MM MPI

Diego Russo

Chi sono

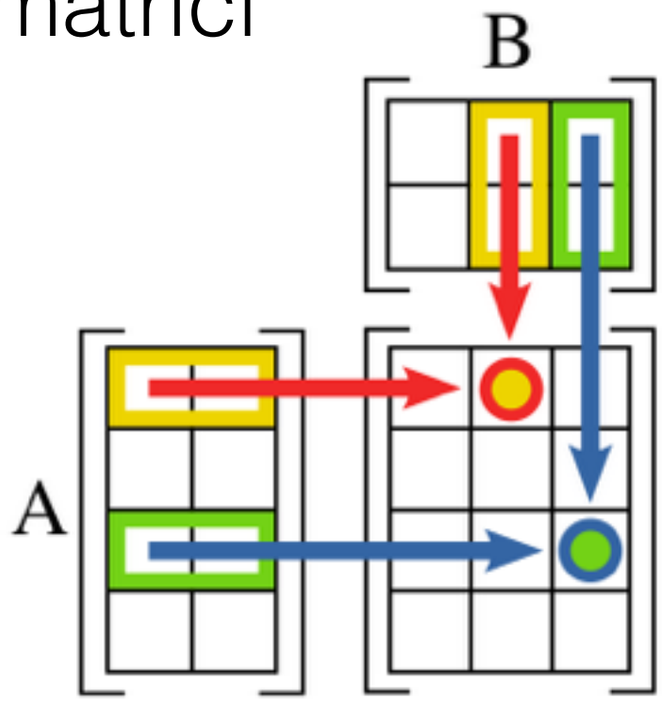
- Diego Russo
- Studente Specialistica, 231423
- Sviluppatore Software @ ARM Ltd
- Vivo e lavoro a Cambridge, UK
- http://www.diegor.uk - me@diegor.it
- @diegor

Il progetto

- Sviluppo ed implementazione di un codice parallelo per il calcolo del prodotto matrice-matrice
- L'algoritmo deve utilizzare la decomposizione a blocchi per le matrici e fare uso di primitive collettive/virtuali
- In particolare, lo studente deve sviluppare un proprio algoritmo parallelo non triviale e valutarne speedup ed efficienza.

La moltiplicazione tra matrici

- Prodotto tra righe e colonne tra due matrici
- Diverse tipologie di matrici
- # colonne di A = # di righe di B
- A[4x2] x B[2x3] = C[4x3]



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

Cannon

A

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

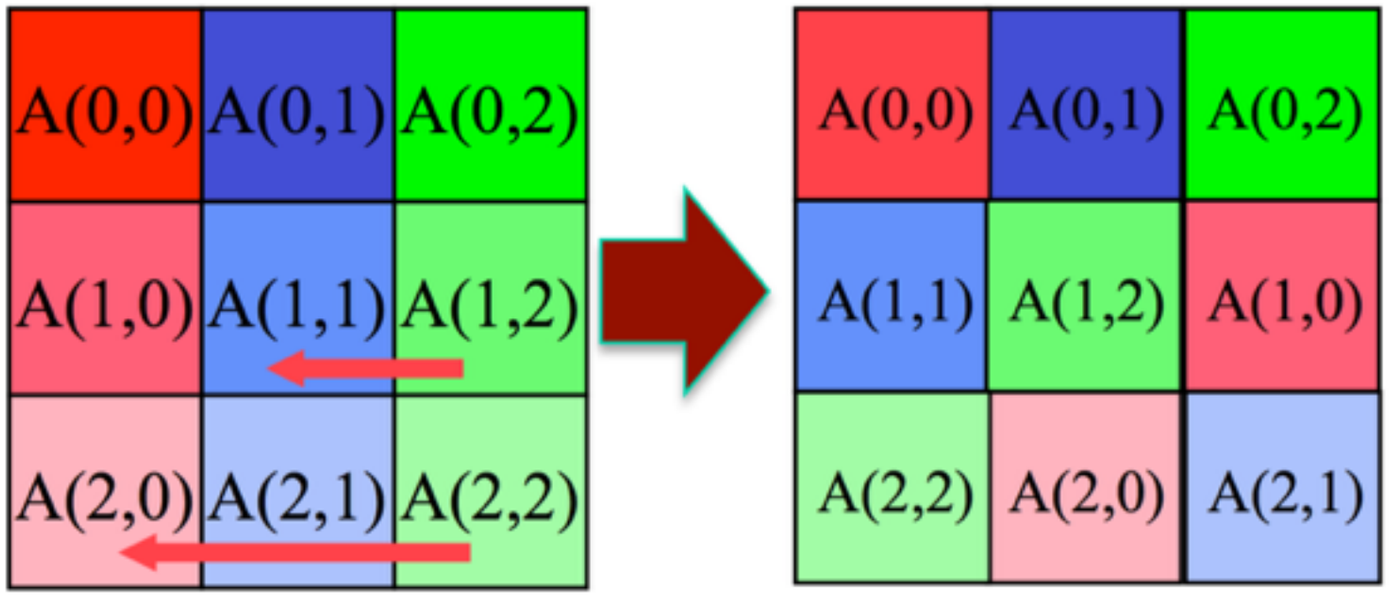
B

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

C[1,2] = A[1,0] * B[0,2] +A[1,1] * B[1,2] + A[1,2] * B[2,2]

