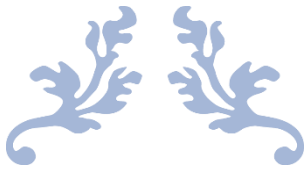


GUIDA COMPLETA PROGETTI HPC

Progetti anno 2023

UNIVERSITA' DEGLI STUDI DI PARMA

Laboratorio: Ambiente del cluster HPC.unipr.it

Esercizi timing e plotting :

Consegna :
Creare la directory time-plot e copiare i file indicati:
mkdir -p ~/HPC2223/time-plot
cp /hpc/group/T_2022_HPCPROGPAR/time-plot/* ~/HPC2223/time-plot/
- Generare i plot **IMB_thr.png networks_barplot.png IMB_lat.png**

Svolgimento :
Eseguire i due comandi della consegna, aprire il file "**IMB_plot.py**", sostituire le parole <nome> e <data> nelle righe 12 e 22. Una volta fatto generate le due immagini contenente un grafico con il comando "**python IMB_plot.py**". Infine, generare l'ultima immagine rimasta contenente il grafico con il comando "**python networks_barplot.py**". Dal codice di questi due file si nota che :
1) Vengono utilizzate due librerie in questi due file python :

- **pandas** https://pandas.pydata.org/ è una libreria software scritta per il linguaggio di programmazione Python per la manipolazione e l'analisi dei dati.
- **matplotlib** -> https://matplotlib.org/ è una libreria per la creazione di grafici per il linguaggio di programmazione Python progettata per assomigliare a quella di MATLAB.-

2) Il primo file python legge la tabella contenuta nel file "**IMB.dat**" e genera due grafici :

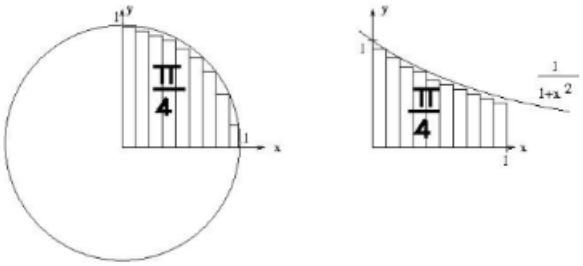
- Il primo grafico ha l'attributo "bytes" sull'asse X e l'attributo "MBPS" sull'asse Y. Quindi questo grafico mostra il **throughput** come si vede anche dal titolo del grafico generato.
  Sustained Performance (Throughput) : Prestazioni effettive misurate, di un componente hardware o di un sistema di calcolo tramite l'esecuzione di specifici programmi **(Benchmark)**.
- Il secondo grafico ha l'attributo "bytes" sull'asse X e l'attributo "times" (espresso in us) sull'asse Y. Quindi questo grafico mostra la **latenza** come si vede anche dal titolo del grafico generato.
  Latenza : E' il tempo che intercorre tra l'invio di un'informazione e la ricezione della risposta. In altre parole, è il tempo che impiega un sistema informatico a elaborare una richiesta e a restituire un risultato. La latenza è un fattore importantissimo per il calcolo parallelo perché c'è continuo scambio di piccoli dati.

3) Il secondo file python legge la tabella contenuta nel file "**network.dat**" e genera un grafico a barre contente un confronto di prestazioni tra 4 dispositivi differenti (infatti sull'asse X ci sono i nomi dei dispositivi) :

	Bitrate (Gb/s)	Bandwidth (GB/s)	Latency (microsec.)	Costo scheda (K€)
GbEthernet (tcp/ip)	1	0.1	47	0.03
10GbEthernet (tcp/ip)	10	0.9	13	0.1
Intel OmniPath	100	12	1	1
Infiniband Mellanox EDR	100	12	1	1

cpi.c :

Consegna :
Il programma calcola pigreco utilizzando due diversi metodi di integrazione numerica.
Per il calcolo dell'integrale viene utilizzato il metodo dei rettangoli nell'intervallo tra 0 e 1.



Compilare ed eseguire il programma cpi.c localmente:
gcc cpi.c -o cpi -lm -O2
./cpi -h
Individuate nel programma le funzioni f1() e f2() che implementano i due metodi, la funzione getopt() per la gestione dell'input dei parametri (selezione della funzione e del numero di intervalli) .
bash cpi1.bash
bash cpi2.bash
Eseguire il programma utilizzando il cluster HPC:
sbatch cpi1.slurm
sbatch cpi_scaling.slurm
hpc-squeue # per verificare lo stato del Job

Al termine dell'esecuzione nella directory di lavoro si troveranno stdout e stderr nel file <nomeprogramma>.o<jobid> e i file generati del programma.
Con i risultati ottenuti generare il plot cpi_scaling.png, adattando nome e data.

Svolgimento:

Questo programma in C implementa due funzioni per il calcolo del pi greco, la funzione f1 implementa la funzione matematica 1 / (1 + x^2), mentre la funzione f2 implementa la funzione matematica sqrt(1 - x^2), dove sqrt è la radice quadrata. Di default viene chiamata la funzione f1, per chiamare f2 usare il comando 2.

```
double f1( double x ) {
  return (1.0 / (1.0 + x*x));
}

double f2( double x ) {
   return (sqrt(1-x*x) );
}
```

Eseguire quindi i seguenti comandi :

- “gcc cpi.c -o cpi -lm -O2” -> Comando in cui usa la libreria gcc per compilare il programma in c e creare un eseguibile di nome cpi.
- “./cpi -h” -> Mostra un messaggio di aiuto tramite il comando h, in questo caso a cosa servono i vari comandi per questo programma.
- “bash cpi1.bash” -> Esegue lo script bash contenuto in questo file, in cui viene avviato il programma C con la funzione f1 con 1000000000 passato come parametro n, che è il numero di volte in cui viene chiamata la funzione e mostra i risultati.

