensione da shiftare
- disp: movimento (> 0: shift in alto, < 0 shift in basso)

- rank_source: rank del processo sorgente (output)
- rank_dest: rank del processo destinazione (output)

Ritorna i sorgenti e destinazioni shiftati data una direzione ed un numero di shift. La funzione è utilizzata solo in mxm.c, ovvero il file dove l'algoritmo di Cannon è implementato: necessito di questa primitiva per effettuare l'allineamento iniziale e finale.

MPI_Sendrecv_replace

```
1 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)
```

- buf: indirizzo iniziale del buffer da inviare e per ricevere (output)
- count: numero di elementi nel buffer
- datatype: tipo di elementi nel buffer
- dest: rank di destinazione
- sendtag: tag per il messaggio da inviare
- source: rank sorgennte
- recvtag: tag per il messaggio da ricevere
- comm: comunicatore
- status: oggetto status (output)

La funzione invia e riceve dati utilizzando un singolo buffer in maniera bloccante. È presente solo in mxm.c ed invia/riceve blocchi di matrice ai/-dai blocchi adiacenti per implementare l'algoritmo di Cannon utilizzando il comunicatore con topologia cartesiana.

MPI_Isend

```
1 int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,
      int tag, MPI_Comm comm, MPI_Request *request)
```

- buf: indirizzo del buffer da inviare
- count: numero di elementi contenuti nel buffer

- datatype: tipo di dati contenuti nel buffer
- dest: rank di destinazione
- tag: tag del messaggio
- comm: comunicatore
- request: richiesta di comunicazione (output)

La funzione implementa un invio non bloccante: questo vuol dire che ritorna anche se il messaggio non è stato inviato. È utilizzata in mxm.c nella versione non bloccante utilizzando un doppio buffer.

MPI_Irecv

```
1 int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int
      tag, MPI_Comm comm, MPI_Request *request)
```

- buf: indirizzo del buffer da ricevere
- count: numero di elementi contenuti nel buffer
- datatype: tipo di dati contenuti nel buffer
- dest: rank sorgente
- tag: tag del messaggio
- comm: comunicatore
- request: richiesta di comunicazione (output)

La funzione implementa una ricezione non bloccante: questo vuole dire che ritorna anche sel il messaggio non è stato ricevuto. È utilizzato in mxm.c nella versione non bloccante utilizzando un doppio buffer.

MPI_Waitall

```
1 int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status
      array_of_statuses[])
```

- count: lunghezza della lista
- array_of_requests: array di richieste
- array_of_statuses: array di status (output)

La funzione aspetta che tutte le request passate abbiano terminato. La primitiva è utilizzata in mxm.c nella versione non bloccante e serve per bloccare l'esecuzione fino a che ogni gli shift a sinistra ed in alto abbiano finito.

MPI_Comm_free

```
int MPI_Comm_free(MPI_Comm *comm)
```

- comm: comunicatore

La primitiva marca il comunicatore pronto per essere deallocato. Questo significa che il comunicatore non è più necessario. Presente sia in gendat.c ed mxm.c per deallocare il comunicatore con topologia cartesiana.

MPI_Finalize

```
int MPI_Finalize( void )
```

Termina l'ambiente di esecuzione MPI ed è presente solo in mm.c, il file responsabile del setup dell'ambiente MPI. Dopo questa primitiva nessun comando MPI può essere chiamato a meno che un altro ambiente MPI sia configurato.

3.2 Compilazione

La compilazione di MM MPI è guidata da una serie di Makefile. Prima della compilazione si deve creare il file *Makefile.mm.inc* oppure creare un link simbolico ad uno dei file già presenti. Una volta che il file *Makefile.mm.inc* esiste, è possibile lanciare il comando *make*. Se non si passa nessun argomento, make compilerà tutte le possibili variazioni di MM MPI: una versione seriale e dodici versioni parallele. Se si vuole compilare solo la versione di Cannon non bloccante ottimizzata con cblas, basterà digitare:

```
$ make 2D-cannon-nonblock-cblas
```

I binari prodotti inizieranno con *x.mm_* seguiti dalla implementazione. La lista completa dei binari è:

```
$ ls x.mm_*
x.mm_2D_cannon
x.mm_2D_cannon_cblas
x.mm_2D_cannon_nonblock
x.mm_2D_cannon_nonblock_cblas
x.mm_2D_cannon_nonblock_openmp_inner
x.mm_2D_cannon_nonblock_openmp_middle
x.mm_2D_cannon_nonblock_openmp_nested
x.mm_2D_cannon_nonblock_openmp_outer
x.mm_2D_cannon_openmp_inner
```

```
x.mm_2D_cannon_openmp_middle
x.mm_2D_cannon_openmp_nested
x.mm_2D_cannon_openmp_outer
x.mm_serial
```

3.3 Esecuzione

Prima di spiegare come eseguire MM MPI, è dovereso spiegare come la generazione dei dati ed il controllo funzionano.

Data file

MM MPI prende in input un file di testo con il seguente formato

```
$ cat data/2_repititions
4 4 4 2
8 8 8 2
16 16 16 2
32 32 32 2
64 64 64 2
128 128 128 2
192 192 192 2
256 256 256 2
384 384 384 2
512 512 512 2
768 768 768 2
1024 1024 1024 2
1536 1536 1536 2
2048 2048 2048 2
3072 3072 3072 2
4096 4096 4096 2
```

Il formato del file è il seguente:

```
M   L   N   #repitizioni
```

dove la matrice A ha dimensioni MxL, la matrice B ha dimensioni LxN. In questo modo è sempre possibile effettuare la moltiplicazione. L'ultimo parametro indica il numero di volte che la moltiplicazione deve essere effettuata senza rigenerare i dati. Nell'esempio precedente vediamo che tutte le matrici generate sono quadrate di dimensione n e che la moltiplicazione verrà ripetuta 2 volte. La ripetizione è molto importante soprattutto quando si vogliono collezionare dati e si vuole attenuare eventuale rumore durante gli esperimenti.

Generazione delle matrici A e B

Le matrici A e B vengono generate come nella figura 3.1. Sia A sia B sono matrici quadrate di dimensioni 4.

La matrice A ha tutte le righe uguali ed ogni cella ha il valore dell'i-esima colonna +1. L'ultima colonna contiene sempre il valore di L nel file precedente.

La matrice B invece ha tutte le colonne uguali ed ogni cella contiene numero dell'i-esima riga + 1 che divide 1 ($\frac{1}{i+1}$). La prima riga ha sempre 1 mentre l'ultima riga contiene sempre $\frac{1}{N}$

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

A

1	1	1	1
0.5	0.5	0.5	0.5
0.33	0.33	0.33	0.33
0.25	0.25	0.25	0.25

B

Figura 3.1: Generazione delle matrici A e B

La generazione delle matrici parallele è un po' più complessa perchè le matrici sono rappresentate in un array e distribuite su diversi nodi. Per quanto riguarda la matrice A:

```
a[i * L_DBLOCK + j] = (double)((coordinates[1] * L_DBLOCK) + j + 1);
```

- i: valore nel M_DBLOCK
- j: valore nel L_DBLOCK
- L_DBLOCK: larghezza del blocco L
- coordinates[1]: valore della colonna nelle coordinate cartesiane

Per quanto riguarda la matrice B:

```
b[i * N_DBLOCK + j] = 1.0 / (double)((coordinates[0] * L_DBLOCK) + i + 1);
```

- i: valore nel L_DBLOCK
- j: valore nel N_DBLOCK
- N_DBLOCK: larghezza del blocco N
- coordinates[0]: valore della riga nelle coordinate cartesiane

Controllo della matrice finale

Le matrici così generate producono un risultato prevedibile come mostrato in figura 3.2. Il risultato oltre che prevedibile è anche facile da verificare. La matrice C infatti avrà in ogni cella il valore L.

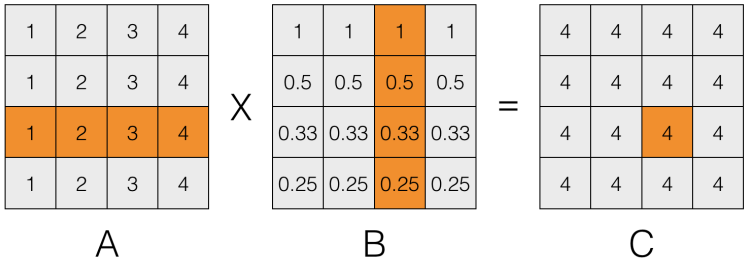


Figura 3.2: Moltiplicazione delle matrici A e B

Ora si hanno tutti gli elementi per eseguire l'applicazione.

Esecuzione seriale

La versione seriale è la più semplice da eseguire.

```
./x.mm_serial -f data/demo_data
```

Dove *demo_ data* è il file contenente le informazioni per effettuare la moltiplicazione.

Esecuzione parallela su singola macchina

Se si vuole testare la versione MPI senza avere a disposizione il cluster, lo si può fare attraverso *mpirun*

```
mpirun -n 16 ./x.mm_2D_cannon -f data/demo_data
```

In questo caso sto eseguendo la versione standard di Cannon utilizzando 16 processi.

Esecuzione parallela su cluster

L'esecuzione sul cluster è leggermente un pochino più complessa rispetto alle altre.

```
$ PROC_NAME=x.mm_2D_cannon
$ qsub -lnodes=8:ppn=2,mem=9050mb -d . -N $PROC_NAME -e logs -o logs -v EXE=
    $PROC_NAME,DATA=example_data run_mm_mpi_pg.sh
```

Il job è sottomesso utilizzando qsub specificando che si ha bisogno di 8 nodi con 2 processori e un utilizzo totale di 9GB di memoria.*-d* specifica l'area di lavoro, *-N* il nome del processo, *-e/-o* la directory dove i log verranno memorizzati ed infine *-v* specifica la lista di variabili da passare allo script *run_mm_mpi_pg.sh* che le utilizzerà per eseguire MM MPI.

```
#!/bin/sh

mpirun_rsh -np $PBS_NP -hostfile $PBS_NODEFILE ./$EXE -f $DATA
```

Notare che le variabili *$PBS_NODEFILE $PBS_NP* sono impostate in automatico da Torque.