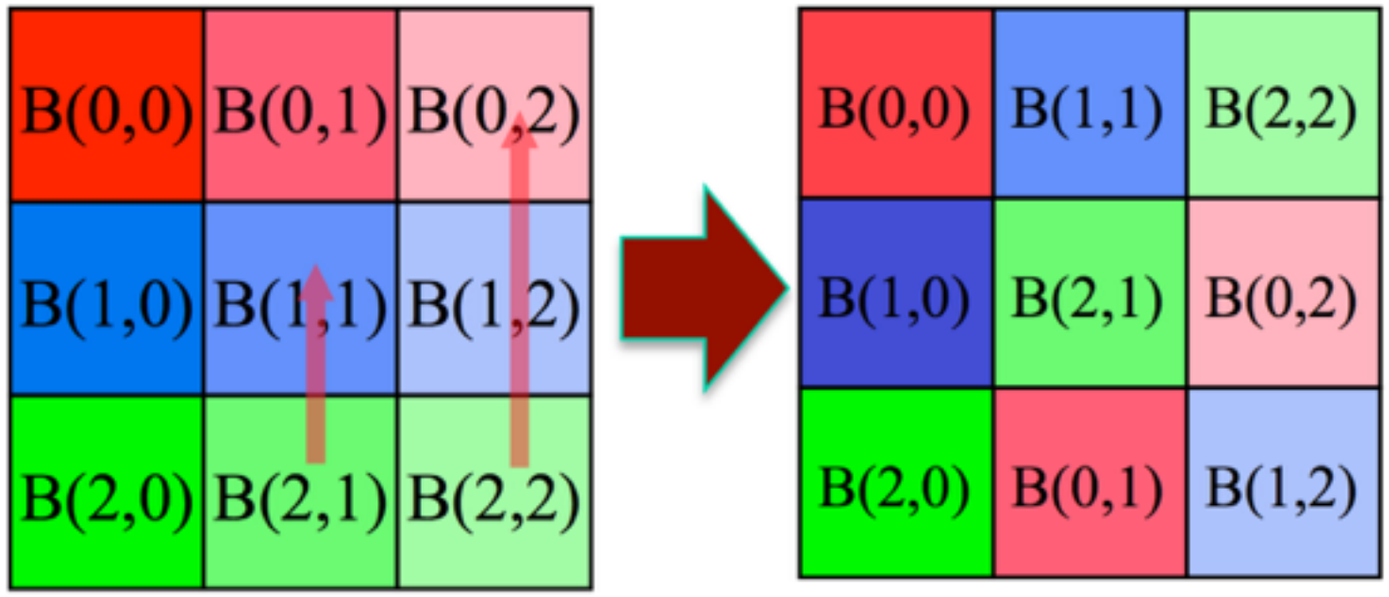
- Partizione in p blocchi quadrati
- Ogni processo ha un blocco n / √p
- Dati inviati in maniera incrementale in √p passi

Cannon



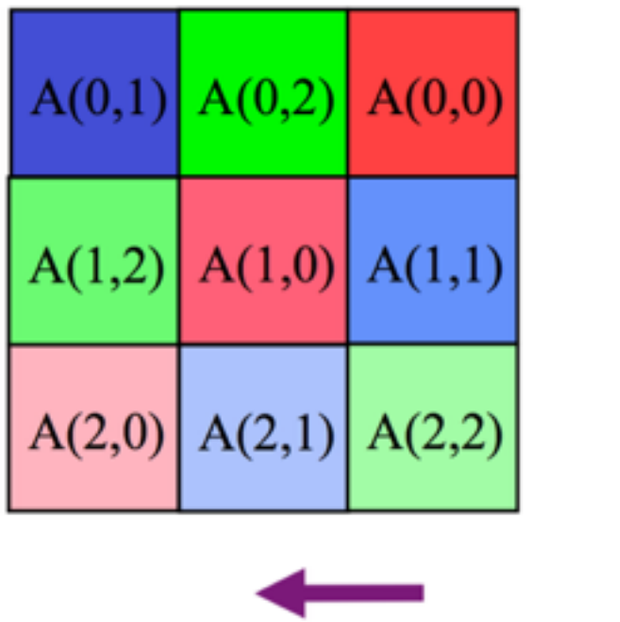
Shift Iniziale di A: left di i step su ogni riga

Cannon

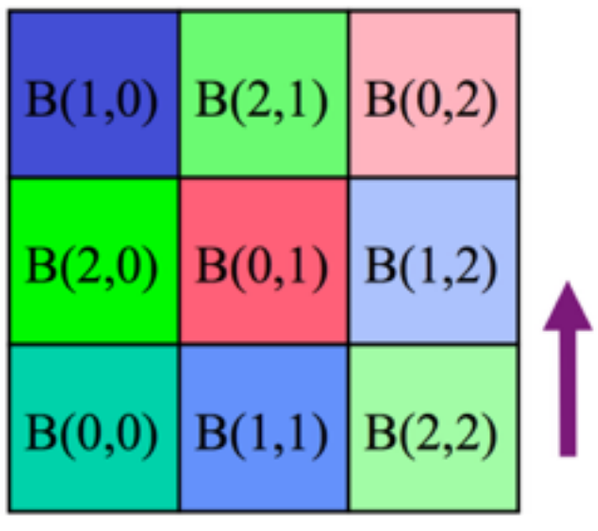


Shift Iniziale di B: up di j step su ogni colonna

Cannon



$$T_p = \frac{n^3}{p} + 2\sqrt{p}(t_s + \frac{t_w n^2}{p}) + 2(t_s + \frac{t_w n^2}{p})$$



- Costo moltiplicazione di √p matrici
- Costo di √p shift
- Costo shift iniziale

Ulteriore shift di A e B

MM MPI: il codice

- Scritto interamente in C
- Strutturato in moduli: serial, cannon, shared
- Ben documentato: commenti in ogni funzione
- Makefile everywhere: diverse versioni da compilare
- libreria mvapich2
- Compilatore: gcc

```
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
```

MM MPI: src/

```
format.c
format.h
reset.c
reset.h
shared.h
utils.c
utils.h
```

src/shared/

```
check.c
gendat.c
Makefile
mm.c
mxm.c
```

src/serial/

```
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
```

src/cannon/

MPI: primitive utilizzate

- MPI_Init, MPI_Finalize
- MPI_Comm_size, MPI_Comm_rank
- MPI_Barrier, MPI_Wtime
- MPI_Reduce
- MPI_Cart_create, MPI_Cart_coords, MPI_Cart_shift
- MPI_Sendrecv_replace
- MPI_Isend, MPI_Irecv, MPI_Waitall
- MPI_Comm_free

