Prodotto Parallelo tra Matrici

Sistemi di Calcolo Parallelo e Applicazioni

Alessandro Lioi, 0333693

Introduzione

Introduzione - Obiettivi

Sviluppare un nucleo di calcolo parallelo per: $C \leftarrow C + AB$ con

- $A \in \mathbb{R}^{m \times k}$
- $B \in \mathbb{R}^{k \times n}$
- quindi $C \in \mathbb{R}^{m \times n}$

Tramite:

- MPI: basato su scambio di messaggi su processi diversi
- OpenMP: basato su memoria condivisa
- MPI+OpenMP: approccio ibrido

Introduzione - Metriche

- FLOPS $= \frac{2 \cdot mnk}{T}$: operazioni in aritmetica floating-point per secondi
- Speed-Up $= \frac{T_s}{T_p}$
- Relative Error $= \frac{||C_s - C_p||}{||C_s||}$ con
  - C_s: C calcolata con computazione seriale
  - C_p: C calcolata con computazione parallela
  - $||\cdot||$: norma di **Frobenius**

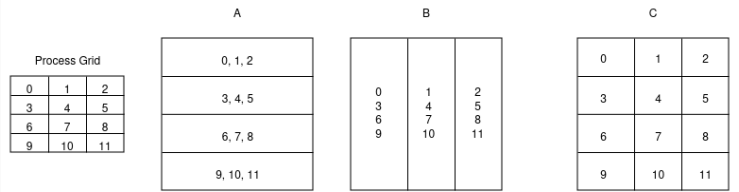
MPI: Message Passing Interface

MPI

Il programma si divide in 3 fasi:

1. Distribuzione iniziale delle matrici nei vari processi
2. Calcolo matrice locale C
3. Raccolta matrice finale C

MPI - Distribuzione Matrici

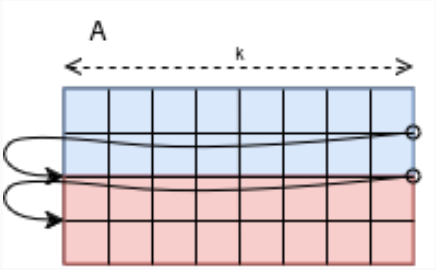


- ispirata alla distribuzione dei dati di **ScaLAPACK**
- **2D Block Distribution** sulla matrice *C*
- distribuzione il più *fair* possibile
- gestisce ogni dimensione di matrice e numero di processi, mantenendo l'errore relativo nullo

MPI - Distribuzione Matrici — Variante 1

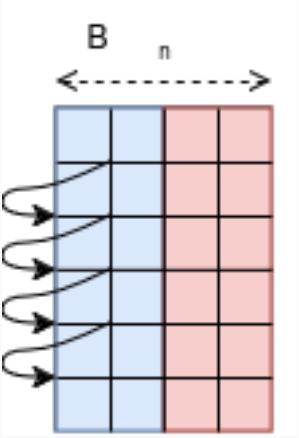
- Processo **root** genera e distribuisce le matrici ai vari processi tramite `MPI_Scatterv` e l'uso di `MPI_Datatype` custom
- Uso di un `MPI_Datatype` per le righe e uno per le colonne
- In `MPI_Scatterv` si specificano quante righe/colonne inviare e da quale riga/colonna partire
- Esempio Reale: analisi dei dati, algoritmi di machine learning centralizzati

MPI - Distribuzione Matrice A — Variante 1



- `MPI_Datatype` di tipo `vector` con:
  - numero di blocchi 1
  - lunghezza del blocco k
  - *stride* k

MPI - Distribuzione Matrice B — Variante 1

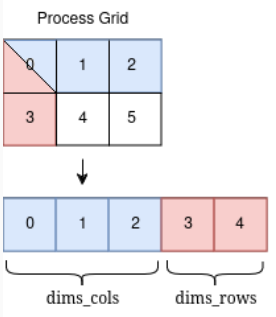


- `MPI_Datatype` di tipo `vector` con:
  - numero di blocchi k
  - lunghezza del blocco 1
  - *stride* n

MPI - Distribuzione Matrici — Variante 2

- Processo **root** genera e distribuisce i *seed* per ogni parte della matrice
- Ogni processo si genera le proprie sotto-matrici A e B locali
- Esempio Reale: simulazioni in ambito scientifico; ogni blocco di una matrice è generata a partire da delle equazioni

MPI - Distribuzione Matrici — Variante 2



- Processo **root** genera `dims_rows` + `dims_cols` *seed*
- Tramite `MPI_Scatter` invia ad ogni processo il seed corrispondente all'indice di riga e colonna
  (e.g. processo 1 riceve seeds 1 e 3)