- “bash cpi2.bash” -> Esegue lo script bash contenuto in questo file, in cui viene avviato il programma C con la funzione f2 con 1000000000 passato come parametro n, che è il numero di volte in cui viene chiamata la funzione e mostra i risultati.

Dai vari test, avviando più volte questi due script, si nota che il programma, quando si usa la funzione f1() è circa un secondo più veloce rispetto a quando si usa la funzione f2()

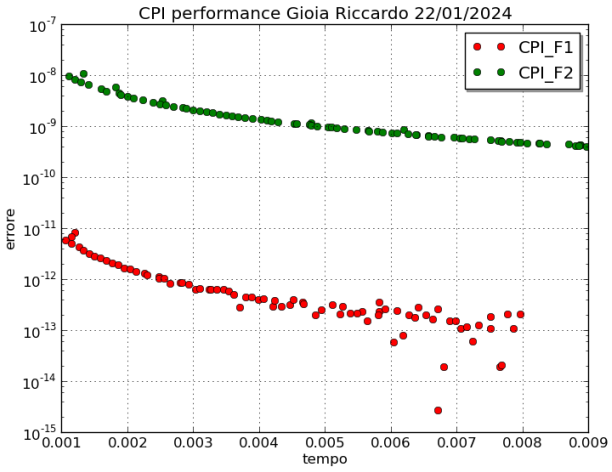
Ora dobbiamo eseguire il programma C utilizzando il cluster HPC con i seguenti script SLURM (**N.B. L'utilizzo di SLURM è consigliato per distribuire lavoro su cluster di computer in ambienti ad alte prestazioni, mentre l'avvio tramite Bash è più adatto per operazioni più semplici o in sistemi locali**) :

- “sbatch cpi1.slurm” -> Avvia uno script SLURM in cui viene avviato il programma (solo con la funzione f1() di default), con 1000000000 passato come parametro n, che è il numero di volte in cui viene chiamata la funzione e mostra i risultati .
- “sbatch cpi_scaling.slurm” -> Avvia uno script SCLURM contente un ciclo for che itera attraverso una sequenza di valori di N da 100.000 a 900.000 con un passo di 10.000. Questo significa che il programma cpi verrà eseguito più volte con diversi valori di N, che è il numero di volte in cui viene chiamata la funzione sia f1 che f2, per valutare le prestazioni al variare della dimensione del problema e infine registra i risultati in un file CSV di nome “cpi_scaling.csv”.
- “hpc-squeue” -> per verificare lo stato dei Job avviati nei due comandi precedenti.

Al termine dell'esecuzione nella directory di lavoro si troveranno stdout e stderr nel file <nomeprogramma>.o<jobid> e i file generati del programma. Quindi aggiornare i nuovi file generati da MobaXterm e il file CSV aggiornato. **Dal file CSV si nota anche qui che la funzione f1() è più veloce**.

Infine :

- Modificare nel file “cpi_scaling_plot.py” l’attributo “nome” e “data” nella riga 13.
- Avviare lo script python che genera il plot “cpi_scaling.png” tramite il comando “python cpi_scaling_plot.py”.
  Questo script estrae i dati dal CSV e li filtra in due array, uno con i dati ricavati tramite la funzione f1() e l’altro con i dati ricavati tramite la funzione f2() e genera un grafico contenente entrami gli array generati. Da questo grafico, nell’asse X c’è il tempo impiegato mentre nell’asse Y c’è l’errore.



Da questo grafico generato concludo che la funzione f1 ha un margine di errore più basso ed è leggermente più veloce, tuttavia la funzione f2() è più costante in margine di errore per ogni ciclo, mentre f1() è più altalenante. Poi copiare tutto tramite questo comando : “rsync -av --chmod=D755,F644 ~/HPC2223/ ${USER}@studenti.unipr.it@didattica-linux.unipr.it:html/HPC2223/”

Laboratorio: Performance

Consegna :

GPU

La distribuzione CUDA include programmi per testare le GPU., tra cui:
samples/1_Utilities/deviceQuery
samples/1_Utilities/bandwidthTest
samples/5_Simulations/nbody

Vedi sul cluster tutti i samples della distribuzione CUDA 11.5.2: /hpc/share/tools/cuda/11.5.2/samples/

Eseguire i 3 benchmarks su una GPU P100, una A100, e eventualmente una V100, quindi confrontate in una tabella le caratteristiche delle GPU nel file GPU.txt:
- memoria globale, L2 cache size, GPU max clock rate, cuda cores
- bandwidth host to device , device to host, device to device
- GFLOP/s in doppia precisione.

Svolgimento:

- Usare questo comando per copiare la cartella “perf” -> “rsync -av /hpc/group/T_2022_HPCPROGPAR/perf/ ~/HPC2223/perf/”
- entrare dentro alla cartella “CUDA” -> “cd HPC2223/perf/CUDA”
- Entrare in tutte e tre le cartelle ed eseguire i file con estensione SLURM per tutte e tre le tipologie di CPU (A100, P100 e V100), se capita che non c’è il file SLURM per qualcuna delle tipologie aggiungetelo voi o modificatelo e avviatelo, in modo di avere i risultati per tutte le GPU.
- Costruire un file di testo di nome “GPU.txt” contenente i risultati di tutti i file generati dagli script SLURM su tutte e tre le architetture e i loro risultati su :
  - memoria globale, L2 cache size, GPU max clock rate, cuda cores
  - bandwidth host to device , device to host, device to device
  - GFLOP/s in doppia precisione.

```
# Device Query

+-----------------------+-----------+------------+-----------+
| CARATTERISTICHE \ GPU |    P100   |    A100    |    V100   |
+-----------------------+-----------+------------+-----------+
|    MEMORIA GLOBALE    | 12184 MB  |  81085 MB  |  32501 MB |
+-----------------------+-----------+------------+-----------+
|     L2 CACHE SIZE     | 3145728 B | 41943040 B | 6291456 B |
+-----------------------+-----------+------------+-----------+
|   GPU MAX CLOCK RATE  |  1.33 GHz |  1.41 GHz  |  1.38 GHz |
+-----------------------+-----------+------------+-----------+
|       CUDA CORES      |    3584   |    6912    |    5120   |
+-----------------------+-----------+------------+-----------+

# Bandwidth Test

+------------------+-----------+-----------+------------+
|  BANDWIDTH \ GPU |    P100   |    A100   |    V100    |
+------------------+-----------+-----------+------------+
|  HOST TO DEVICE  | 11.8 GB/s | 26.1 GB/s |  13.1 GB/s |
+------------------+-----------+-----------+------------+
|  DEVICE TO HOST  | 13.1 GB/s | 26.3 GB/s |  13.6 GB/s |
+------------------+-----------+-----------+------------+
| DEVICE TO DEVICE |  382 GB/s | 1300 GB/s | 726.4 GB/s |
+------------------+-----------+-----------+------------+

# NBody

+--------------------------+----------+----------+----------+
|          \ GPU           |   P100   |   A100   |   V100   |
+--------------------------+----------+----------+----------+
| Double-precision GFLOP/s | 2812.183 | 7469.012 | 5221.432 |
+--------------------------+----------+----------+----------+
```