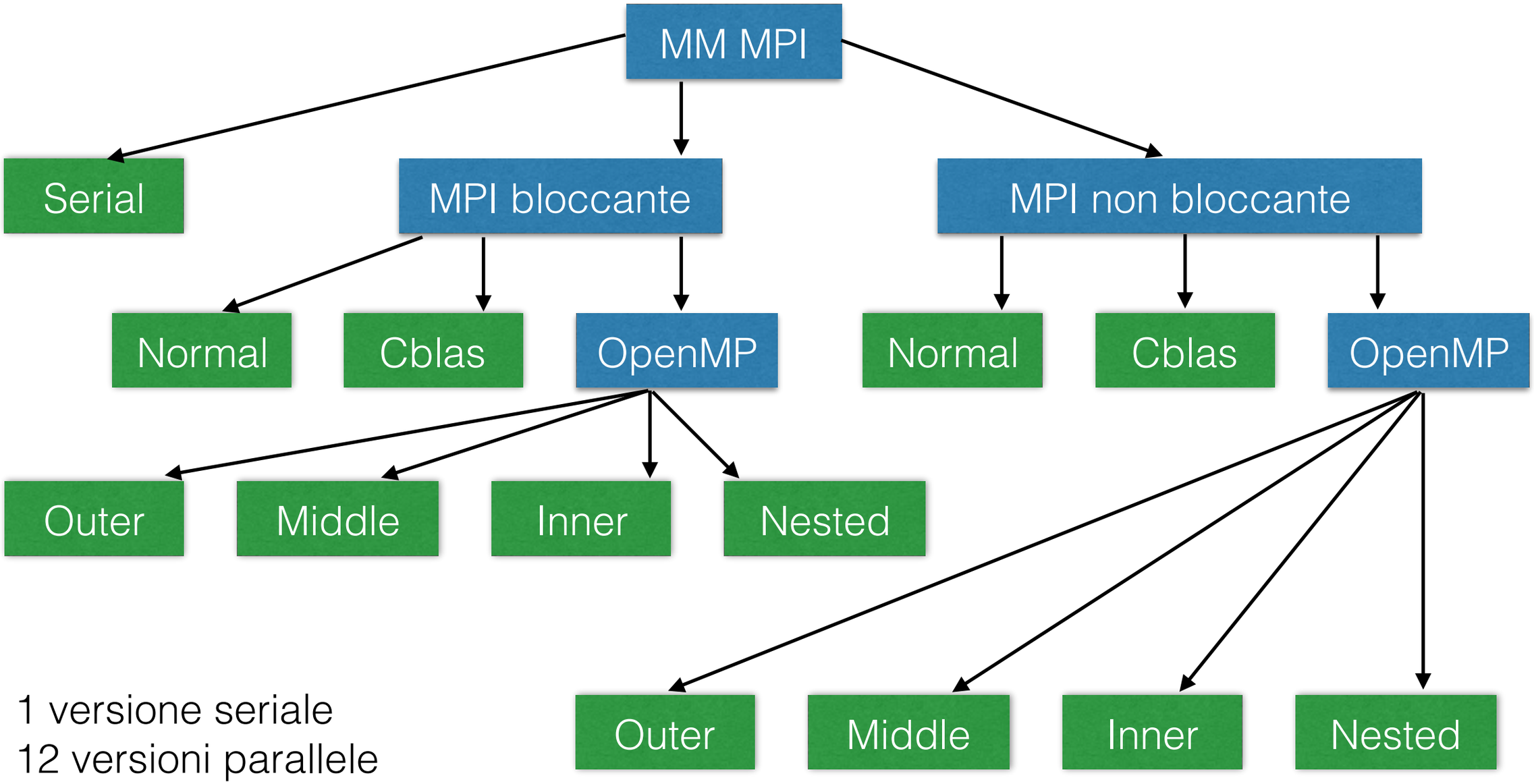
MM MPI: i Makefile

```
include ../../Makefile.mm.inc

VPATH = ../shared

OBJECTS = mm.o check.o gendat.o mxm.o mxm-local.o \
          format.o reset.o utils.o

EXEC    = x.mm

${EXEC}: clean ${OBJECTS}
  @echo
  ${LD_MPI} ${LDFLAGS} ${OBJECTS} ${LIBS} -o ${EXEC}

.SUFFIXES:  .c .o
.c.o    :
  @echo
  ${CC_MPI} ${CFLAGS} -DCBLAS ${INCS} -c -o $@ $<

clean:
  /bin/rm -f ${OBJECTS} ${EXEC}
```

Makefile.mm.inc

```
SELF_DIR := $(realpath .)

CC      = gcc-5
CC_MPI  = mpicc
INCS    = -I "$(SELF_DIR)/../shared/" -I /System/Library/Frameworks/
    Accelerate.framework/Versions/Current/Frameworks/vecLib.framework/
    Versions/Current/Headers/
CFLAGS  = -O3 -mcmodel=medium -fopenmp
LD      = $(CC)
LD_MPI  = $(CC_MPI)
LDFLAGS = $(CFLAGS) -lcblas
LIBS    = -L /System/Library/Frameworks/Accelerate.framework/Versions/
    Current/Frameworks/vecLib.framework/Versions/Current/ -pthread
```

Ottimizzazioni



- 1 versione seriale
- 12 versioni parallele

Generazione di matrici

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

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

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

Controllo del risultato

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

A

X

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

=

4	4	4	4
4	4	4	4
4	4	4	4
4	4	4	4

C

C[2,3] = A[2,0] * B[0,2] + A[2,1] * B[1,2] + A[2,2] * B[2,2] + A[2,3] * B[3,2]

C[2,3] = 1 * 1 + 2 * 0.5 + 3 * 0.33 + 4 * 0.25 = 4

MM MPI: esecuzione

Esecuzione seriale

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

Esecuzione con MPI su macchina singola

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

```
$ 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
```