3.4 Debug output

Il debug dell'applicazione puoò essere attivato attraverso l'opzione "-d". Il debug sarà differente a seconda della versione seriale e parallela: la versione parallela ha informazioni addizionali quali il rank del processo, le inforamzioni relative alla topologia cartesiana ed ai relativi shift.

```
[diego@cgcw mm_mpi]$ ./x.mm_serial -f data/demo_data  -d
Executing ./x.mm_serial
Generating data for matrix A[4 x 4]
[function: gendat] a[0][0] = 1
[function: gendat] a[0][1] = 2
[function: gendat] a[0][2] = 3
[function: gendat] a[0][3] = 4
[function: gendat] a[1][0] = 1
[function: gendat] a[1][1] = 2
[function: gendat] a[1][2] = 3
[function: gendat] a[1][3] = 4
[function: gendat] a[2][0] = 1
[function: gendat] a[2][1] = 2
[function: gendat] a[2][2] = 3
[function: gendat] a[2][3] = 4
[function: gendat] a[3][0] = 1
[function: gendat] a[3][1] = 2
[function: gendat] a[3][2] = 3
[function: gendat] a[3][3] = 4
Generating data for matrix B[4 x 4]
[function: gendat] b[0][0] = 1
[function: gendat] b[0][1] = 1
[function: gendat] b[0][2] = 1
[function: gendat] b[0][3] = 1
[function: gendat] b[1][0] = 0.5
[function: gendat] b[1][1] = 0.5
[function: gendat] b[1][2] = 0.5
[function: gendat] b[1][3] = 0.5
[function: gendat] b[2][0] = 0.333333
[function: gendat] b[2][1] = 0.333333
[function: gendat] b[2][2] = 0.333333
[function: gendat] b[2][3] = 0.333333
[function: gendat] b[3][0] = 0.25
[function: gendat] b[3][1] = 0.25
[function: gendat] b[3][2] = 0.25
```

```
[function: gendat] b[3][3] = 0.25
Matrix C will be [4 x 4]
Let's do the math (1 repetitions)
mm: Matrix-matrix multiply test C(m,n) = A(m,l)*B(l,n)
-----------------------------------------------------------
      Problem size     |            |             |    |
   m   |   l   |   n   |  Time (s)  |  (Gflop/s) | OK |
-----------------------------------------------------------
      4|      4|      4|      0.0000|  4.2667e-02|  T |
-----------------------------------------------------------
```

L'output fornisce tutte le informazioni necessare per eseguire il debug dell'applicazione. Per quanto riguarda l'esecuzione su cluster il debug viene rediretto su stderr, dunque contenuto nel file di errore.

```
[function: gendat, MPI rank: 0] My coordinates are: 0, 0
[function: gendat, MPI rank: 0] BLOCKS - M: 2 L: 2 N: 2
[function: gendat, MPI rank: 0] A[0 * 2 + 0]: 0 * 2 + 0 + 1 = 1
[function: gendat, MPI rank: 0] A[0 * 2 + 1]: 0 * 2 + 1 + 1 = 2
[function: gendat, MPI rank: 0] A[1 * 2 + 0]: 0 * 2 + 0 + 1 = 1
[function: gendat, MPI rank: 0] A[1 * 2 + 1]: 0 * 2 + 1 + 1 = 2
[function: gendat, MPI rank: 0] B[0 * 2 + 0]: 1.0 / ((0 * 2) + 0 + 1) = 1
[function: gendat, MPI rank: 0] B[0 * 2 + 1]: 1.0 / ((0 * 2) + 0 + 1) = 1
[function: gendat, MPI rank: 0] B[1 * 2 + 0]: 1.0 / ((0 * 2) + 1 + 1) = 0.5
[function: gendat, MPI rank: 0] B[1 * 2 + 1]: 1.0 / ((0 * 2) + 1 + 1) = 0.5
[function: mxm, MPI rank: 0] My coordinates are: 0, 0
[function: mxm, MPI rank: 0] Left rank: 1, Up rank: 2
[function: mxm, MPI rank: 0] Initial matrix alignment for A: shift of 0 to 0
[function: gendat, MPI rank: 1] My coordinates are: 0, 1
[function: gendat, MPI rank: 1] BLOCKS - M: 2 L: 2 N: 2
[function: gendat, MPI rank: 1] A[0 * 2 + 0]: 1 * 2 + 0 + 1 = 3
[function: gendat, MPI rank: 1] A[0 * 2 + 1]: 1 * 2 + 1 + 1 = 4
[function: gendat, MPI rank: 1] A[1 * 2 + 0]: 1 * 2 + 0 + 1 = 3
[function: gendat, MPI rank: 1] A[1 * 2 + 1]: 1 * 2 + 1 + 1 = 4
[function: gendat, MPI rank: 1] B[0 * 2 + 0]: 1.0 / ((0 * 2) + 0 + 1) = 1
[function: gendat, MPI rank: 1] B[0 * 2 + 1]: 1.0 / ((0 * 2) + 0 + 1) = 1
[function: gendat, MPI rank: 1] B[1 * 2 + 0]: 1.0 / ((0 * 2) + 1 + 1) = 0.5
[function: gendat, MPI rank: 1] B[1 * 2 + 1]: 1.0 / ((0 * 2) + 1 + 1) = 0.5
[function: mxm, MPI rank: 1] My coordinates are: 0, 1
[function: mxm, MPI rank: 1] Left rank: 0, Up rank: 3
[function: mxm, MPI rank: 1] Initial matrix alignment for A: shift of 0 to 1
[function: gendat, MPI rank: 2] My coordinates are: 1, 0
[function: gendat, MPI rank: 2] BLOCKS - M: 2 L: 2 N: 2
[function: gendat, MPI rank: 2] A[0 * 2 + 0]: 0 * 2 + 0 + 1 = 1
[function: gendat, MPI rank: 2] A[0 * 2 + 1]: 0 * 2 + 1 + 1 = 2
[function: gendat, MPI rank: 2] A[1 * 2 + 0]: 0 * 2 + 0 + 1 = 1
[function: gendat, MPI rank: 2] A[1 * 2 + 1]: 0 * 2 + 1 + 1 = 2
[function: gendat, MPI rank: 2] B[0 * 2 + 0]: 1.0 / ((1 * 2) + 0 + 1) =
    0.333333
[function: gendat, MPI rank: 2] B[0 * 2 + 1]: 1.0 / ((1 * 2) + 0 + 1) =
    0.333333
[function: gendat, MPI rank: 2] B[1 * 2 + 0]: 1.0 / ((1 * 2) + 1 + 1) = 0.25
[function: gendat, MPI rank: 2] B[1 * 2 + 1]: 1.0 / ((1 * 2) + 1 + 1) = 0.25
[function: mxm, MPI rank: 2] My coordinates are: 1, 0
[function: mxm, MPI rank: 2] Left rank: 3, Up rank: 0
[function: mxm, MPI rank: 2] Initial matrix alignment for A: shift of -1 to
    3
[function: gendat, MPI rank: 3] My coordinates are: 1, 1
[function: gendat, MPI rank: 3] BLOCKS - M: 2 L: 2 N: 2
[function: gendat, MPI rank: 3] A[0 * 2 + 0]: 1 * 2 + 0 + 1 = 3
[function: gendat, MPI rank: 3] A[0 * 2 + 1]: 1 * 2 + 1 + 1 = 4
```

```
[function: gendat, MPI rank: 3] A[1 * 2 + 0]: 1 * 2 + 0 + 1 = 3
[function: gendat, MPI rank: 3] A[1 * 2 + 1]: 1 * 2 + 1 + 1 = 4
[function: gendat, MPI rank: 3] B[0 * 2 + 0]: 1.0 / ((1 * 2) + 0 + 1) =
    0.333333
[function: gendat, MPI rank: 3] B[0 * 2 + 1]: 1.0 / ((1 * 2) + 0 + 1) =
    0.333333
[function: gendat, MPI rank: 3] B[1 * 2 + 0]: 1.0 / ((1 * 2) + 1 + 1) = 0.25
[function: gendat, MPI rank: 3] B[1 * 2 + 1]: 1.0 / ((1 * 2) + 1 + 1) = 0.25
[function: mxm, MPI rank: 0] Initial matrix alignment for B: shift of 0 to 0
[function: mxm, MPI rank: 0] Iterate through 2 dimensions
[function: mxm, MPI rank: 1] Initial matrix alignment for B: shift of -1 to
    3
[function: mxm, MPI rank: 3] My coordinates are: 1, 1
[function: mxm, MPI rank: 3] Left rank: 2, Up rank: 1
[function: mxm, MPI rank: 3] Initial matrix alignment for A: shift of -1 to
    2
[function: mxm, MPI rank: 1] Iterate through 2 dimensions
[function: mxm, MPI rank: 1] A: Sending data to 0, Receiving data from 0
[function: mxm, MPI rank: 1] B: Sending data to 3, Receiving data from 3
[function: mxm, MPI rank: 2] Initial matrix alignment for B: shift of 0 to 2
[function: mxm, MPI rank: 2] Iterate through 2 dimensions
[function: mxm, MPI rank: 2] A: Sending data to 3, Receiving data from 3
[function: mxm, MPI rank: 3] Initial matrix alignment for B: shift of -1 to
    1
[function: mxm, MPI rank: 3] Iterate through 2 dimensions
[function: mxm, MPI rank: 3] A: Sending data to 2, Receiving data from 2
[function: mxm, MPI rank: 0] A: Sending data to 1, Receiving data from 1
[function: mxm, MPI rank: 0] B: Sending data to 2, Receiving data from 2
[function: mxm, MPI rank: 0] A: Sending data to 1, Receiving data from 1
[function: mxm, MPI rank: 2] B: Sending data to 0, Receiving data from 0
[function: mxm, MPI rank: 1] A: Sending data to 0, Receiving data from 0
[function: mxm, MPI rank: 1] B: Sending data to 3, Receiving data from 3
[function: mxm, MPI rank: 1] Final matrix alignment for A: shift of 0 to 1
[function: mxm, MPI rank: 1] Final matrix alignment for B: shift of 1 to 3
[function: mxm, MPI rank: 2] A: Sending data to 3, Receiving data from 3
[function: mxm, MPI rank: 2] B: Sending data to 0, Receiving data from 0
[function: mxm, MPI rank: 2] Final matrix alignment for A: shift of 1 to 3
[function: mxm, MPI rank: 2] Final matrix alignment for B: shift of 0 to 2
[function: mxm, MPI rank: 3] B: Sending data to 1, Receiving data from 1
[function: mxm, MPI rank: 3] A: Sending data to 2, Receiving data from 2
[function: mxm, MPI rank: 3] B: Sending data to 1, Receiving data from 1
[function: mxm, MPI rank: 3] Final matrix alignment for A: shift of 1 to 2
[function: mxm, MPI rank: 3] Final matrix alignment for B: shift of 1 to 1
[function: mxm, MPI rank: 0] B: Sending data to 2, Receiving data from 2
[function: mxm, MPI rank: 0] Final matrix alignment for A: shift of 0 to 0
[function: mxm, MPI rank: 0] Final matrix alignment for B: shift of 0 to 0
```