MPI - Calcolo e Raccola Risultati

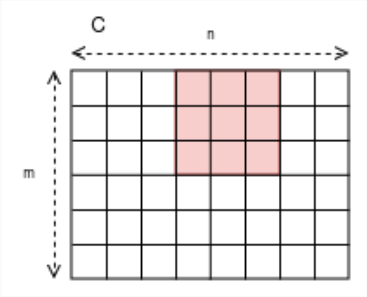


Figure 1: Matrice calcolata dal processo 1 dall'esempio precedente

- Ogni processo calcola la sua parte di C
- Si raccolgono i risultati con `MPI_Allreduce`
  - Si usa `MPI_IN_PLACE` per non allocare un buffer aggiuntivo

OpenMP

OpenMP

```
// matrix_transpose in matrix.c
  int i, j;
#pragma omp parallel for
          private(i, j)
          shared(source, transposed)
  for (j = 0; j < cols; j++)
#pragma omp simd
    for (i = 0; i < rows; i++)
      transposed[j * rows + i] =
          source[i * cols + j];
```

- Una variante: con il modello shared-memory, le matrici sono già disponibili per tutti i threads (sia A e B che C)
- Si esegue il nucleo di calcolo parallelo che richiede l'esecuzione della trasposta di B
  - sezione parallela con variabili:
    - private per `i`, `j` (variabili di controllo dei loop)
    - condivise per le matrici `source` e `transposed`
  - sezione SIMD per usare le istruzioni vettoriali

MPI+OpenMP

MPI+OpenMP

```
add_executable(scpa-mpi-omp-v1
        src/mpi/main-v1.c ...)
add_executable(scpa-mpi-omp-v2
        src/mpi/main-v2.c ...)

target_link_libraries(scpa-mpi-omp-v1
        PUBLIC
        MPI::MPI_C
        OpenMP::OpenMP_C m)
target_link_libraries(scpa-mpi-omp-v2
        PUBLIC
        MPI::MPI_C
        OpenMP::OpenMP_C m)
```

- si utilizzano le versioni di MPI
- si compila abilitando anche OpenMP

Nucleo di Calcolo Parallelo

```
  int i, j, l, ii, jj, ll;
#pragma omp parallel for private(i, j, l, ii, jj, ll) shared(a, b, c) collapse(3)
  for (i = 0; i < sub_m; i += 16)
    for (j = 0; j < sub_n; j += 16)
      for (l = 0; l < k; l += 16)
        // Block multiplication
        for (ii = i; ii < MIN(i + 16, sub_m); ++ii)
          for (jj = j; jj < MIN(j + 16, sub_n); ++jj) {
            float sum = 0;
#pragma omp simd reduction(+ : sum)
            for (ll = l; ll < MIN(l + 16, k); ll++)
              sum += a[ii * k + ll] * b[jj * k + ll];
            c[(ii + row_offset) * n + (jj + col_offset)] += sum;
          }
```

Nucleo di Calcolo Parallelo - Loop Esterni

```
int i, j, l, ii, jj, ll;
#pragma omp parallel for private(i, j, l, ii, jj, ll) shared(a, b, c) collapse(3)
for (i = 0; i < sub_m; i += 16)
  for (j = 0; j < sub_n; j += 16)
    for (l = 0; l < k; l += 16)
```

- ogni processo/thread esegue un prodotto a blocchi di dimensione $16 = \frac{64}{4}$
  - 64 bytes: dimensione linea di cache
  - 4 bytes: dimensione `float`
- sezione parallela con:
  - variabili di controllo dei loop *private*
  - variabili delle matrici *condivise*
  - `collapse(3)` per unire i cicli esterni; ogni thread si occupa quindi di un blocco

Nucleo di Calcolo Parallelo - Loop Interni

```
      // Block multiplication
      for (ii = i; ii < MIN(i + 16, sub_m); ++ii)
        for (jj = j; jj < MIN(j + 16, sub_n); ++jj) {
          float sum = 0;
#pragma omp simd reduction(+ : sum)
          for (ll = l; ll < MIN(l + 16, k); ll++)
            sum += a[ii * k + ll] * b[jj * k + ll];
          c[(ii + row_offset) * n + (jj + col_offset)] += sum;
        }
```