Input file

Esecuzione su ChemGrid

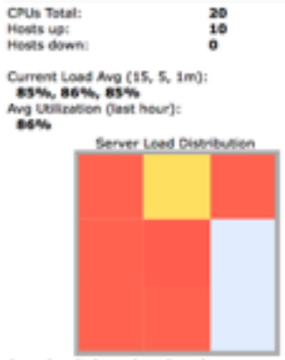
```
$ PROC_NAME=x.mm_2D_cannon
$ qsub -lnodes=8:ppn=2,mem=9050mb -d . -N $PROC_NAME -e logs -o logs -v EXE=
    $PROC_NAME,DATA=example_data run_mm_mpi_pg.sh
```

```
#!/bin/sh

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

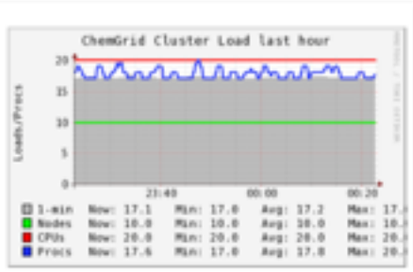
Il cluster: ChemGrid

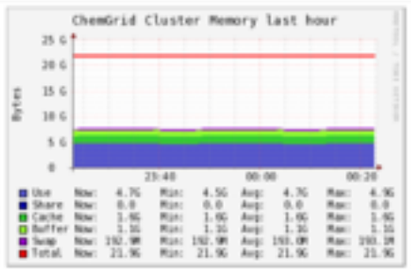
- 11 Nodi: 2 CPU 3GHz, 1GB RAM (4GB 3 nodi)
- 32 bit
- 3 nodi offline: cg03, cg06, cg11
- 8 nodi, 16 CPU disponibili
- 4, 9 16: quadrati perfetti, 9 non usabile
- 16 = 8 nodi x 2 CPU

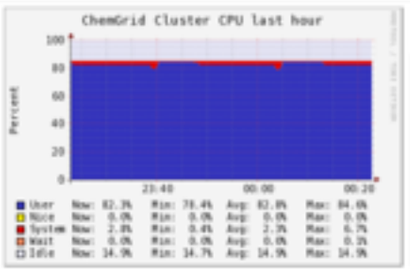
II cluster

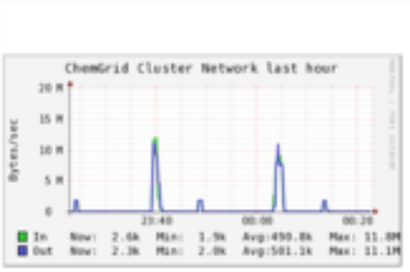