Consegna :

IMB Benckmarks

Il benchmark IMB di Intel può eseguire diversi tipi di test di performance in un cluster HPC.
Git: https://github.com/intel/mpi-benchmarks
Userguide: https://www.intel.com/content/www/us/en/develop/documentation/imb-user-guide/top.html

- Scaricare il tar mpi-benchmarks-IMB
- compilare i sorgenti con il compilatore intel

module load intel impi
make IMB-MPI1
make IMB-IO
Provare i seguenti benchmark sulle performance di comunicazione e di I/O su storage:

Comunicazione (Benchmank IMB-MPI1)

Gli script IMB-MPI1-n2.slurm e IMB-MPI1-n1.slurm determinano i tempi nella comunicazione tra due processi utilizzando il benchmark IMB-MPI1

- sullo stesso nodo (n1), verificando quindi la comunicazione via shared memory (SHM)
- su 2 nodi distinti (n2), connessi da rete Intel OmniPath Architecture (OPA)
- su 2 nodi distinti (n2), connessi da rete Infiniband (IB)

Input/Output (Benchmark IMB-IO)

Lo script IMB-IO.slurm esegue un test di performance di accesso allo storage utilizzando le modalità S_Wirite_indv e S_Read_indv del benchmark IMB-IO. Eseguire il test di I/O in lettura e scrittura su HOME e SCRATCH (storage ad alte prestazioni).

Per testare l'area SCRATCH copiare IMB-IO e IMB-IO.slurm ed eseguire il job:

```
cp IMB-IO.slurm    $SCRATCH
cp IMB-IO        $SCRATCH
cd $SCRATCH
sbatch  IMB-IO.slurm
```

Input/output (disktest)

Questo e' un semplice benchmark dello storage che può essere usato in alterativa a IMB-IO o per fare un controllo incrociato dei risultati.
https://www.ibm.com/support/pages/how-use-unix-dd-command-monitor-writing-performance-disk-device
La script disktest.bash utilizza il comando dd per testare la velocità di lettura e scrittura su disco:
cd HPC2223/perf/disktest/
cp /hpc/group/T_2022_HPCPROGPAR/perf/disktest/* .
Eseguire lo script slurm, quindi compilare a mano il file disktest.csv e generare il plot.

Realizzare i seguenti plot (adattare gli script python in time-plot/):
1) Confronto della banda (Mbyes/s) tra SHM, OPA e IB al crescere della dimensione dei messaggi
2) Confronto della latenza (tempo in us) tra SHM, OPA e IB al crescere della dimensione dei messaggi da (1 a 100 byte).
3) Confronto della banda (Mbytes/s) in lettura e scrittura su disco al crescere della dimensione dei dati, mettendo a confronto HOME e SCRATH

Svolgimento:

- Entrare nel sito https://github.com/intel/mpi-benchmarks e scaricare l'archivio.
- Mettere l'archivio all'interno della cartella IBM con il drag and drop dal PC.
- Estrarre l'archivio con il comando "unzip <nomeFileZip>.zip".
- Entrare nella cartella estratta (con il comando cd <nomeCartella>) ed usare i seguenti comandi :
  - module load intel impi
  - make IMB-MPI1
  - make IMB-IO
- Noterete che nella cartella vengono generati nuovi file, copiateli nella cartella IBM : "cp IMB-IO <percorso cartella IBM>" e "cp IMB-MPI1 <percorso cartella IMB>"
- Tornare nella cartella IBM con il comando "cd .." e avviate tutti i batch "IMB-IO.slurm" con i seguenti comandi :
  - "sbatch IMB-IO.slurm"
  - "sbatch IMB-MPI1-n1.slurm"
  - "sbatch IMB-MPI1-n2.slurm"

Come noterete dalla consegna, ci serve il file .dat anche di IB, quindi dobbiamo farlo noi lo script slurm che genera il suo dat, in questo modo (io l'ho chiamato "IMB-MPI1-n3.slurm"), creare il file ed aggiungerci questo contenuto :

```
#!/bin/bash
#SBATCH --job-name=IMB-MPI1-ib     #
#SBATCH --output=%x.o%j  # Nome del file per lo strandard output
##SBATCH --error=%x.e%j  # Se non specificato stderr è rediretto su stdout
#SBATCH --partition=cpu   # Nome della partizione
#SBATCH --qos=cpu   # Nome della partizione
#SBATCH --nodes=2                # numero di nodi richiesti
##SBATCH --constraint=omnipath
#SBATCH --constraint=infiniband
#SBATCH --ntasks-per-node=1        # numero di cpu per nodo
##SBATCH --mem=2G                # massima memoria utilizzata
#SBATCH --time=0-00:10:00          # massimo tempo di calcolo

echo  "#SLURM_JOB_NODELIST     : $SLURM_JOB_NODELIST"
module load intel impi
CMD="mpirun    IMB-MPI1 pingpong"
echo "# $CMD"
eval $CMD   >  IMB-MPI1.ib.dat
```