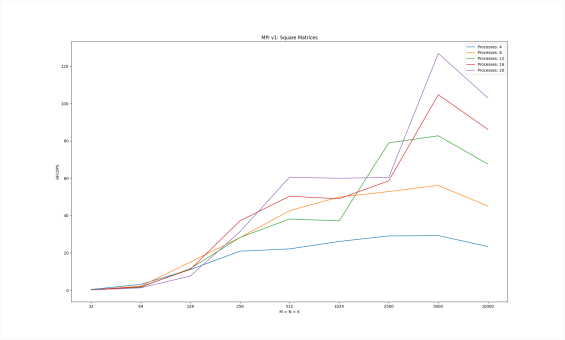
- calcolo effettivo del valore di $C_{ii,jj}$
- uso della variabile temporanea `sum` per ridurre gli accessi in C
- sezione di OpenMP `SIMD` con funzione di riduzione `sum` per usare istruzioni vettoriali
- `row_offset` e `col_offset` impostati da MPI per scrivere solo sul blocco di interesse di C

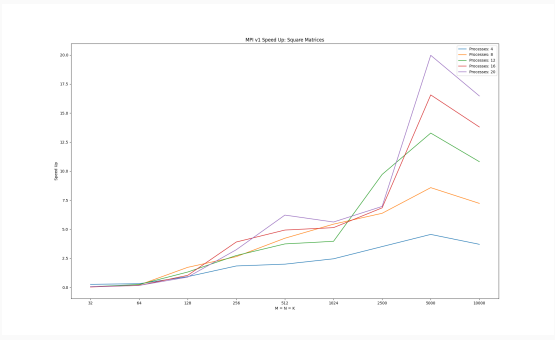
Nucleo di Calcolo Parallelo - Considerazioni Aggiuntive

- se possibile, si allocano le matrici con la funzione `aligned_alloc` con dimensione 16
- si usa la keyword `restrict` nella segnatura della funzione del kernel di calcolo
- opzioni di compilazione:
  - `-O3`
  - `-march=native`
  - `-ffast-math`

Performance

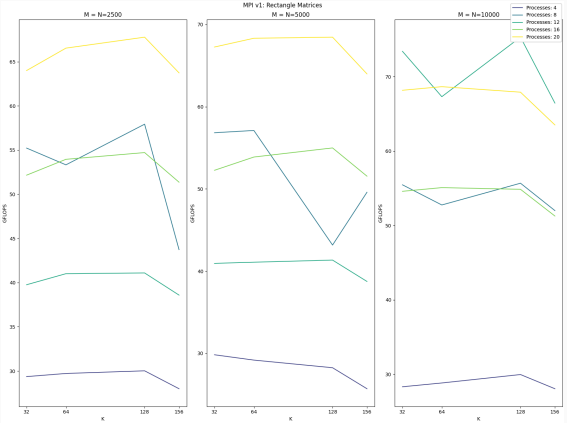
MPI Performance - Matrici Quadrate

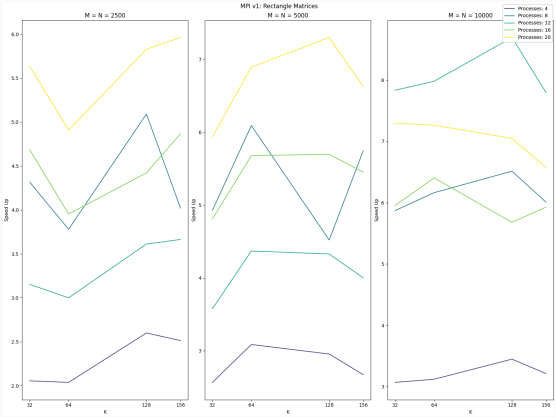




- Picco di 120 GFLOPS in $M = N = K = 5000$
- Speed Up massimo in $M = N = K = 5000$

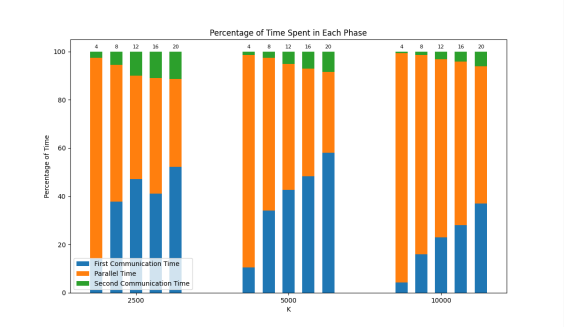
MPI Performance - Matrici Rettangolari

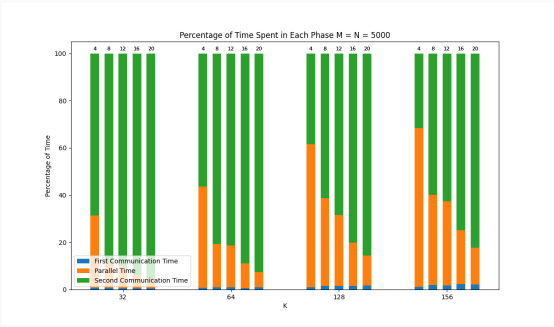




- Picco di quasi 70 GFLOPS in $M = N = 5000$ e $K = 128$
- situazione analoga al caso quadrato