La prima cosa che si nota è la verbositè del debug. La versione parallela contiene molte più informazioni rispetto alla corrispettiva parallela. Altra cosa da notare è la non linearità delle informazioni che si hanno: MPI esegue codice su diversi nodi/macchine e non assicura ovviamente un'esecuzione perfettamente sincronizzata.

3.5 Ottimizzazioni

MM MPI ha due implementazioni di base: la versione seriale e quella parallela. Ci sono poi una serie di ottimizzazioni sulla versione parallela per migliorare le performance di esecuzione.

3.5.1 MPI non bloccante

La prima ottimizzazione è a livello MPI. La versione standard di MM MPI utilizza *MPI_Sendrecv_replace* che ha un comportamento bloccante: questo significa che fino a che le copie non sono state effettuate la funzione non ritorna. Inoltre la funzione utilizza un solo buffer per inviare e ricevere dati e questo potrebbe rappresentare un collo di bottiglia per quanto riguarda le comunicazioni.

Si è provato a superare questo problema utilizzando due chiamate non bloccanti utilizzando un doppio buffer. La sezione di codice che lo implementa è la seguente:

```
1  // Shift matrix A left by one
2  MPI_Isend(a_buf[i % 2], A_DBLOCK, MPI_DOUBLE, left, 1, comm_2d, &
       request_handles[0]);
3  MPI_Irecv(a_buf[(i + 1) % 2], A_DBLOCK, MPI_DOUBLE, right, 1, comm_2d, &
       request_handles[1]);
4  debug_printf(__func__, rank, "A: Sending data to %i, Receiving data from %i\
       n", left, right);
5
6  // Shift matrix B up by one
7  MPI_Isend(b_buf[i % 2], B_DBLOCK, MPI_DOUBLE, up, 1, comm_2d, &
       request_handles[2]);
8  MPI_Irecv(b_buf[(i + 1) % 2], B_DBLOCK, MPI_DOUBLE, down, 1, comm_2d, &
       request_handles[3]);
9  debug_printf(__func__, rank, "B: Sending data to %i, Receiving data from %i\
       n", up, down);
10
11 // Let's wait all the shifts/communications to happen
12 MPI_Waitall(4, request_handles, status_handles);
```

a_buf e b_buf sono due array così definti:

```
1  a_buf[0] = a;
2  a_buf[1] = (double *)malloc(A_DBLOCK * sizeof(double));
3
4  b_buf[0] = b;
5  b_buf[1] = (double *)malloc(B_DBLOCK * sizeof(double));
```

così da avere un doppio buffer, uno per l'invio di dati ed uno per la ricezione di dati. In questo modo le operazioni di invio e di ricezione sono completamente indipendenti l'una dall'altra.

3.5.2 Moltiplicazione con OpenMP

Le altre due ottimizzazioni invece sono a livello di moltipicazioni tra matrici. La prima, OpenMP va a parallelizzare l'algoritmo seriale della moltiplicazioni tra matrici. L'algoritmo seriale ha 3 cicli for annidati e si è provato a parallelizzarli in diversi modi: sul ciclo esterno, su quello centrale, su quello interno e su quello interno e centrale insieme. Si è così generato 4 versioni per l'implementazione bloccante e 4 per l'implementazione non bloccante.

Un estratto di codice dell'ottimizzazione del ciclo esterno:

```
    // The parallel pragma starts a parallel block
    // By default data are shared
    // t, k, j are private, so just i is shared
    // the scheduler is automatic: the compiler will pick up the best one
#pragma omp parallel for default(shared) private(t,k,j) schedule(auto)
    for (i = 0; i < m; i++) {
        for (j = 0; j < l; j++) {
            t = a[i * l + j];
            for (k = 0; k < n; k++) {
                c[i * n + k] = c[i * n + k] + t * b[j * n + k];
            }
        }
    }
```

La regola gnerale di OpenMP è che tutte le variabili in uno scope esterno sono condivise di default nella regione parallela. Tutte le variabili in sola lettura dovrebbero essere condivise mentre le variabili che scrivo dovrebber essere locali e marcate come private.

3.5.3 Moltiplicazione con CBLAS dgemm

L'ultima ottimizzazione è l'utilizzo di CBLAS dgemm per la moltiplicazione delle sottomatrici. Questo sostituisce completamente i 3 cicli for annidati con la seguente chiamata:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, l, 1.0, a, l, b
    , n, 1.0, c, n);
```

- CblasRowMajor: indica che le matrici sono memorizzate riga x colonna
- CblasNoTrans: indica che le matrici A e B non devono essere trasposte prima della moltiplicazione
- m, n, l: interi che indicano le grandezze delle matrici: A[m x l], B[l x n], C[m x n]
- alpha: valore reale per scalare il prodotto delle matrici A e B
- a: matrice A

- l: numero di colonne in A
- b: matrice B
- n: numero di colonne in B
- alpha: valore reale per scalare la matrice C
- c: matrice C
- n: numero di colonne di C

Questa ottimizzazione genera due versioni di MM MPI: una per la versione bloccante ed una per la versione non bloccante

In totale si hanno 12 versioni di MM MPI:

- 1 versione seriale
- 1 versione parallela (senza ottimizzazioni)
- 8 versioni di OpenMP (4 bloccanti e 4 non bloccanti)
- 2 versioni cblas (1 bloccante ed una non bloccante)

Nel prossimo capitolo verranno analizzate le performance di tutte le versioni su diversi input.

4 Esperimenti

4.1 Cluster ChemGrid

Tutti gli esperimenti sono stati eseguiti sul cluster ChemGrid. Il cluster è composto da 11 nodi a 32 bit con 2 processori Intel(R) Pentium(R) 4 CPU 3.06GHz. Tutti i nodi hanno 1GB di memoria tranne 4 nodi (8, 9 10, 11) che hanno 4GB. 3 nodi (3, 6, 11) sono offline e dunque non utilizzabili per sottomettere job. Dunque i nodi utilizzabili sono 8 per un totale di 16 CPU.

Da ricordare che ci serve un quadrato perfetto per eseguire l'algoritmo di Cannon: 4, 9 e 16 sono i possibili candidati. 9 non è possibile utilizzarlo perchè non si hanno a disposizione sufficienti host. Il 4 potrebbe essere derivato da 4 host con una sola CPU oppure da 2 host con 2 CPU. Infine 16 è ottenibile solo con 8 host a 2 CPU l'uno. Tutti gli esperimenti verranno eseguiti con questa ultima configurazione: 8 host e 2 CPU per nodo, per un totale di 16 CPU.

4.2 Data file

Il file di input utilizzato per gli esperimenti ha le seguenti dimensioni:

```
4 4 4 2
8 8 8 2
16 16 16 2
32 32 32 2
64 64 64 2
128 128 128 2
192 192 192 2
256 256 256 2
384 384 384 2
512 512 512 2
768 768 768 2
1024 1024 1024 2
```

```
1536 1536 1536 2
2048 2048 2048 2
3072 3072 3072 2
4096 4096 4096 2
```

Non ha senso andare oltre 4096 perchè la computazione inizia ad essere impegnativa impiegando tutte le risorse del cluster e, come si vedrà in seguito le prestazioni iniziano ad essere stabili dal 1024.

Si consideri che una matrice quadrata con n = 4096 ha 134217728 elementi. Ogni elemento è un double che occupa 8 bytes, dunque una matrice occupa circa 131Mb.

Tutti i test sono stati fatti utilizzando 2 iterazioni.

Per ogni esperimento verranno mostrati due grafici: uno dei tempi di esecuzione di colore blu e l'altro di colore rosso che rappresenta il throughput in Gflop/s. Il throughput è calcolato nel seguente modo:

$$gflop/s = \frac{2*n^3}{time} * 1.0e-9$$

Dove *time* è la media dei tempi tra le due iterazioni.

4.2.1 Seriale

La versione seriale è stata eseguita su un nodo (cg11) che ha 4GB di memoria per permettere alle dimensioni più grandi si essere moltiplicate senza problemi. Di seguito i suoi risultati:

```
   n|         time|    Gflops/s
------------------------------
   4|       0.0000|  1.2800e-01
   8|       0.0000|  4.0960e-01
  16|       0.0000|  7.6205e-01
  32|       0.0001|  8.5250e-01
  64|       0.0006|  8.6170e-01
 128|       0.0045|  9.4038e-01
 192|       0.0163|  8.7050e-01
 256|       0.0409|  8.1963e-01
 384|       0.1301|  8.7073e-01
 512|       0.3523|  7.6190e-01
 768|       1.2080|  7.4996e-01
1024|       3.1203|  6.8823e-01
1536|      11.0251|  6.5739e-01
2048|      27.9096|  6.1555e-01
3072|      89.7327|  6.4616e-01
4096|     222.7798|  6.1693e-01
```

Le rappresentazioni grafiche dei dati sono nelle figure 4.1 e 4.2.

Si nota subito che l'andamento del tempo è esponenziale al crescere delle dimensioni delle matrici mentre il throughput si inizia a stabilizzare dalla matrice a 1024. Per le matrici con dimensioni che vanno da 32 a 384 si hanno dei throughput migliori.

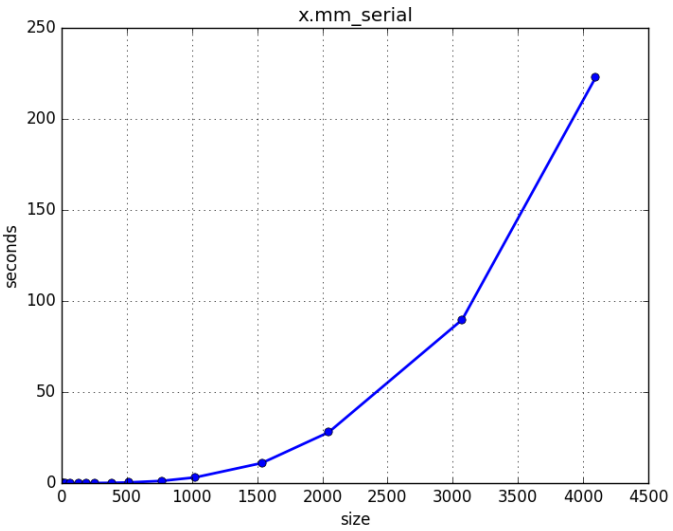


Figura 4.1: MM MPI seriale: tempo di esecuzione

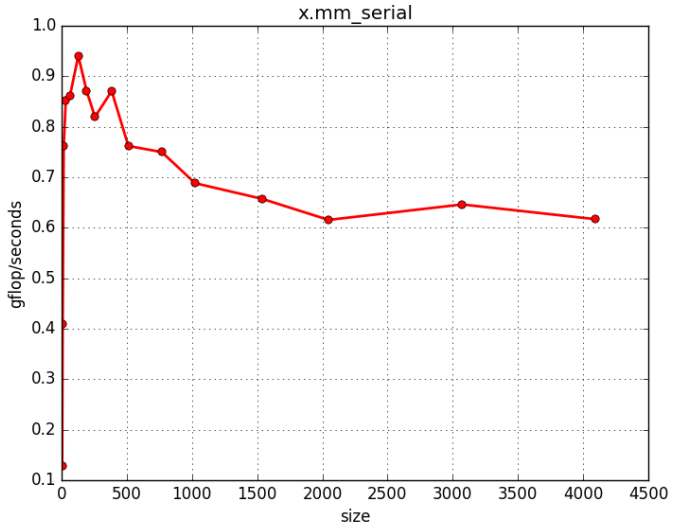


Figura 4.2: MM MPI seriale: Gflop/s

4.2.2 Versione MPI bloccante

Qui di seguito i risultati della version MPI **bloccante** senza alcuna ottimizzazione. Si ricorda che questa versione fa uso della primitivat MPI *MPI_Sendrecv_replace()* e che utilizza un solo buffer sia per l'invio che per la ricezione.

```
   n|         time|     Gflops/s
------------------------------
   4|       0.0247|   5.1870e-06
   8|       0.0138|   7.4089e-05
  16|       0.0083|   9.8622e-04
  32|       0.0057|   1.1505e-02
  64|       0.0054|   9.6908e-02
 128|       0.0073|   5.7146e-01
 192|       0.0131|   1.0801e+00
 256|       0.0233|   1.4377e+00
 384|       0.0514|   2.2017e+00
 512|       0.3167|   8.4772e-01
 768|       0.6362|   1.4241e+00
1024|       8.8529|   2.4257e-01
1536|      25.2136|   2.8745e-01
2048|     103.0118|   1.6678e-01
3072|     353.7942|   1.6389e-01
4096|     875.8602|   1.5692e-01
```

Le figure 4.3 e 4.4 sono le relative rappresentazioni grafiche per il tempo trascorso e la velocità di trasmissione dei dati.

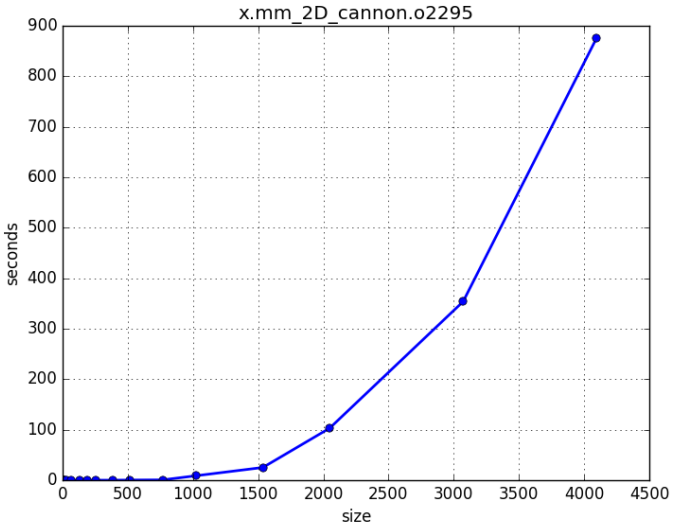


Figura 4.3: MM MPI bloccante: tempo di esecuzione

Come nella versione seriale, si vede che i tempi di esecuzione crescono in maniera esponenziale e che dalla matrice 1024 il throughput si stabilizza.

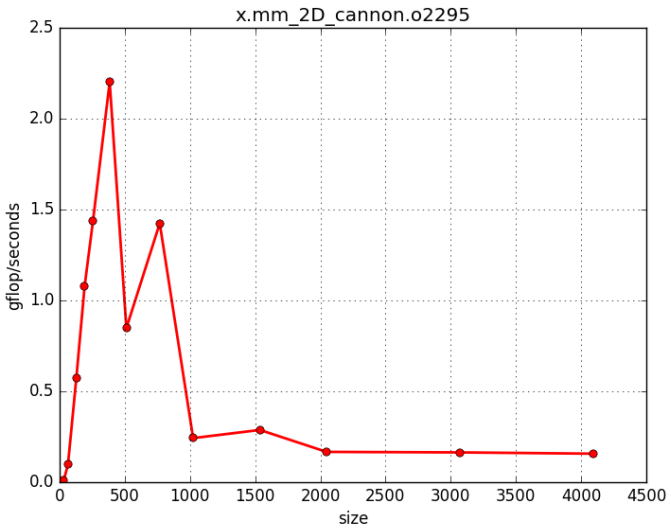


Figura 4.4: MM MPI bloccante: Gflop/s

Speedup ed efficienza Prima di calcolare speedup ed efficienza, si ricorda come vengono calcolati. Lo speedup è così definito:

$$S(p) = \frac{T(1)}{T(p)}$$

e misura la riduzione del tempo di esecuzione rispetto all'algoritmo seriale. L'efficienza invece è definita come:

$$E(p) = \frac{S(p)}{p}$$

e misura quanto l'algoritmo sfrutta il parallelismo del calcolatore. Ecco di seguito i risultati tra l'implementazione seriale e quella parallela bloccante.

```
size| serial time|  paral time|     speedup|  efficiency
-----------------------------------------------------------
   4|         0.0|      0.0247|      0.0000|      0.0000
   8|         0.0|      0.0138|      0.0000|      0.0000
  16|         0.0|      0.0083|      0.0000|      0.0000
  32|      0.0001|      0.0057|      0.0175|      0.0011
  64|      0.0006|      0.0054|      0.1111|      0.0069
 128|      0.0045|      0.0073|      0.6164|      0.0385
 192|      0.0163|      0.0131|      1.2443|      0.0778
 256|      0.0409|      0.0233|      1.7554|      0.1097
 384|      0.1301|      0.0514|      2.5311|      0.1582
 512|      0.3523|      0.3167|      1.1124|      0.0695
 768|       1.208|      0.6362|      1.8988|      0.1187
1024|      3.1203|      8.8529|      0.3525|      0.0220
1536|     11.0251|     25.2136|      0.4373|      0.0273
```