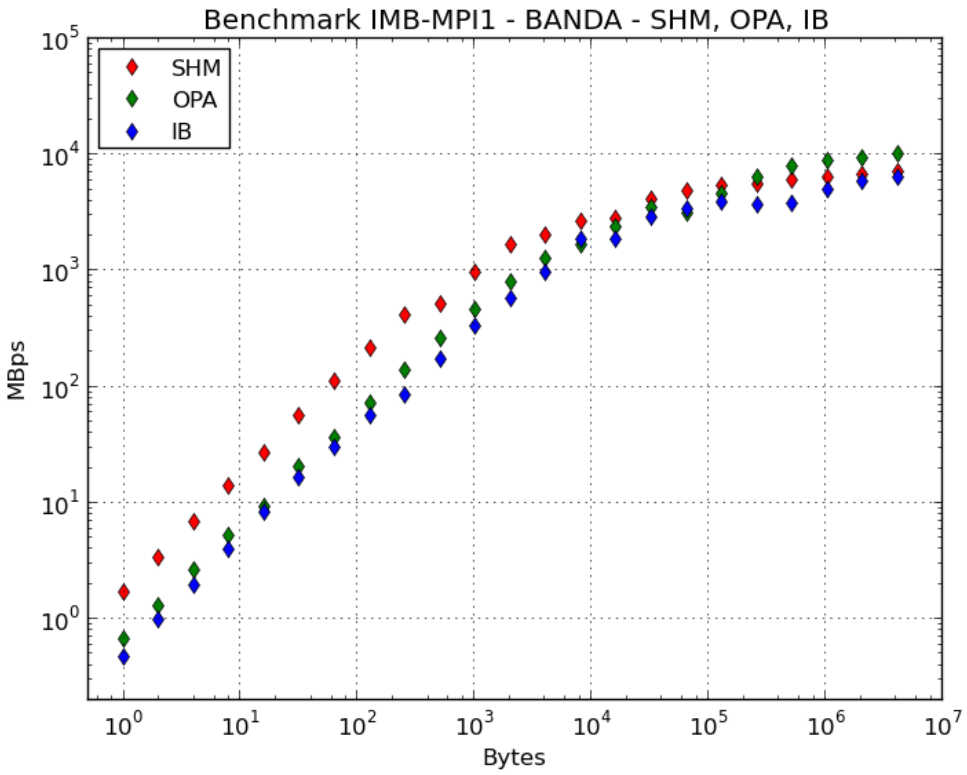
- Ora eseguire il nuovo script : "sbatch IMB-MPI1-n3.slurm"
- Dobbiamo anche fare la parte sull'area SCRATCH :
  - cp IMB-IO.slurm $SCRATCH
  - cp IMB-IO $SCRATCH
  - cd $SCRATCH
  - sbatch IMB-IO.slurm
- Tornare nella cartella IMB "cd /hpc/home/<nome.cognome>/HPC2223/perf/IMB/"

Ora che abbiamo generato tutti i file, bisogna fare i grafici di confronto con python, come abbiamo fatto nel precedente esercizio di laboratorio. Però non dobbiamo dimenticare di creare la cartella "scratch" per il terzo grafico. Usiamo i seguenti comandi dalla cartella IMB :

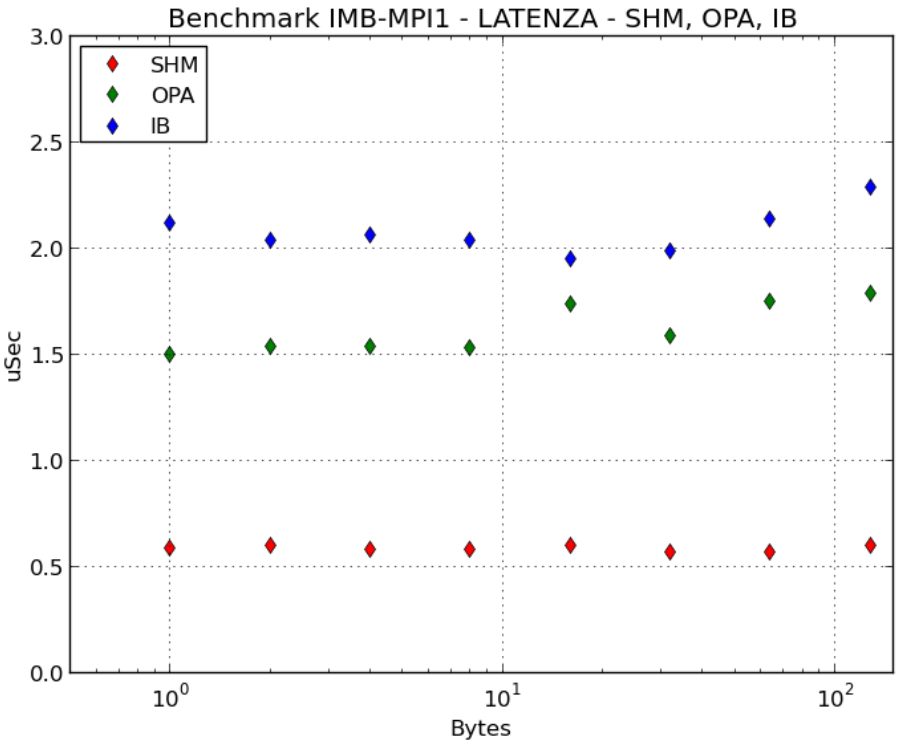
- mkdir scratch
- cd $SCRATCH (la cartella dello scratch)
- cp IMB-IO-S_Read-indv.dat /hpc/home/<nome.cognome>/HPC2223/perf/IMB/scratch/
- cp IMB-IO-S_Write-indv.dat /hpc/home/<nome.cognome>//HPC2223/perf/IMB/scratch/
- Tornare nella cartella IMB
- Avviare il file python, per generare i grafici bisogna usare i seguenti array che prendono i dati dai file di estensione .dat generati dagli script SLURM (prendere spunto dai grafici dell'esercizio del laboratorio precedent per costruirli) :
  - opa_data = pd.read_csv("IMB-MPI1.opa.dat",comment="#", sep='\s+', names=["bytes", "repetitions","time","MBps"])
  - shm_data = pd.read_csv("IMB-MPI1.shm.dat",comment="#", sep='\s+', names=["bytes", "repetitions","time","MBps"])
  - ib_data = pd.read_csv("IMB-MPI1.ib.dat",comment="#", sep='\s+', names=["bytes", "repetitions","time","MBps"])
  - home_io_read_data = pd.read_csv("IMB-IO-S_Read-indv.dat", comment="#", sep='\s+', names=["bytes", "repetitions","time","MBps"])
  - home_io_write_data = pd.read_csv("IMB-IO-S_Write-indv.dat", comment="#", sep='\s+', names=["bytes", "repetitions","time","MBps"])
  - scratch_io_read_data = pd.read_csv("scratch/IMB-IO-S_Read-indv.dat", comment="#", sep='\s+', names=["bytes", "repetitions","time","MBps"]
  - scratch_io_write_data = pd.read_csv("scratch/IMB-IO-S_Write-indv.dat", comment="#", sep='\s+', names=["bytes", "repetitions","time","MBps"])