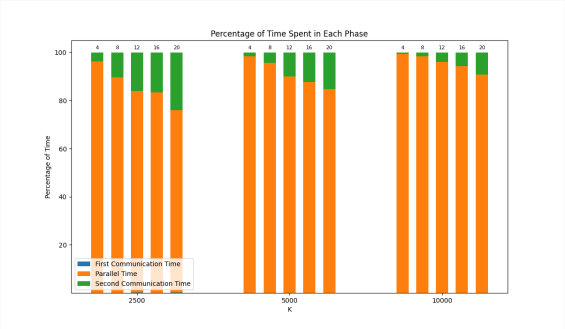
MPI v1 Performance - Distribuzione Tempo

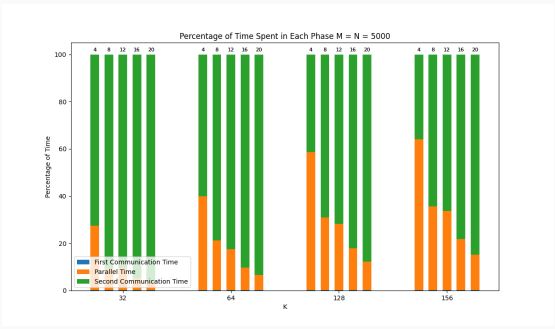




- Caso quadrato: maggior parte del tempo usato per inviare le matrici iniziali
- Caso rettangolare: raccolta dei risultati più pesante
  - comunicazione iniziale e computazione parallela più brevi
  - comunicazione finale con gli stessi tempi del caso quadrato

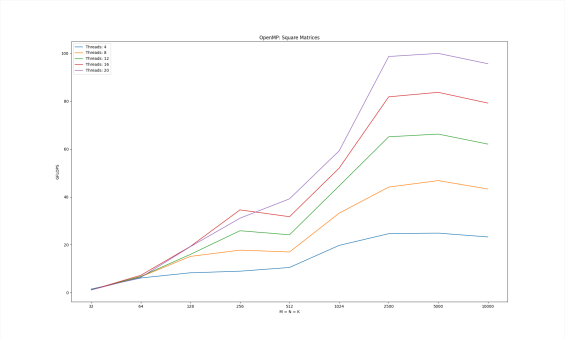
MPI v2 Performance - Distribuzione Tempo

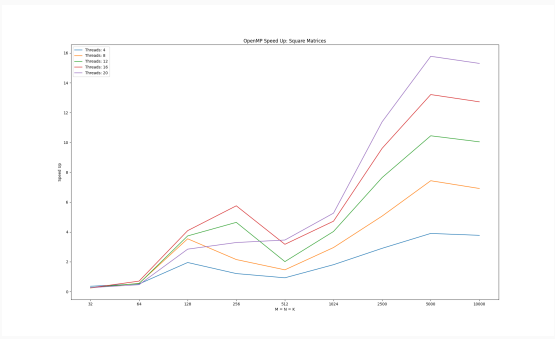




- Comunicazione iniziale praticamente nulla
- Caso rettangolare: comunicazione finale sembra essere più pesante, ma valgono le stesse considerazioni della variante 1