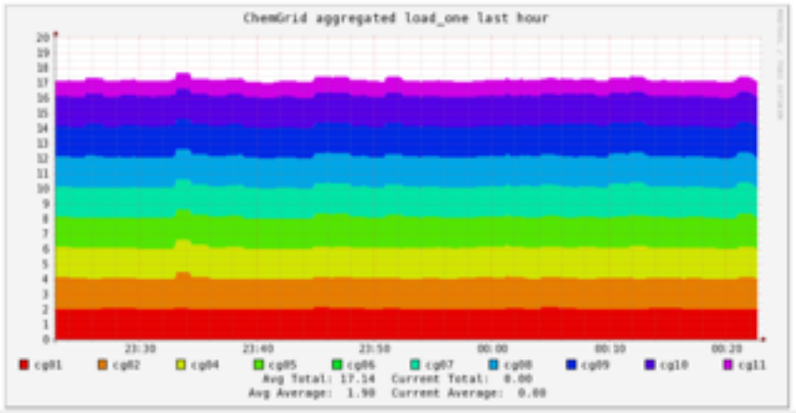
Stacked Graph - load_one







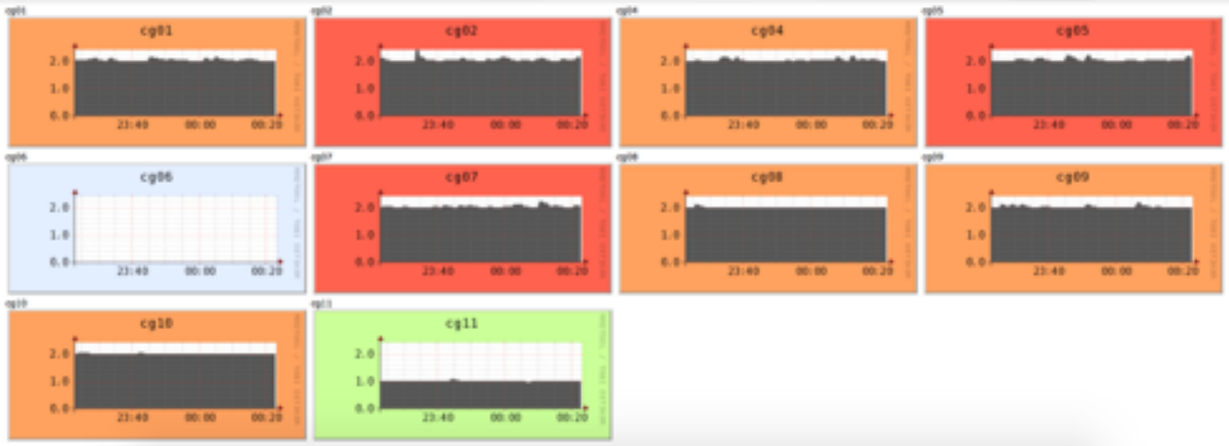




ChemGrid **load_one** last **hour** sorted **by name**

Metric load_one | Show Hosts Scaled: Auto | Same | None | Size small | Columns 4 (0 = metric + reports)

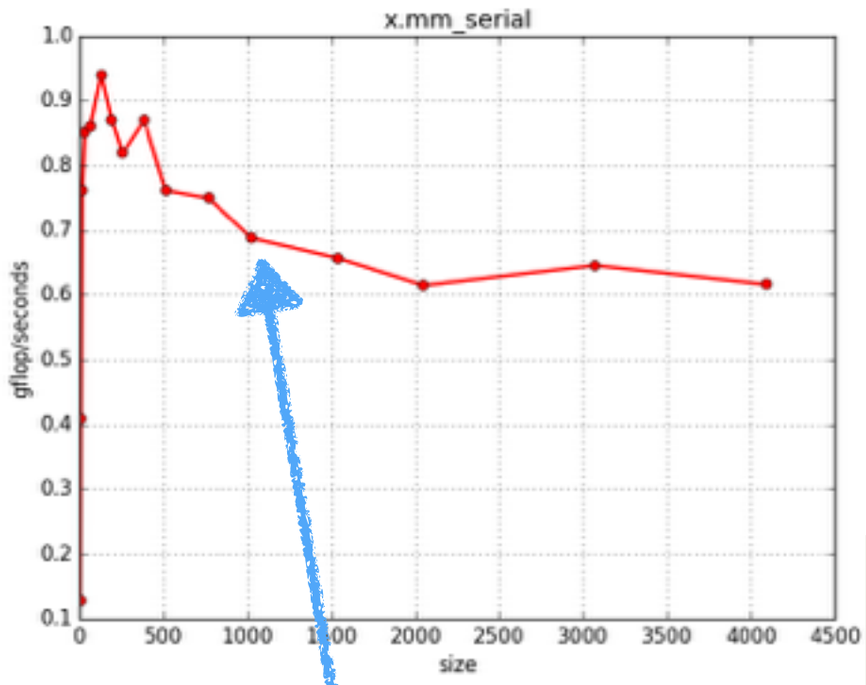
Show only nodes matching | Filter | Max graphs to show all | Sorted ascending | descending | by name



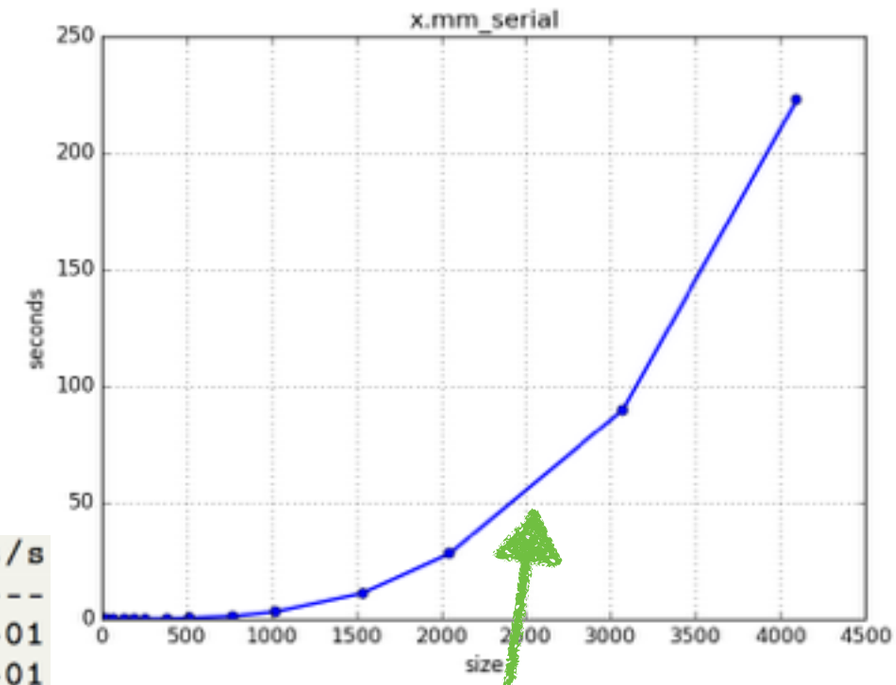
Esperimenti

- Matrici quadrate di 4, 8, 16, 32, 64, 128, 192, 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096
- Non più grandi: computazioni troppo lunghe
- 2 ripetizioni per ogni dimensione
- 4096 * 4096 * 8 = 134217728 bytes ≈ 131MB