Ecco qui i grafici generati :

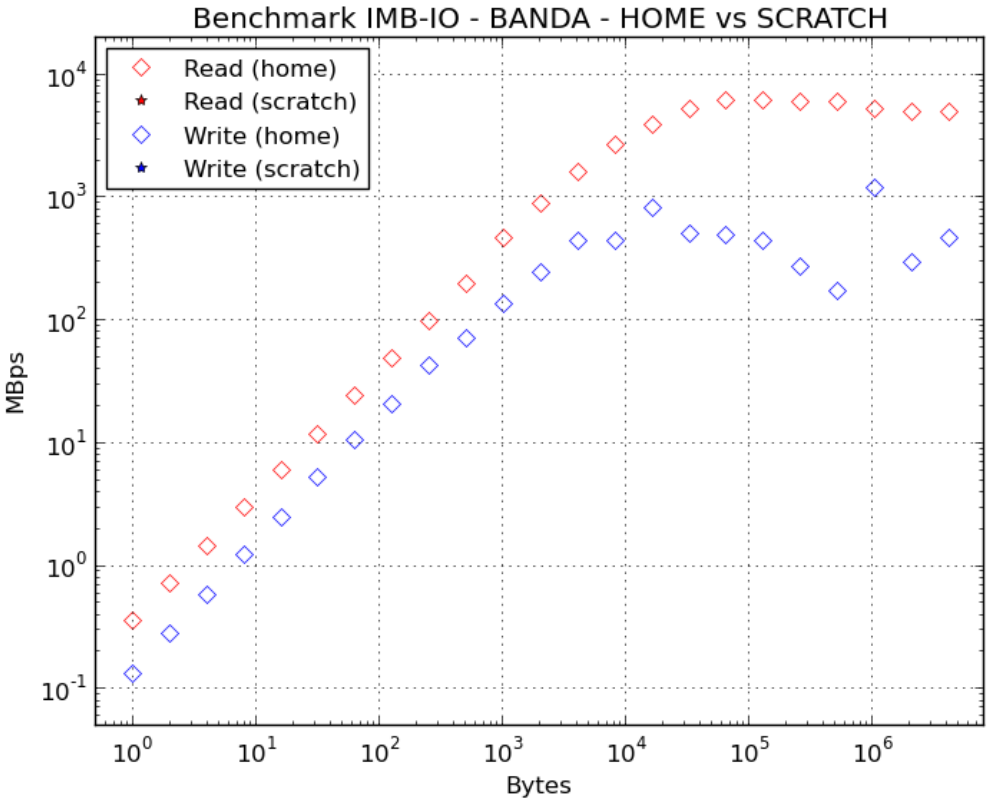
- Confronto della banda (Mbit/s) tra SHM, OPA e IB al crescere della dimensione dei messaggi :



- Confronto della latenza (tempo in us) tra SHM, OPA e IB al crescere della dimensione dei messaggi da (1 a 100 byte) :



- Confronto della banda (Mbytes/s) in lettura e scrittura su disco al crescere della dimensione dei dati, mettendo a confronto HOME e SCRATH :



Infine, fare l'ultimo grafico di confronto per il disktest tra home e scratch :

- Andare nella cartella disktest -> "cd ../disktest"
- Avviare lo script SLURM -> "sbatch disktest.slurm"
- Aspettare che il JOB termini e verificare i risultati nel file di output generato :

```
#SLURM_JOB_NODELIST: wn20
##################################
# HOME perf
Write to file:
2048+0 records in
2048+0 records out
2147483648 bytes (2.1 GB) copied, 3.47642 s, 618 MB/s

Read from file:
2048+0 records in
2048+0 records out
2147483648 bytes (2.1 GB) copied, 2.50543 s, 857 MB/s
##################################
# SCRATCH perf
Write to file:
2048+0 records in
2048+0 records out
2147483648 bytes (2.1 GB) copied, 1.90982 s, 1.1 GB/s

Read from file:
2048+0 records in
2048+0 records out
2147483648 bytes (2.1 GB) copied, 3.91692 s, 548 MB/s
##################################
```

- Aprire il file "disktest.csv" e inserire i risultati generati delle performance in lettura e scrittura per home e scratch (io ho voluto metterli in colonne separate perché mi trovavo meglio per lo script python successivo per generare i grafici) :