```
2048|      27.9096|     103.0118|       0.2709|       0.0169
3072|      89.7327|     353.7942|       0.2536|       0.0159
4096|     222.7798|     875.8602|       0.2544|       0.0159
```

La figura 4.5 mostra il grafico dello speedup tra le due implementazioni.

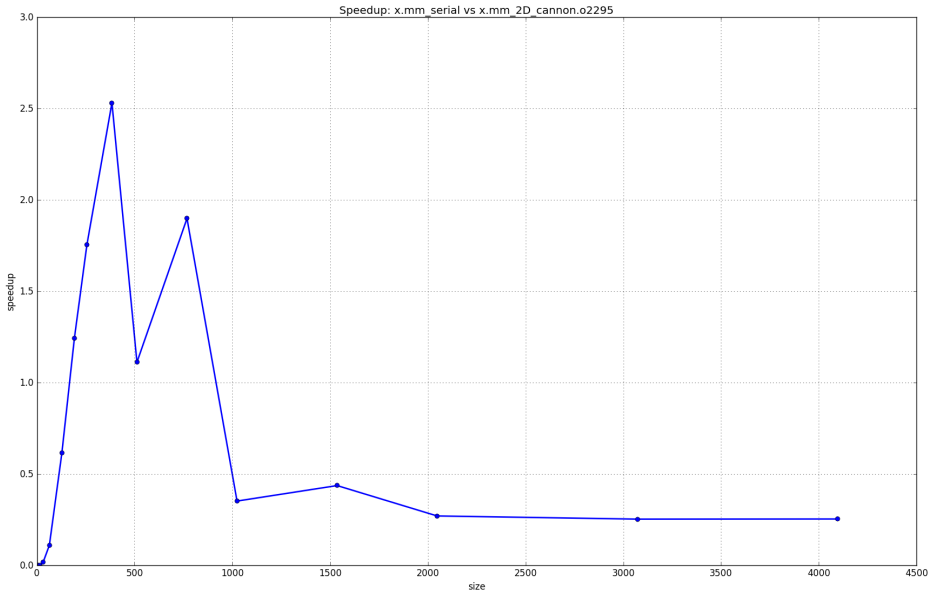


Figura 4.5: Speedup seriale vs 2d cannon

Come si può vedere l'algoritmo ha uno speedup piuttosto basso tranne per le dimensioni che vanno da 192 a 768 che è maggiore di 1. L'efficienza nel caso migliore non supera lo 0.15. Si può concludere che l'implementazione parallela bloccante non e' migliore rispetto alla corrispettiva parallela, tranne che per alcune dimensioni. Questo vuole dire che le comunicazioni tra i nodi occupano la maggior parte del tempo nell'algoritmo.

4.2.3 Versione MPI NON bloccante

Di seguito i dati della versione non bloccante

```
   n|         time|     Gflops/s
------------------------------
   4|       0.0017|   7.4010e-05
   8|       0.0021|   4.9119e-04
  16|       0.0023|   3.5350e-03
  32|       0.0026|   2.5267e-02
  64|       0.0035|   1.5188e-01
```

```
 128|       0.0063|   6.6103e-01
 192|       0.0118|   1.1991e+00
 256|       0.0211|   1.5869e+00
 384|       0.0482|   2.3513e+00
 512|       0.1109|   2.4210e+00
 768|       0.5203|   1.7411e+00
1024|       8.7743|   2.4475e-01
1536|      25.3017|   2.8645e-01
2048|     102.1089|   1.6825e-01
3072|     358.1640|   1.6189e-01
4096|     892.7144|   1.5396e-01
```

e le relative rappresentazioni grafiche in figura 4.6 e 4.7

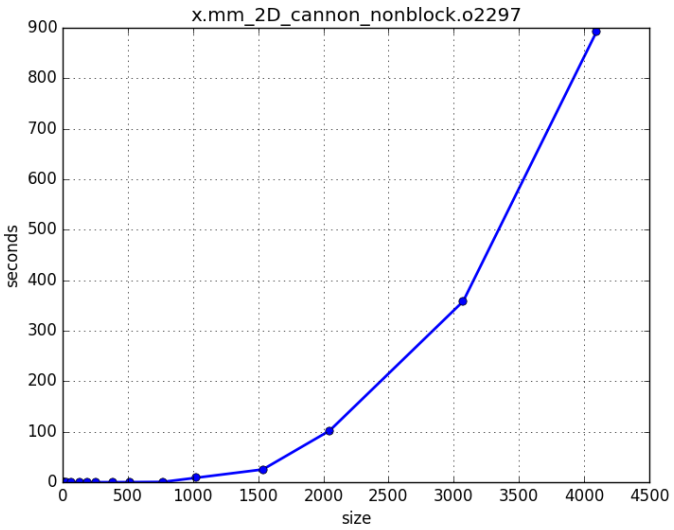


Figura 4.6: MM MPI NON bloccante: tempo di esecuzione

La situazione generale dell'implementazione parallela utilizzando primitive non bloccanti è simile alla corrispettiva bloccante. Sembra leggermente meno lineare intorno alla dimensione 512.

Speedup ed efficienza Di seguito speedup ed efficienza dell'algoritmo parallelo non bloccante rispetto all'implementazione seriale.

```
size| serial time|  paral time|     speedup|  efficiency
-----------------------------------------------------------
   4|         0.0|      0.0017|      0.0000|      0.0000
   8|         0.0|      0.0021|      0.0000|      0.0000
  16|         0.0|      0.0023|      0.0000|      0.0000
  32|      0.0001|      0.0026|      0.0385|      0.0024
  64|      0.0006|      0.0035|      0.1714|      0.0107
 128|      0.0045|      0.0063|      0.7143|      0.0446
 192|      0.0163|      0.0118|      1.3814|      0.0863
```

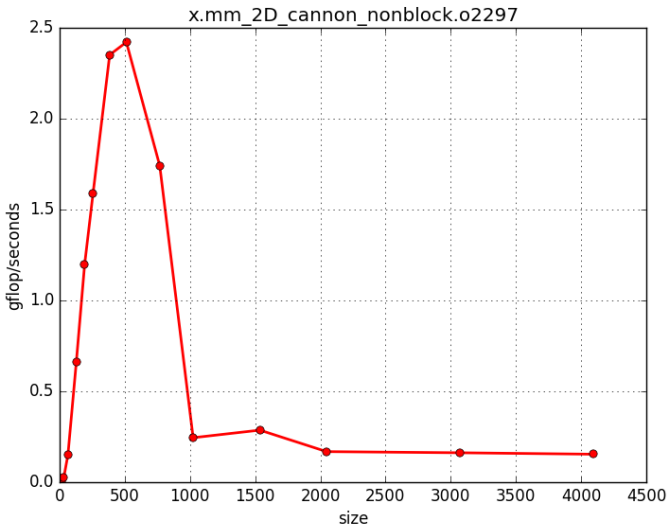


Figura 4.7: MM MPI NON bloccante: Gflop/s