- Gflop/s = 2 * n^3 / time * 1.0e-9

Risultato seriale



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



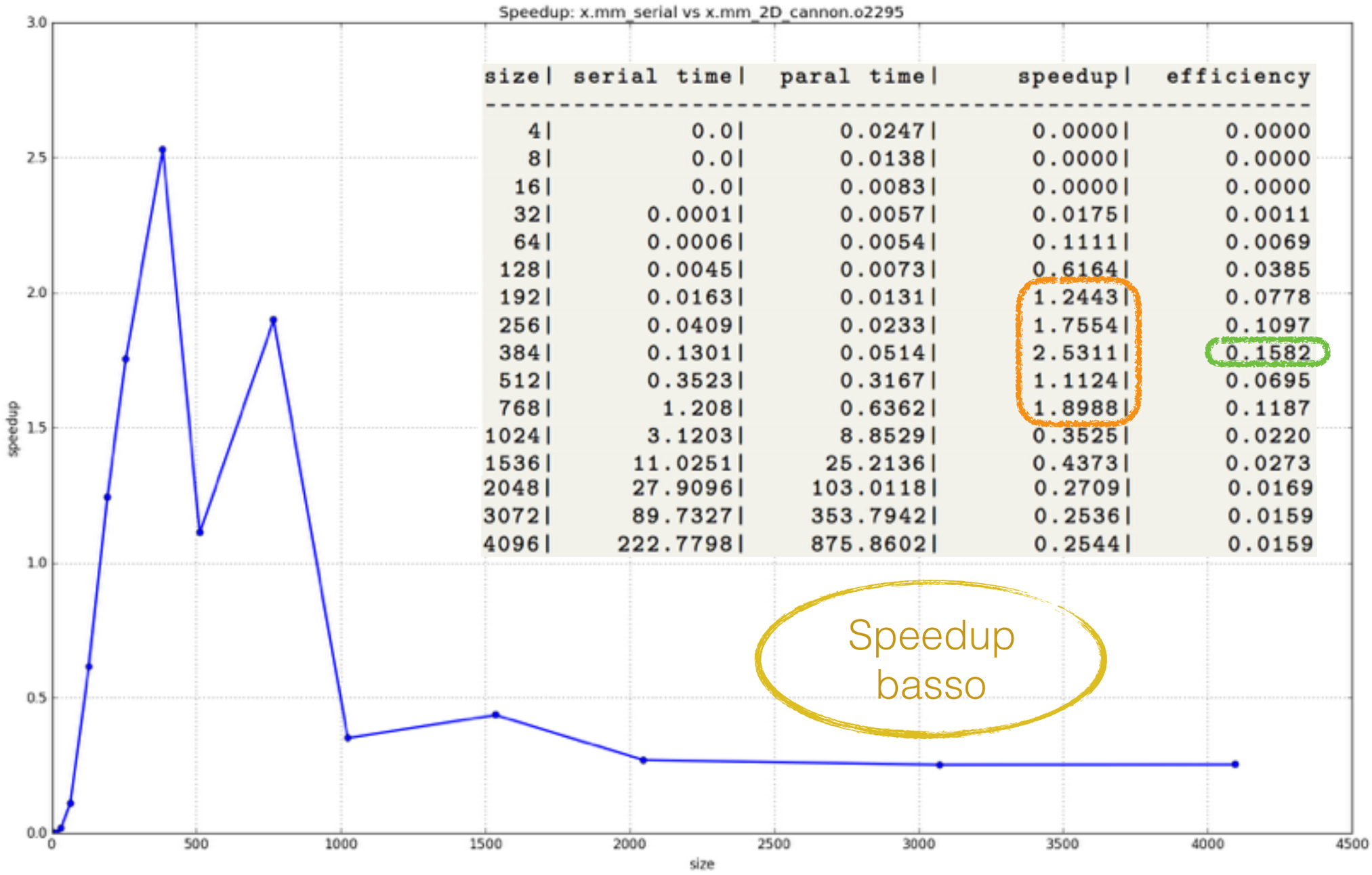
Stabile

Crescita esponenziale

Speedup ed Efficienza

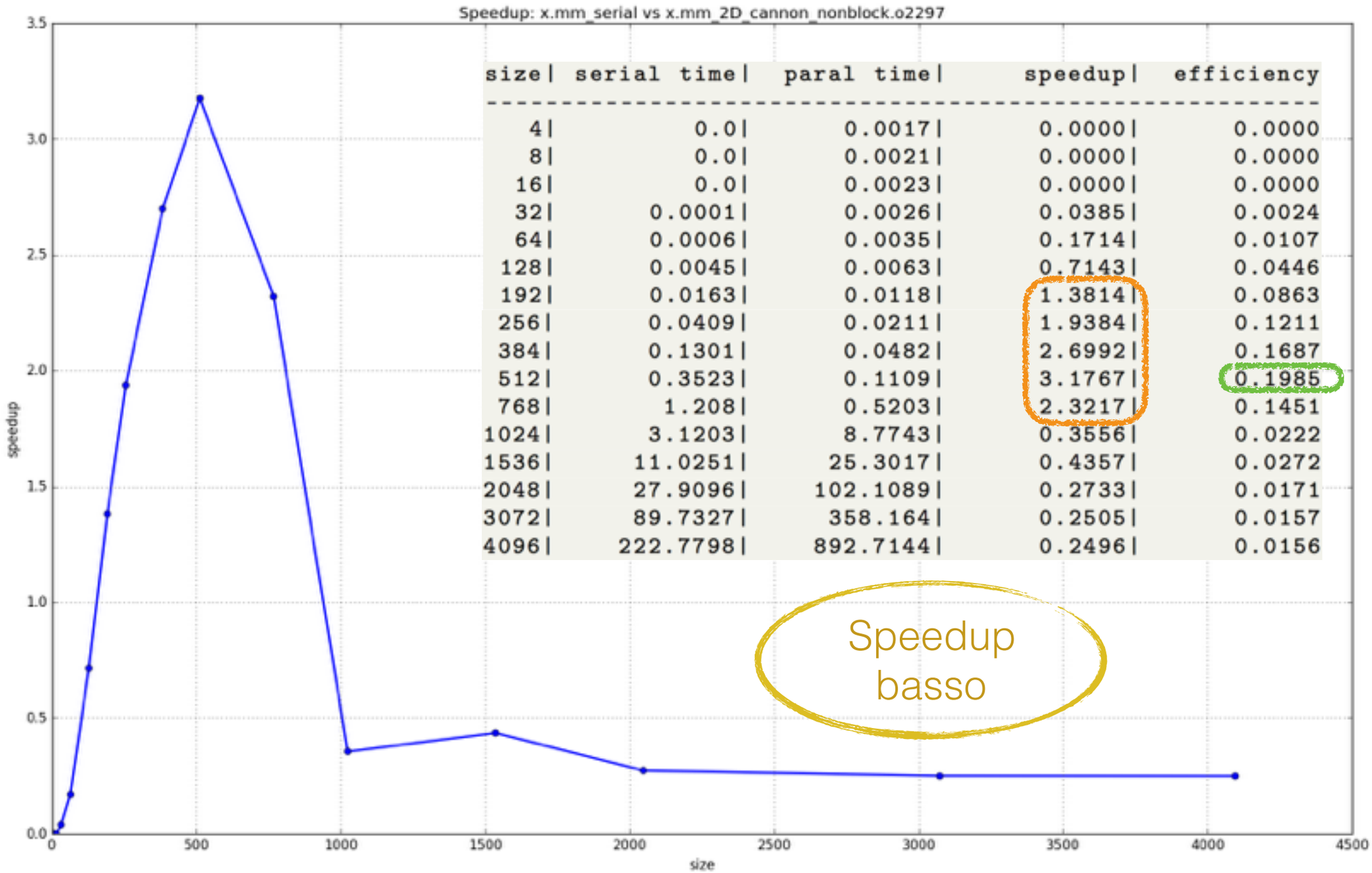
- Speedup = T(1) / T(p)
- Speedup ideale = p
- Efficienza = S(p) / p
- Efficienza ideale = 1

Risultato parallelo 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

Risultato parallelo NON bloccante

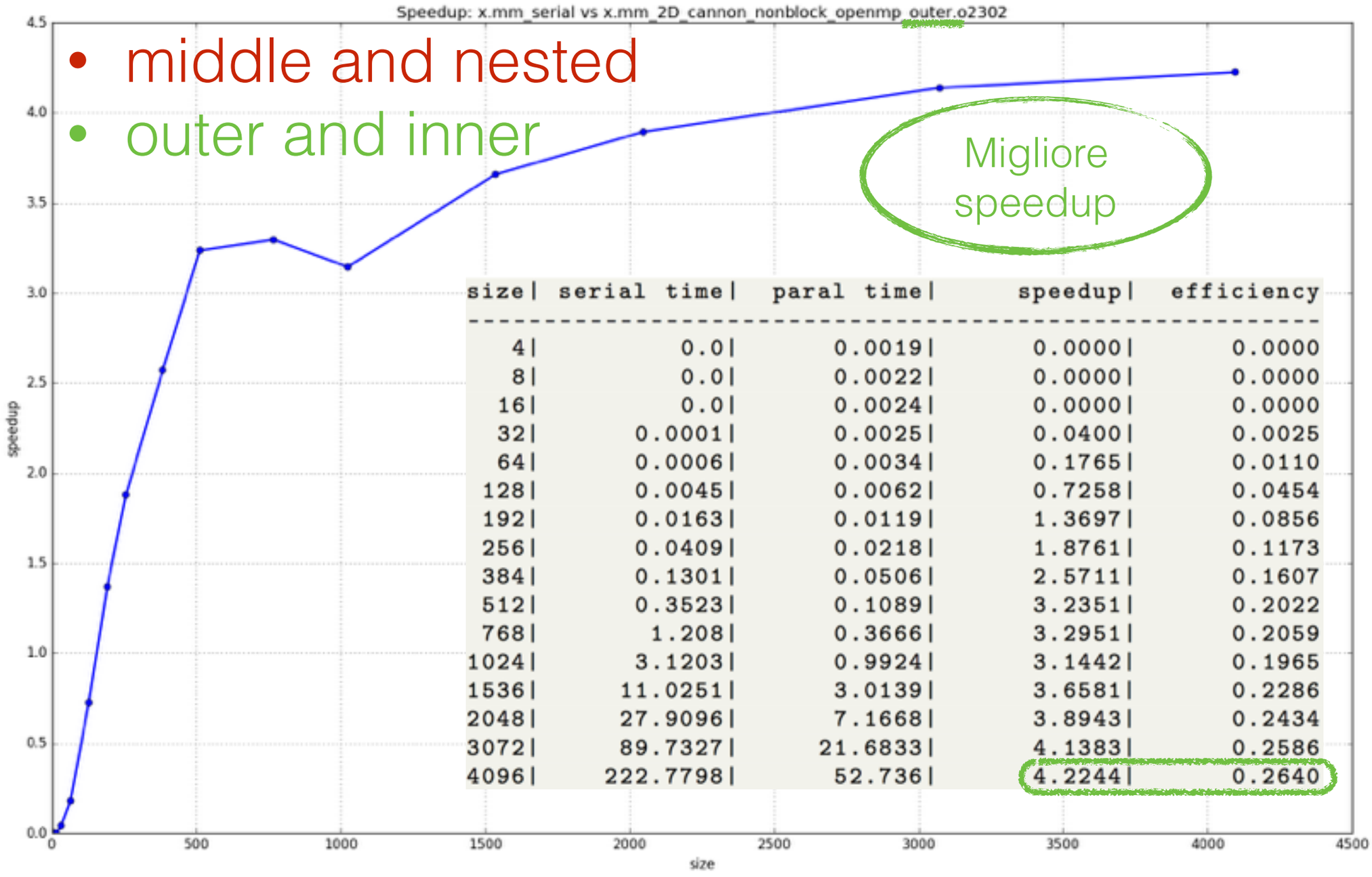


size	serial time	paral time	speedup	efficiency
4	0.0	0.0017	0.0000	0.0000
8	0.0	0.0021	0.0000	0.0000
16	0.0	0.0023	0.0000	0.0000
32	0.0001	0.0026	0.0385	0.0024
64	0.0006	0.0035	0.1714	0.0107
128	0.0045	0.0063	0.7143	0.0446