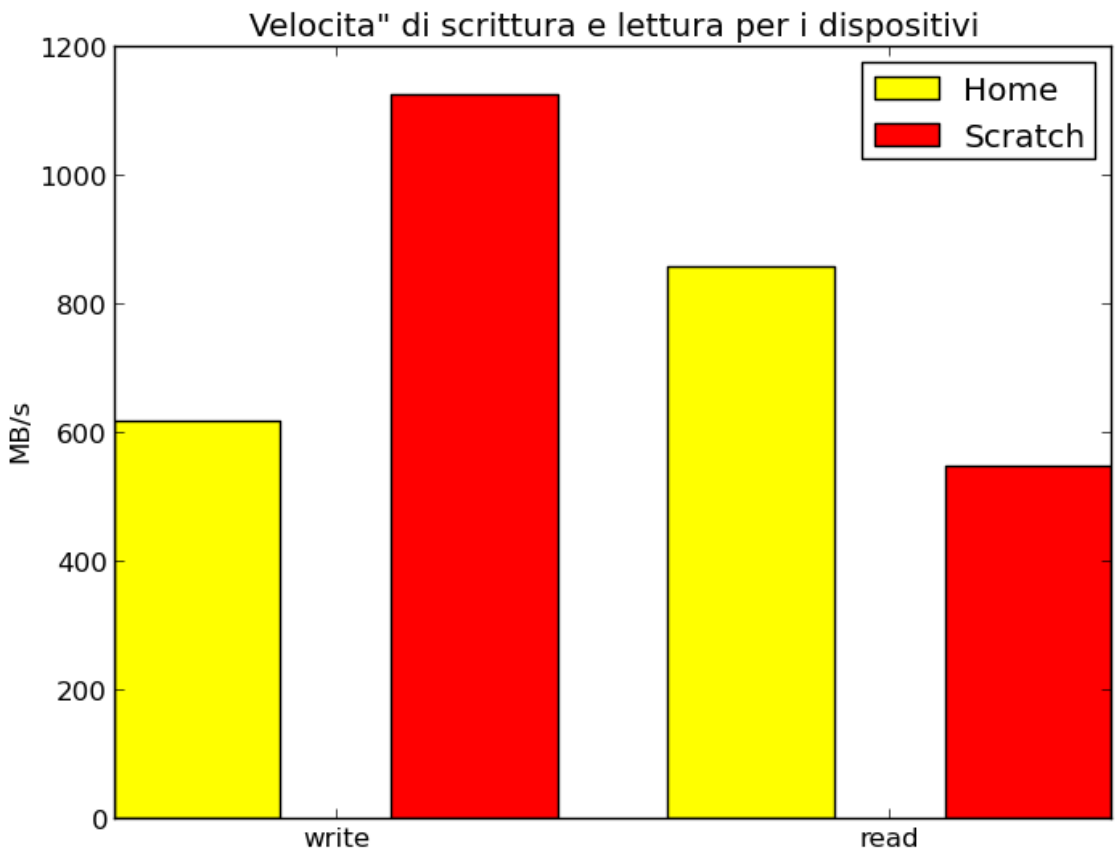
	A	B	C	D
1	device	write	read	
2	home	618	857	
3	scratch	1126	548	
4				
5				
6				

- Avviare lo script python per generare i grafici, io l'ho fatto così per adeguarlo alle modifiche che ho fatto al CSV (se non avete modificato il formato del CSV lasciando le righe in una colonna, non modificatelo) :

disktest_barplot.py

```python
import matplotlib.pyplot as plt
import pandas as pd

# Leggi i dati dal CSV utilizzando il punto e virgola come separatore
df = pd.read_csv('disktest.csv', sep=';')

# Seleziona i dati delle colonne 'write' e 'read' e li converte in un array
array_dati = df[['write', 'read']].values

# Dati dal CSV
data = {
    'operations': ['write', 'read'],
    'home': array_dati[0],
    'scratch': array_dati[1]
}

# Creazione di un DataFrame da dati
df = pd.DataFrame(data)

# Creazione del grafico a barre
fig, ax = plt.subplots()
bar_width = 0.3
space_between_bars = 0.2  # spazio tra le barre di un gruppo
bar_positions_write = range(len(df['operations']))
bar_positions_read = [pos + bar_width + space_between_bars for pos in bar_positions_write]

# Barre 'write' in giallo e 'read' in rosso
ax.bar(bar_positions_write, df['home'], width=bar_width, color='yellow', label='Home')
ax.bar(bar_positions_read, df['scratch'], width=bar_width, color='red', label='Scratch')

# Etichette e legenda
ax.set_xticks([pos + bar_width + space_between_bars/2 for pos in bar_positions_write])
ax.set_xticklabels(df['operations'])
ax.set_ylabel('MB/s')
ax.set_title('Velocita" di scrittura e lettura per i dispositivi')
ax.legend()

# Salvataggio del grafico
plt.savefig('disktest_barplot.png')
```

- Generare il grafico con il comando “python disktest_barplot.py”
- Verificare il grafico generato nel file “disktest_barplot.png” :



Da questo grafico si nota che home ha maggiori performance di lettura, mentre scratch ha maggiori performance di scrittura.

- Copiare sulla macchina linux tutto ciò che abbiamo fatto : “rsync -av --chmod=D755,F644 ~/HPC2223/ ${USER}@studenti.unipr.it@didattica-linux.unipr.it:html/HPC2223/”

Laboratorio: Progettazione di programmi paralleli

Consegna :

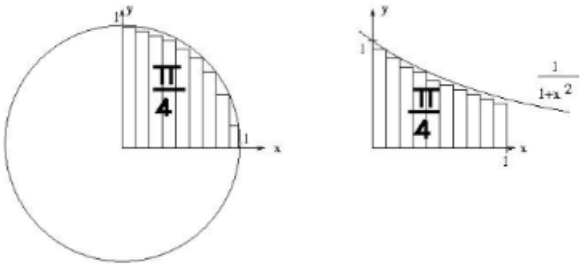
I seguenti programmi seriali verranno analizzati per valutare le possibili performance nella parallelizzazione a memoria condivisa e distribuita, tenendo conto della legge di Amdahl e degli overhead.