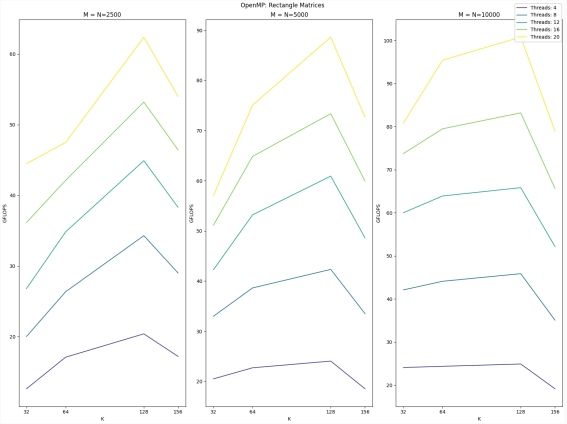
OpenMP Performance - Matrici Quadrate

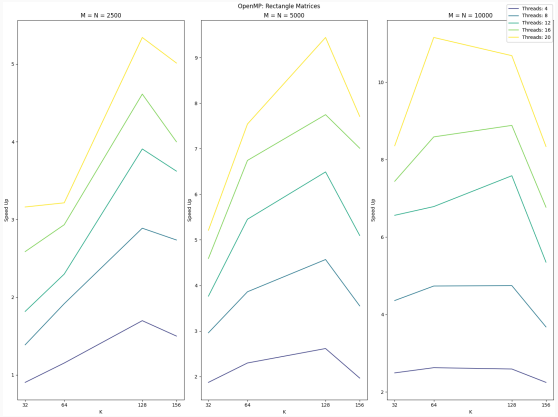




- Picco dei GFLOPS e Speed Up in $M = N = K = 5000$
- Calo delle performance in $M = N = K = 512$:
  - eseguendo con `callgrind` e `perf` non risultano anomalie particolari sull'uso di memoria/cache
  - aumentando la dimensione del blocco a 32, si ha un aumento delle performance

OpenMP Performance - Matrici Rettangolari

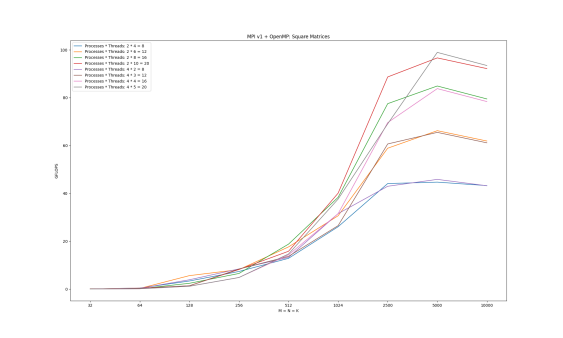


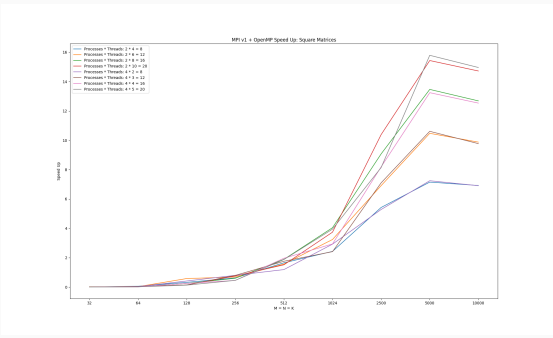


Situazione analoga al caso quadrato

MPI+OpenMP Performance - Matrici Quadrate

Configurazione con 2 o 4 processi e numero di thread variabili in modo tale da avere $\texttt{process} \cdot \texttt{thread} = 20$

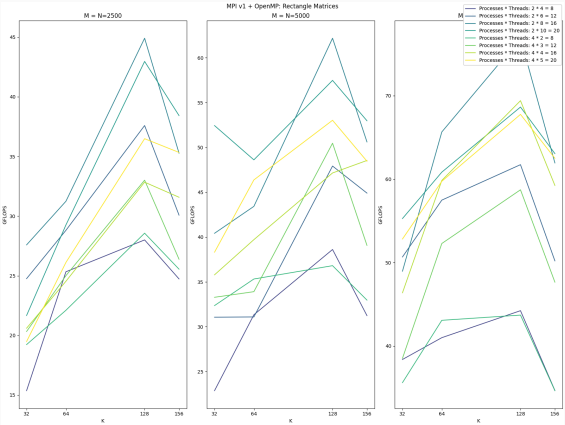


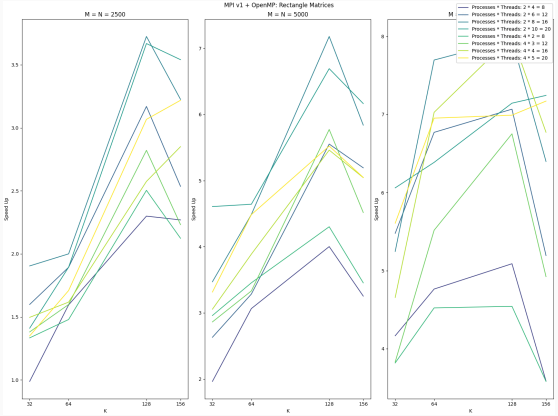


Picco delle performance con la configurazione con 4 processi e 5 threads

MPI+OpenMP Performance - Matrici Rettangolari

Configurazione con 2 o 4 processi e numero di thread variabili in modo tale da avere $\texttt{process} \cdot \texttt{thread} = 20$





Picco delle performance con la configurazione con 2 processi e 10 threads

Grazie Per l'Attenzione!

Il codice è presente su GitHub al seguente link:
https://github.com/lioia/scpa