```
 256|       0.0409|       0.0211|       1.9384|       0.1211
 384|       0.1301|       0.0482|       2.6992|       0.1687
 512|       0.3523|       0.1109|       3.1767|       0.1985
 768|        1.208|       0.5203|       2.3217|       0.1451
1024|       3.1203|       8.7743|       0.3556|       0.0222
1536|      11.0251|      25.3017|       0.4357|       0.0272
2048|      27.9096|     102.1089|       0.2733|       0.0171
3072|      89.7327|      358.164|       0.2505|       0.0157
4096|     222.7798|     892.7144|       0.2496|       0.0156
```

La figura 4.8 mostra il grafico dello speedup tra le due implementazioni.

Come nel precedente caso, lo speedup generale rimane sotto lo 0.5 tranne per le dimensioni che vanno da 192 a 768 dove lo speedup supera il 3. L'efficienza generale è abbastanza scarsa per tutti quanti gli input: il caso migliore rimane 384 dove l'efficienza raggiunge quasi 0.20. Un valore comunque basso per definire l'implementazione efficiente.

4.2.4 MPI con OpenMP

Per questioni di spazio non verranno riportati i dati grezzi degli esperimenti fatti con OpenMP, ma nell figure 4.9 ed 4.10 si possono vedere le loro rappresentazioni grafiche

È interessante vedere come diverse ottimizzazioni OpenMP impattano sulle performance generali dell'algoritmo addirittura rendendo a volte l'algoritmo meno veloce dell'implementazione seriale. Questi sono i casi delle otti-

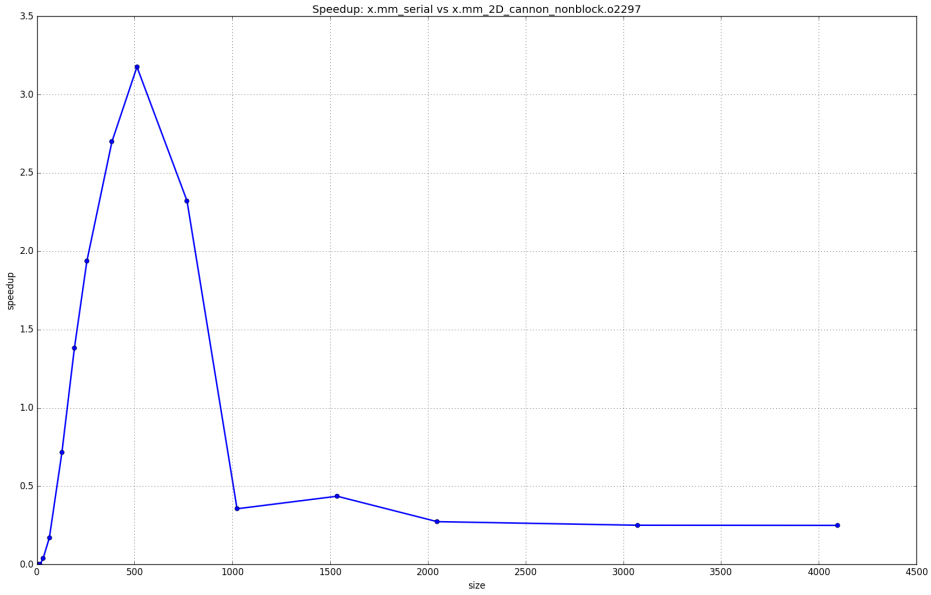


Figura 4.8: Speedup seriale vs 2d cannon NON bloccante

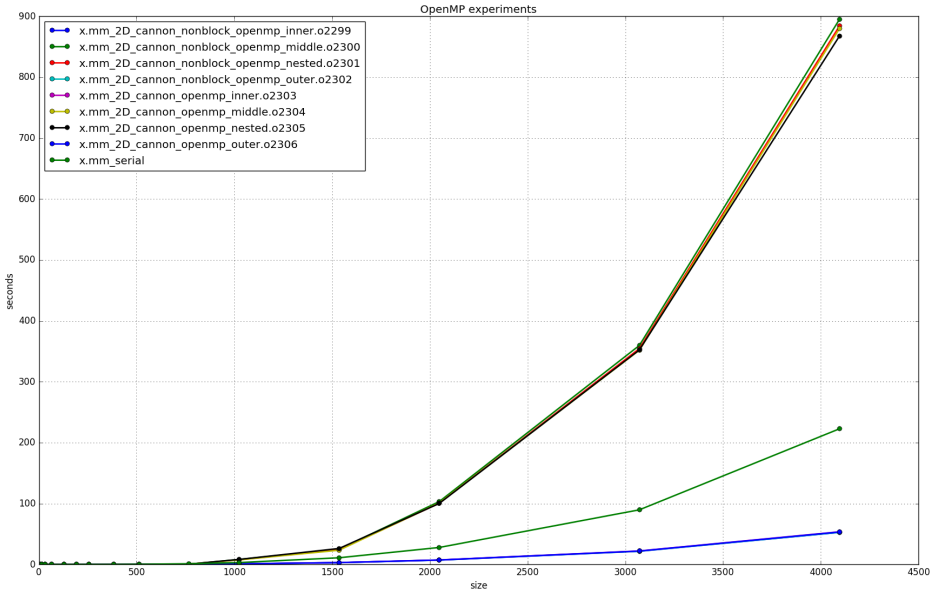


Figura 4.9: MM MPI with OpenMP: tempo di esecuzione

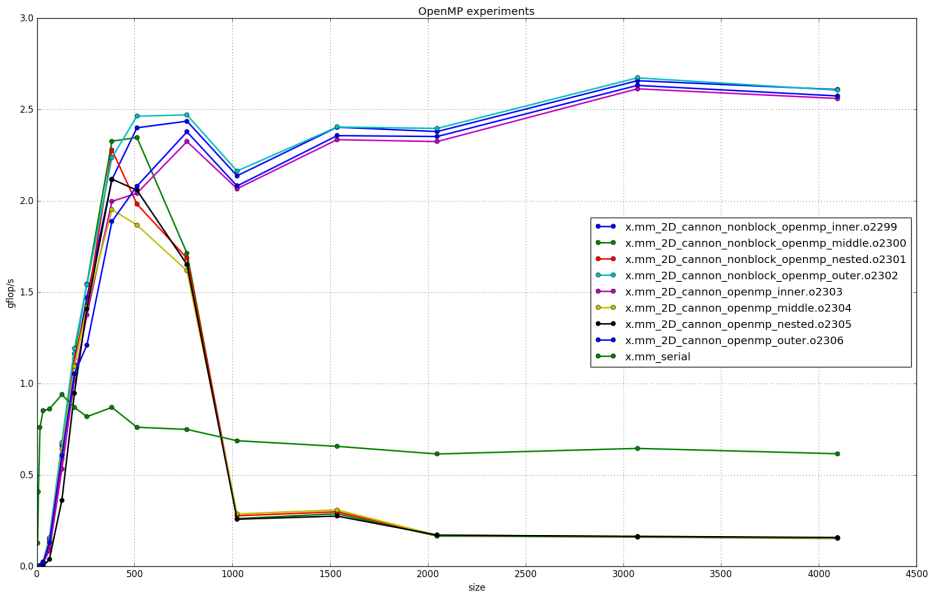


Figura 4.10: MM MPI with OpenMP: Gflop/s

mizzazioni middle e nested. Le ottimizzazioni outer ed inner (sia bloccanti, sia non bloccanti) risultano più veloci dell'implementazione seriale.

Speedup ed efficienza Senza calcolare tutti gli speedup e le efficienze di ogni implementazione, si è scelto di calcolare speedup/efficienza prendendo uno degli algoritmi più veloci: la versione non bloccante con l'ottimizzazione nel ciclo più esterno.

```
size| serial time|  paral time|     speedup|  efficiency
-----------------------------------------------------------
   4|          0.0|      0.0019|      0.0000|      0.0000
   8|          0.0|      0.0022|      0.0000|      0.0000
  16|          0.0|      0.0024|      0.0000|      0.0000
  32|       0.0001|      0.0025|      0.0400|      0.0025
  64|       0.0006|      0.0034|      0.1765|      0.0110
 128|       0.0045|      0.0062|      0.7258|      0.0454
 192|       0.0163|      0.0119|      1.3697|      0.0856
 256|       0.0409|      0.0218|      1.8761|      0.1173
 384|       0.1301|      0.0506|      2.5711|      0.1607
 512|       0.3523|      0.1089|      3.2351|      0.2022
 768|        1.208|      0.3666|      3.2951|      0.2059
1024|       3.1203|      0.9924|      3.1442|      0.1965
1536|      11.0251|      3.0139|      3.6581|      0.2286
2048|      27.9096|      7.1668|      3.8943|      0.2434
3072|      89.7327|     21.6833|      4.1383|      0.2586
4096|     222.7798|      52.736|      4.2244|      0.2640
```

La figura 4.11 mostra il grafico dello speedup tra le due implementazioni.

Come si può vedere sia dai dati sia dall figura 4.11, lo speedup inizia ad essere migliore rispetto alle precedenti implementazioni. Infatti si arriva ad avere uno speedup di 4 per le dimensioni più grandi (3072, 4096). Di conseguenza anche l'efficienza ha un massimo di circa 0.27, poco più di un quarto dell'efficienza ideale (1). Questa implementazione inizia ad essere più interessante soprattuto per dimensioni più grandi dove una potenza di calcolo distribuito diventa una necessità.

4.2.5 MPI con CBLAS

Come nel caso precedente, non si riportano i dati grezzi degli esperimenti fatti con l'ottimizzazione CBLAS ma le figure 4.12 e 4.13 rappresentano in maniera esaustiva i risultati di questi esperimenti.

Con l'ottimizzazione cblas si nota subito un netto miglioramento di velocità e throughput, portando la versione non bloccante in leggero vantaggio rispetto alla corrispettiva bloccante. Questo significa che l'utilizzo di primitive non bloccanti ottimizza l'uso delle risorse evitando momenti "morti" che potrebbero essere utilizzati per effettuare la moltiplicazione.

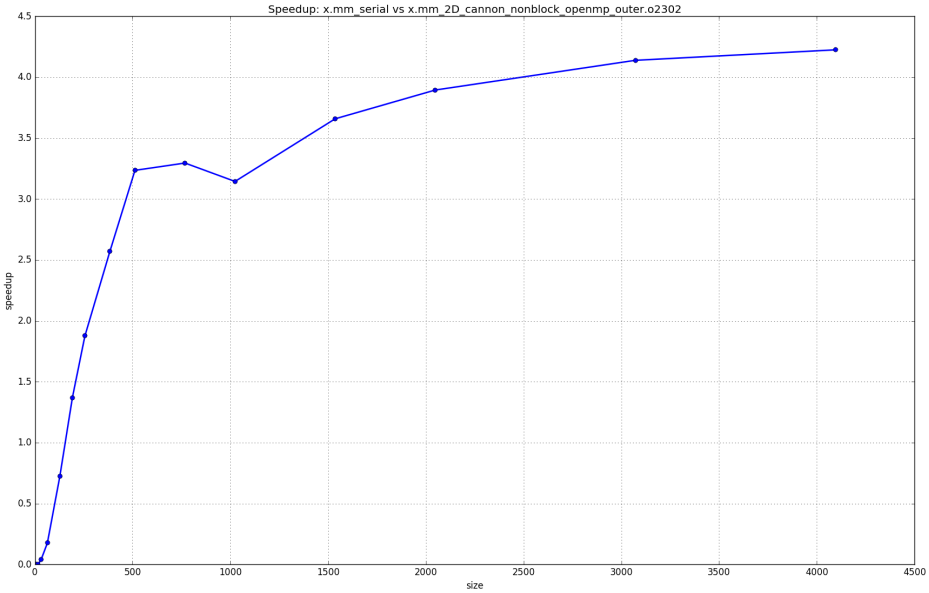


Figura 4.11: Speedup seriale vs 2d cannon NON bloccante OpenMP outer

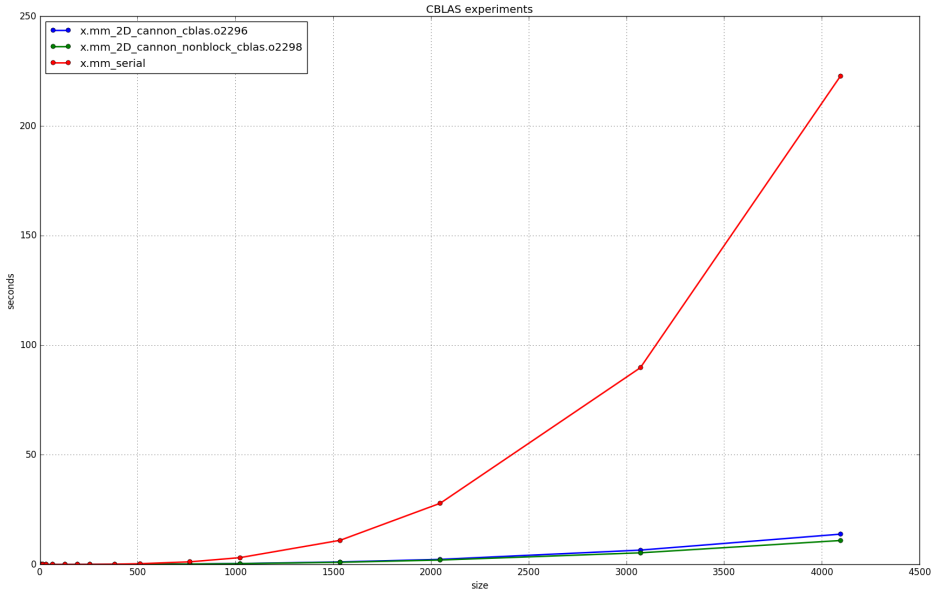


Figura 4.12: MM MPI with CBLAS: tempo di esecuzione

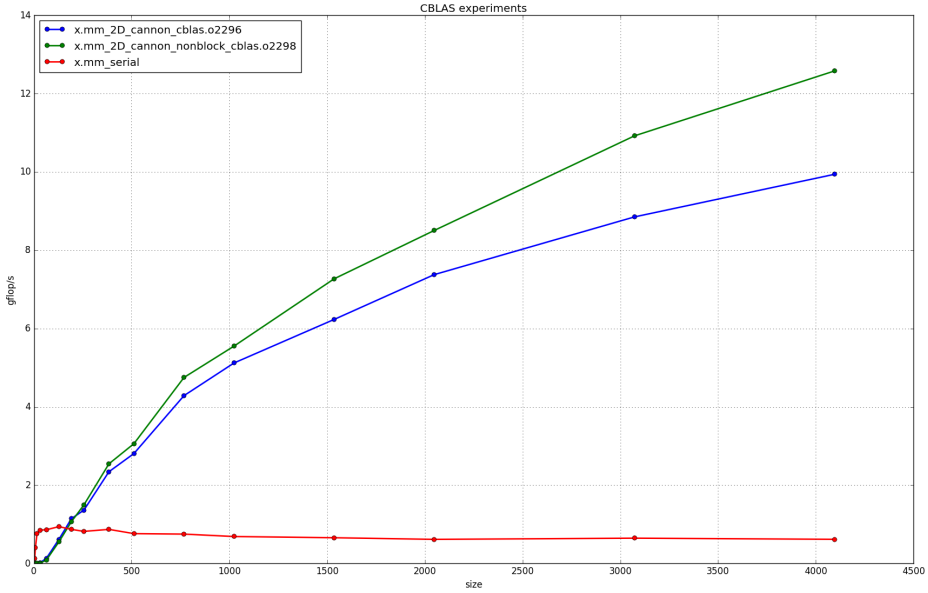


Figura 4.13: MM MPI with CBLAS: Gflop/s

Speedup ed efficienza Dunque l'implementazione cblas più veloce è quella non bloccante: i risultati di questa verranno utilizzati per calcolare speedup ed efficienza.