192	0.0163	0.0118	1.3814	0.0863
256	0.0409	0.0211	1.9384	0.1211
384	0.1301	0.0482	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

Speedup basso

Risultato parallelo con OpenMP

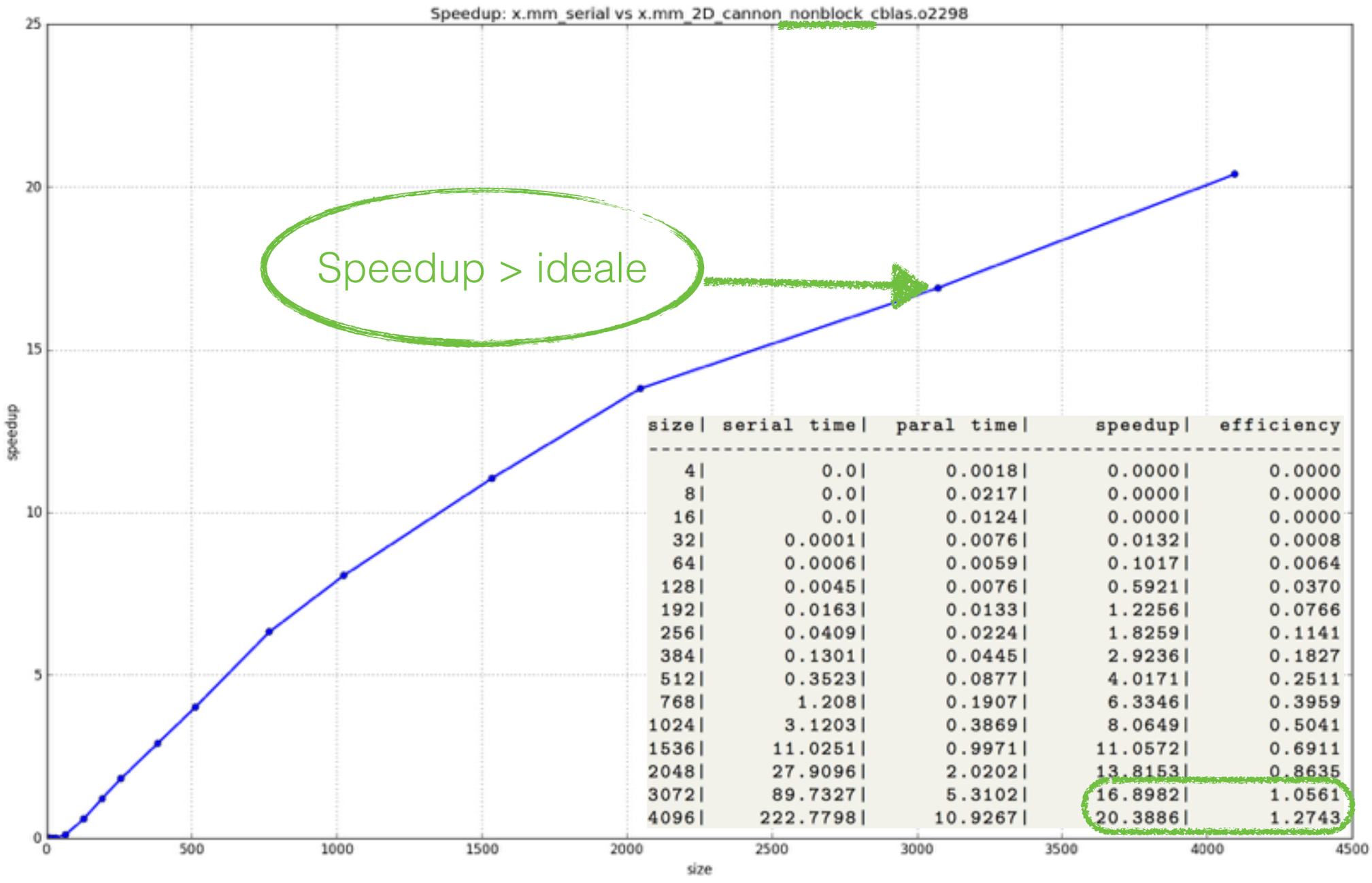
- middle and nested
- outer and inner



Migliore speedup

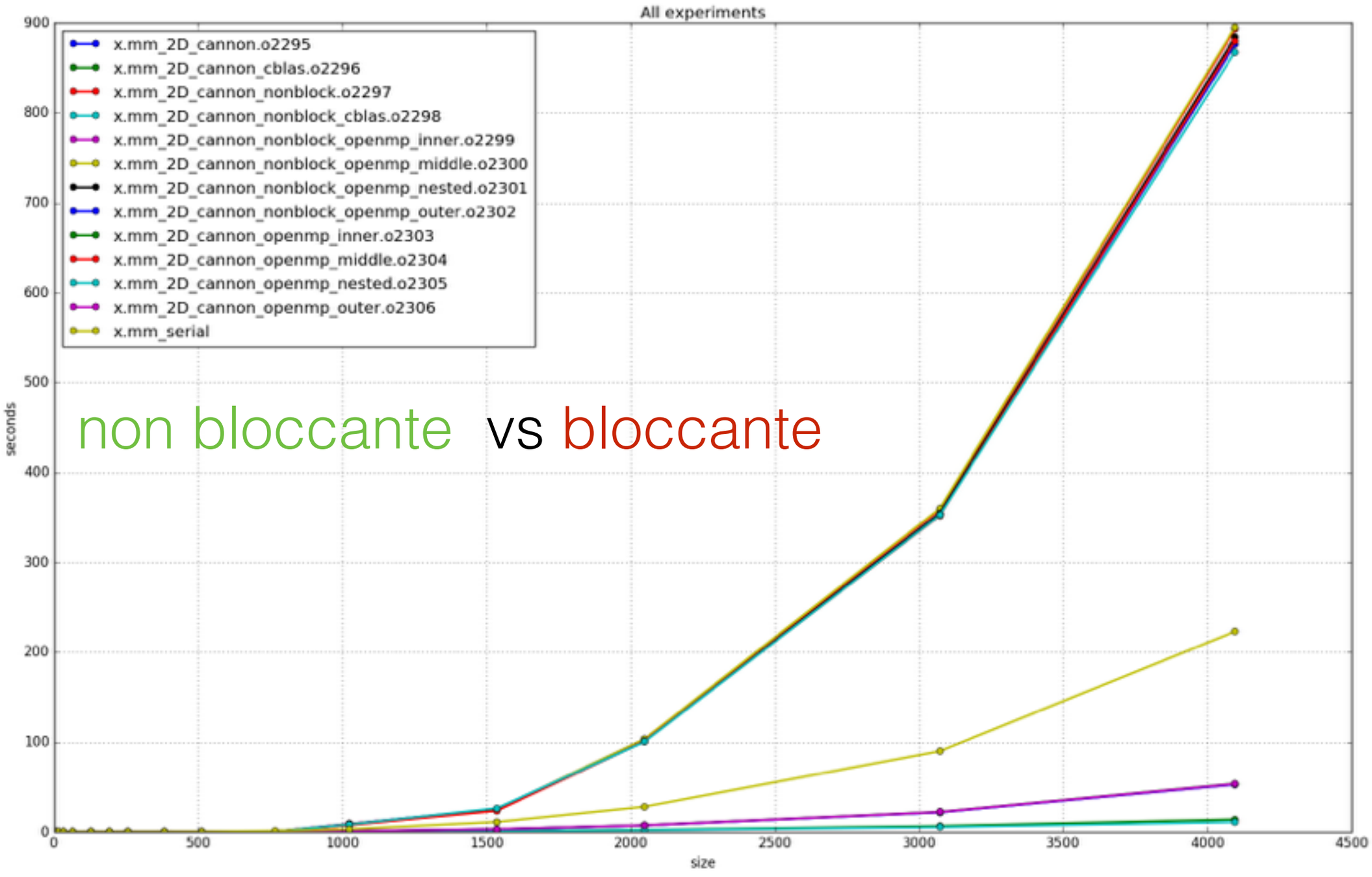
size	serial time	paral time	speedup	efficiency
4	0.0	0.0019	0.0000	0.0000
8	0.0	0.0022	0.0000	0.0000
16	0.0	0.0024	0.0000	0.0000
32	0.0001	0.0025	0.0400	0.0025
64	0.0006	0.0034	0.1765	0.0110
128	0.0045	0.0062	0.7258	0.0454
192	0.0163	0.0119	1.3697	0.0856
256	0.0409	0.0218	1.8761	0.1173
384	0.1301	0.0506	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

Risultato parallelo con CBLAS