I file si trovano sul cluster in /hpc/group/T_2022_HPCPROGPAR/serial/ e si possono copiare con rsync

rsync -av /hpc/group/T_2022_HPCPROGPAR/serial/ ~/HPC2223/serial/

Esercizio CPI :

Il programma calcola pigreco utilizzando due diversi metodi di integrazione numerica.
Per il calcolo dell'integrale viene utilizzato il metodo dei rettangoli nell'intervallo tra 0 e 1.



Compilare ed eseguire il programma cpi.c localmente:
gcc cpi.c -o cpi -lm -O2

Nota: -O2 è il livello di ottimizzazione consigliato. Vedi: https://www.rapidtables.com/code/linux/gcc/gcc-o.html#optimization
./cpi -h

Considerazioni sulla parallelizzazione di cpi.c

Parallelizzazione:

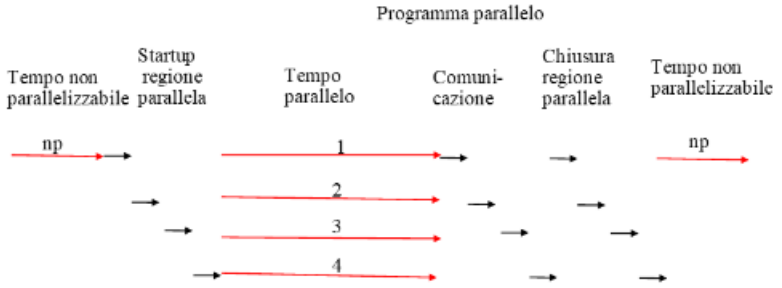
Un **ciclo for** ha il compito di calcolare tutte le aree degli intervalli tra 0 e 1. Le iterazioni sono indipendenti tra loro e possono essere distribuite tra i diversi thread o processi ed eseguite contemporaneamente.

La decomposizione può avvenire :
- a **grana grossa** : dividiamo gli intervalli in P parti, dove P è il numero di processori
- a **grana fine**: dividiamo gli intervalli in blocchi (chunks) che vengono distribuiti alle unità di processamento a rotazione o su richiesta quando hanno completato il chunk precedente.

Legge di Amdahl. La parte non parallelizzabile è trascurabile: gestione delle opzioni prima del ciclo e stampa dei risultati alla fine.

Comunicazione (se usiamo memoria distribuita con MPI). Ogni unità di processamento accumula nella variabile locale sum i contributi del cicli for che ha gestito. Al termine delle iterazioni tutti inviano la propria somma parziale (un double) al processo principale che somma tutti i contributi (operazione di **riduzione**). Vengono inviati p-1 messaggi di 6 byte (un double) una sola volta, al termine. Overhead trascurabile.

Startup: prima di iniziare la regione parallela è necessario creare P processi/thread e eliminarli al termine. Operazioni svolte una sola vota, quindi trascurabili.
Sincronismo: Prima di comunicare i risultati e terminare i processi/thread occorre attivare una barriera (implicita o esplicita), attendendo che tutti abbiamo completato il loro lavoro. Se usiamo decomposizione a grana file non abbiamo tempi di Idle. A gramma grossa occorre fare attenzione che i sottodomini siano di pari dimensione o non ci siano unità di processamento con diverse velocità.



Svolgimento:

Considerazioni sulla parallelizzazione di cpi.c :
Un ciclo for ha il compito di calcolare tutte le aree degli intervalli tra 0 e 1.
Le iterazioni sono indipendenti tra loro e possono essere distribuite tra i diversi thread o processi ed eseguite contemporaneamente. La decomposizione può avvenire a grana grossa (dividiamo gli intervalli in P parti, dove P è il numero di processori) oppure a grana fine se lavoriamo a memoria condivisa con thread. Tutto il resto del programma non è parallelizzabile.

- Eseguire questo comando per copiare la cartella “serial” : “rsync -av /hpc/group/T_2022_HPCPROGPAR/serial/ ~/HPC2223/serial/”
- Entrare nella cartella serial e poi nella cartella cpi “cd serial/cpi”
- Compilare il programma in c e creare un comando eseguibile di nome cpi “gcc cpi.c -o cpi -lm -O2”
- Dal momento che dobbiamo fare i confronti tra le funzioni f1() e f2(),modifichiamo il file “cpi_scaling.slurm” nel seguente modo, dal momento che esegue solo f1 e dobbiamo far eseguire anche f2 :

```
#!/bin/bash

#SBATCH --output=%x.o%j  # Nome del file per lo strandard output
##SBATCH --error=%x.e%j  # Se non specificato stderr è rediretto su stdout
#SBATCH --partition=cpu_guest          # Nome della partizione, cpu per job paralleli
#SBATCH --qos=cpu_guest          #
#SBATCH --nodes=1               # numero di nodi richiesti
#SBATCH --cpus-per-task=1        # numero di cpu
##SBATCH --mem=2G               # massima memoria utilizzata
#SBATCH --time=0-00:10:00        # massimo tempo di calcolo

gcc cpi.c -o cpi -lm -O2
echo "function,iter,pi,error,t_np,t_p,hostname" > cpi_scaling_f1.csv
echo "function,iter,pi,error,t_np,t_p,hostname" > cpi_scaling_f2.csv
```