```
size| serial time|  paral time|     speedup|  efficiency
-----------------------------------------------------------
   4|         0.0|      0.0018|      0.0000|      0.0000
   8|         0.0|      0.0217|      0.0000|      0.0000
  16|         0.0|      0.0124|      0.0000|      0.0000
  32|      0.0001|      0.0076|      0.0132|      0.0008
  64|      0.0006|      0.0059|      0.1017|      0.0064
 128|      0.0045|      0.0076|      0.5921|      0.0370
 192|      0.0163|      0.0133|      1.2256|      0.0766
 256|      0.0409|      0.0224|      1.8259|      0.1141
 384|      0.1301|      0.0445|      2.9236|      0.1827
 512|      0.3523|      0.0877|      4.0171|      0.2511
 768|       1.208|      0.1907|      6.3346|      0.3959
1024|      3.1203|      0.3869|      8.0649|      0.5041
1536|     11.0251|      0.9971|     11.0572|      0.6911
2048|     27.9096|      2.0202|     13.8153|      0.8635
3072|     89.7327|      5.3102|     16.8982|      1.0561
4096|    222.7798|     10.9267|     20.3886|      1.2743
```

La figura 4.14 mostra il grafico dello speedup tra le due implementazioni.

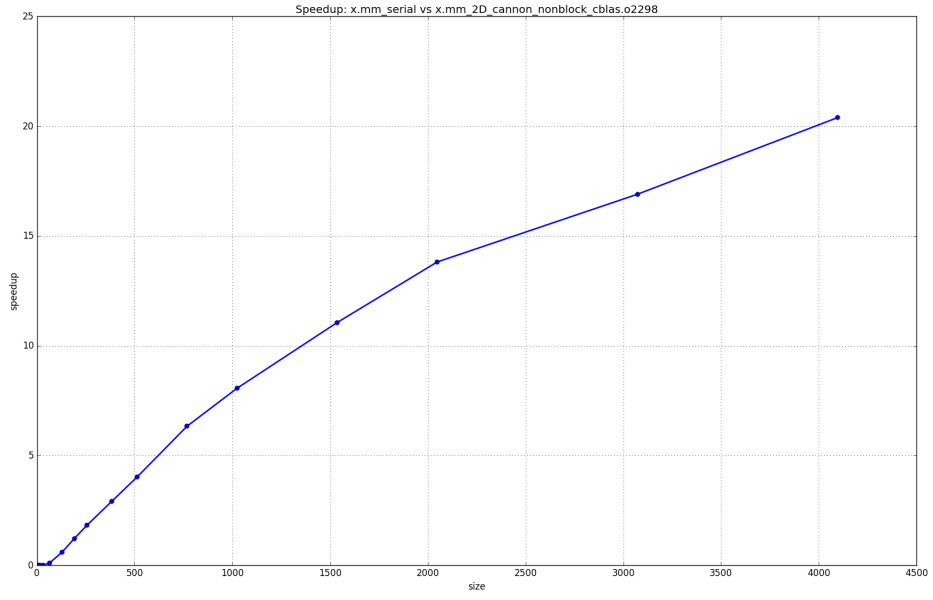


Figura 4.14: Speedup seriale vs 2d cannon NON bloccante CBLAS

Questa è la migliore implementazione in assoluto: infatti a partire dalla dimensione 3072 inizia ad avere uno speedup ed una efficienza maggiori di

quella ideale. Si ricorda che lo speedup ideale equivale a $S_{ideale}(p) = p$ mentre l'efficienza ideale equivale a $E_{ideale}(p) = 1$. Dunque per matrici medio grandi, l'implementazione non bloccante con cblas è più che ideale.

4.2.6 Visioni di insieme

In questa ultima sezione si ha una visione di insieme su tutti esperimenti effettuati (figure 4.15 e 4.16).

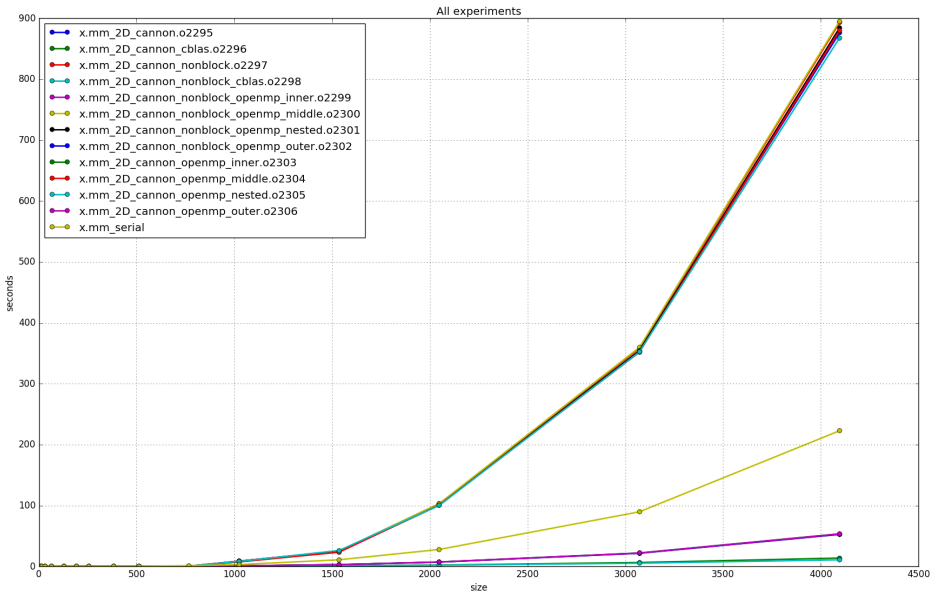


Figura 4.15: MM MPI: tempo di esecuzione

Tutti gli esperimenti bloccanti nelle figure 4.17 e 4.18

Tutti gli esperimenti NON bloccanti nelle figure 4.19 e 4.20