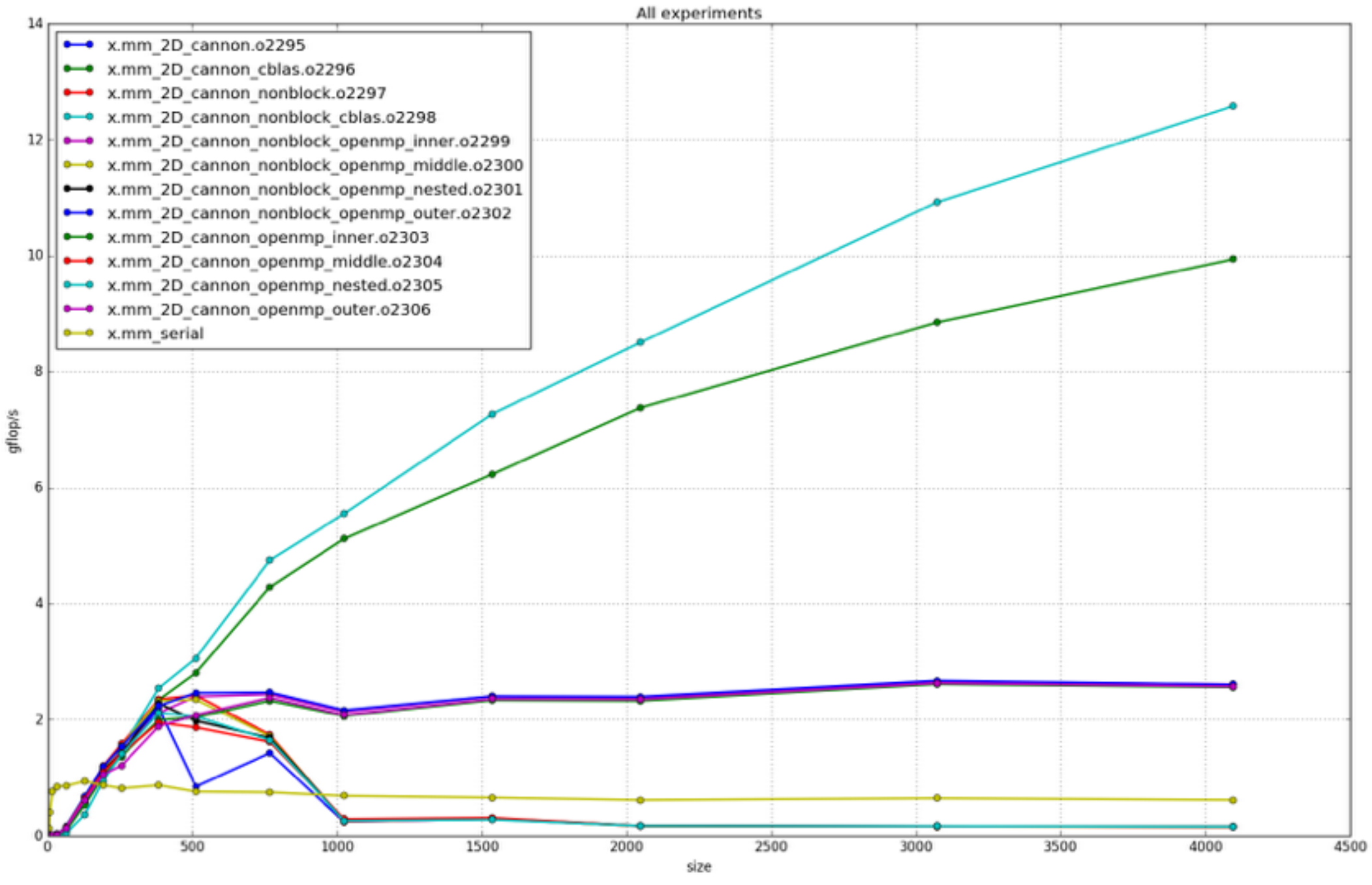
size	serial time	paral time	speedup	efficiency
4	0.0	0.0018	0.0000	0.0000
8	0.0	0.0217	0.0000	0.0000
16	0.0	0.0124	0.0000	0.0000
32	0.0001	0.0076	0.0132	0.0008
64	0.0006	0.0059	0.1017	0.0064
128	0.0045	0.0076	0.5921	0.0370
192	0.0163	0.0133	1.2256	0.0766
256	0.0409	0.0224	1.8259	0.1141
384	0.1301	0.0445	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

All experiments: time



non bloccante vs bloccante

All experiments: gflop/s



Conclusioni

- Proof of concept, risultati interessanti
- Funzionale:
  - I/O nell’applicazione
    - toggle ottimizzazioni runtime (1 binario)
- Performance:
  - OpenMP (più aggressivo)
  - OpenCL e GPU per moltiplicazioni locali