Quello che è interessante vedere in questi grafici è l'impatto che le varie ottimizzazioni hanno sull'algoritmo parallelo. In generale una comunicazione non bloccante favorisce le performance e throughput dell'algoritmo e più è ottimizzata la moltiplicazione locale nel nodo meglio l'algoritmo performa.

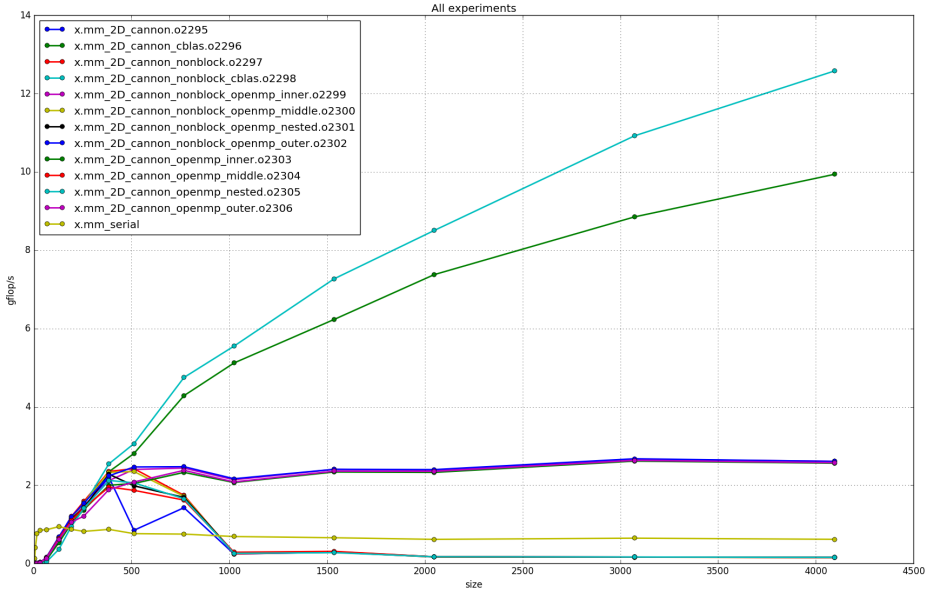


Figura 4.16: MM MPI: Gflop/s

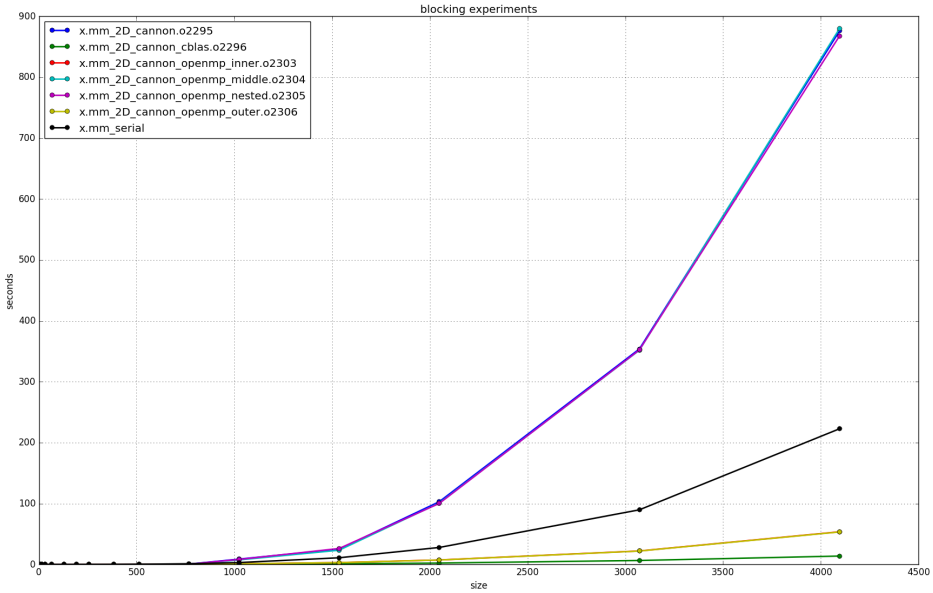


Figura 4.17: MM MPI bloccanti: tempo di esecuzione

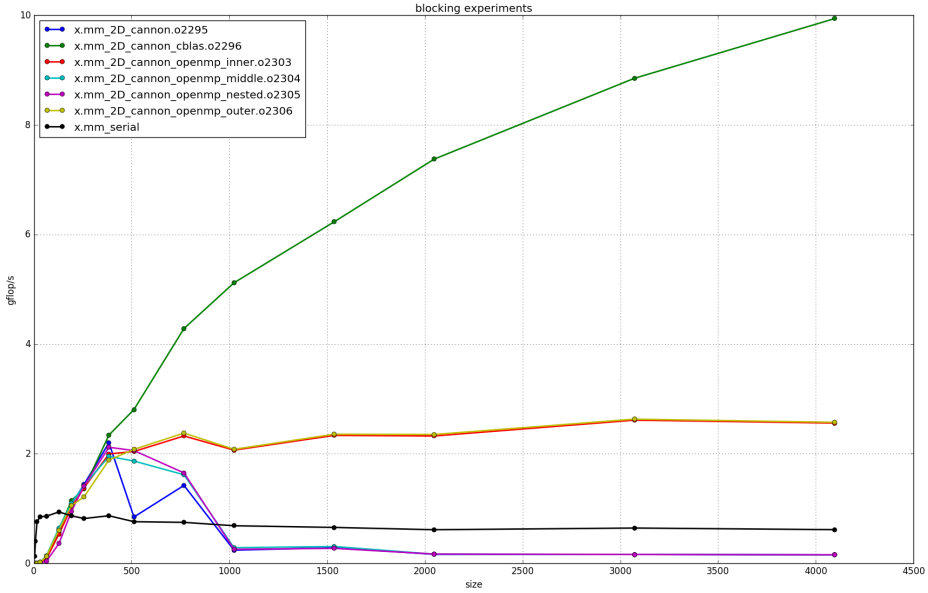


Figura 4.18: MM MPI bloccanti: Gflop/s

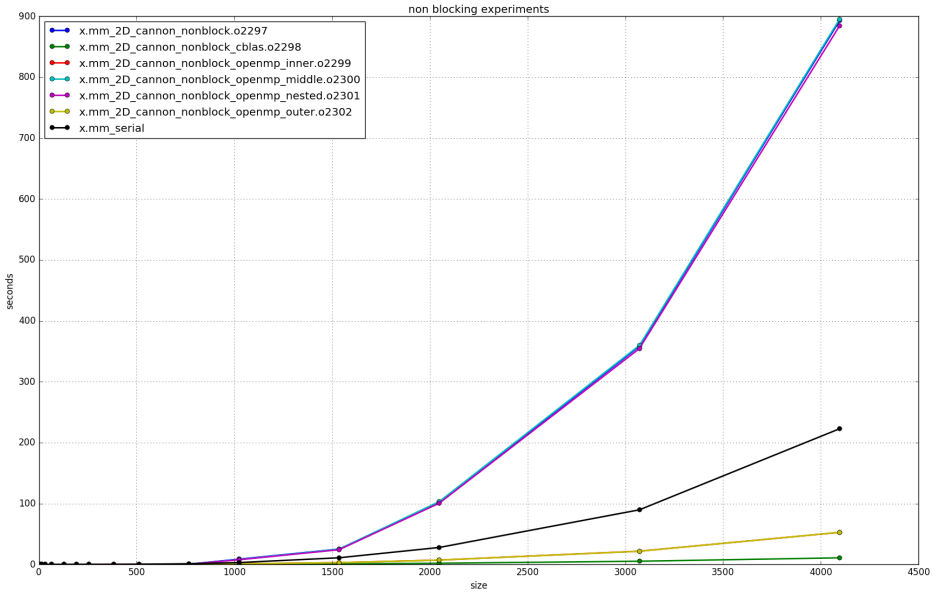


Figura 4.19: MM MPI NON bloccanti: tempo di esecuzione

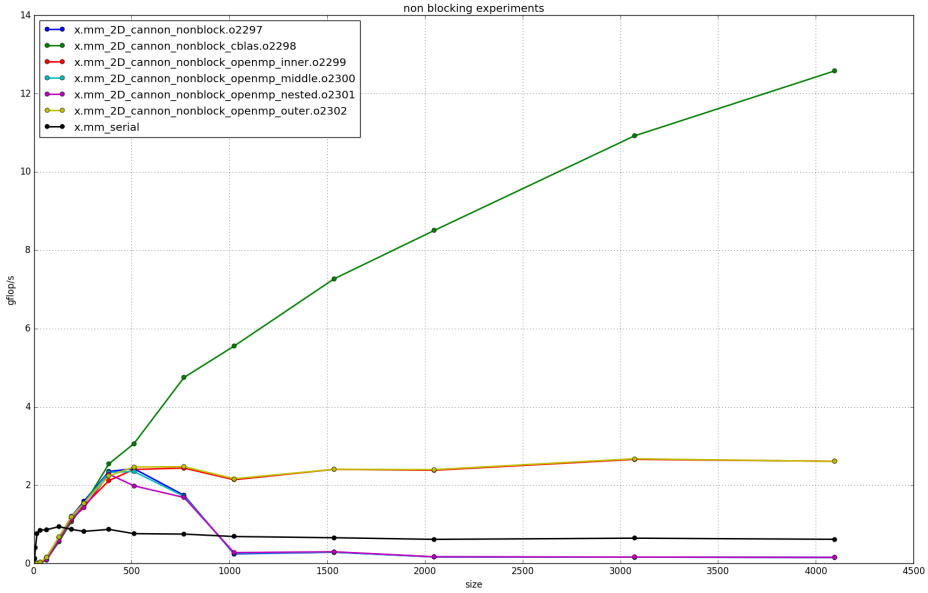


Figura 4.20: MM MPI bloccanti: Gflop/s

5 Conclusione

Il lavoro svolto è un proof of concept di un'applicazione parallela e della sua ottimizzazione per riuscire ad avere degli speedup interessanti. Il percorso effettuato dall'implementazione seriala a quella parallela con CBLAS è stato reso interessante dai risultati degli esperimenti, a volte totalmente inaspettati e sorprendenti. È incredibile vedere come piccoli accorgimenti cambiano totalemente il risultato finale. Alla fine si è raggiunto l'obbiettivo di avere una implementazione parallela che funzioni meglio della sua versione seriale.

5.1 Note finali

MM MPI è lontana da essere utilizzabile in applicazioni reali.

Dal punto di vista funzionale non accetta nessun stream che contenga i dati delle matrici e che non ritorna nessun prodotto.

Invece di generare un binario per ogni ottimizzazione si potrebbe cambiare il design dell'applicazione in modo da passare da console il tipo di ottimizzazione che si vuole attivare.

Questi sono miglioramenti strettamente funzionali che renderebbero l'applicazione utilizzabile in un contesto più grande.

Dal punto di vista delle ottimizzazioni si potrebbe esplorare un pochino meglio OpenMP. Altro suggerimento è quello di utilizzare OpenCL e sfruttare GPU presenti su ogni nodo. Questo renderebbe estremamente veloce le moltiplicazioni locali a favore dell'intero algoritmo.