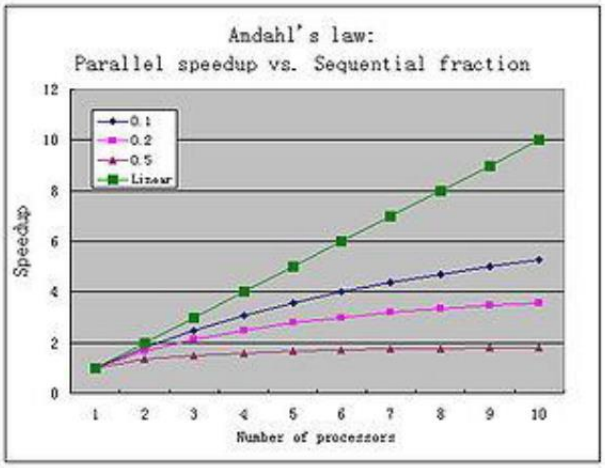
```
for N in $(seq 100000 10000 900000)
#for N  in  1000 2000 3000
do
./cpi -n $N -f 1 >> cpi_scaling_f1.csv
./cpi -n $N -f 2 >> cpi_scaling_f2.csv
done
```

N.B. Il programma in C è diverso dall'omonimo programma della prima esercitazione, dal momento che in questo programma viene restituito sia il tempo seriale, che il tempo in parallelo.
Inoltre, potete generare i file CSV anche con bash, è a vostra discrezione.

- Generare le immagini di confronto tramite lo script python dai due CSV generati. Cioè tra gli attributi dei tempi paralleli e non paralleli delle funzioni e se volete il **Qp** (che è la quota percentuale, Qp=Tp/(Tnp+Tp)), anche di **Qnp** (che è la quota non percentuale, Qnp=Tnp/(Tnp+Tp)).

Samdahl = Tseriale/Tparallelo = (Tnp+Tp) / (Tnp + Tp/P) = 1/(Qnp+Qp/P)

Tp è il tempo parallelizzabile
Tnp è il tempo non parallelizzabile
In termini di quote percentuali:
Qnp= Tnp/(Tnp+Tp) Qp=Tp/(Tnp+Tp)
dove Qnp+Qp =1

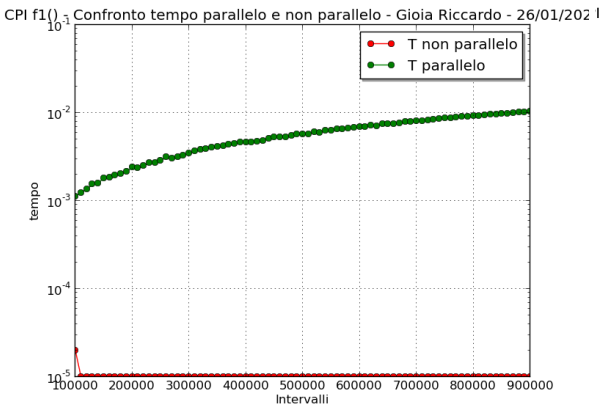


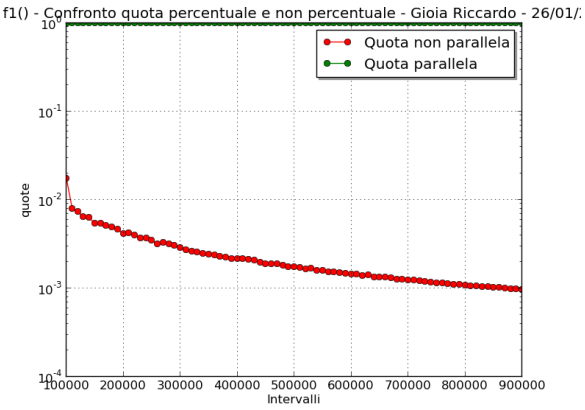
Ecco lo script python per generare tutti i grafici, (modificare il file “**cpi_scaling_plot.py**”) :

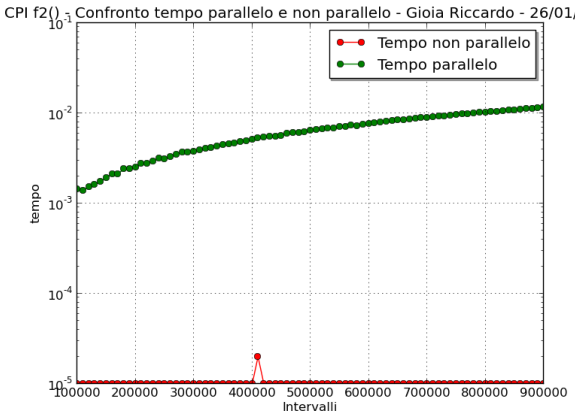
```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import pandas as pd

cpi_1 = pd.read_csv("cpi_scaling_f1.csv")
cpi_2 = pd.read_csv("cpi_scaling_f2.csv")

# Plot con la funzione f1
plt.title('CPI f1() - Confronto tempo parallelo e non parallelo – Nome e cognome - data‘)
plt.grid()
plt.xlabel('Intervalli')
plt.yscale('log')
plt.ylabel('tempo')
plt.plot(cpi_1.iter,cpi_1.t_np,'r-o',label='T non parallelo')
plt.plot(cpi_1.iter,cpi_1.t_p,'g-o',label='T parallelo')
plt.legend(shadow=True)
plt.savefig('cpi_scaling_tempo_f1.png')

plt.clf()

plt.title('CPI f1() - Confronto quota percentuale e non percentuale - Nome e cognome - data ')
plt.grid()
plt.xlabel('Intervalli')
plt.yscale('log')
plt.ylabel('quote')
plt.plot(cpi_1.iter,cpi_1.t_np/(cpi_1.t_p+cpi_1.t_np),'r-o',label='Quota non parallela')
plt.plot(cpi_1.iter,cpi_1.t_p/(cpi_1.t_p+cpi_1.t_np),'g-o',label='Quota parallela')
plt.legend(shadow=True)
plt.savefig('cpi_scaling_quote_f1.png')

# Plot con la funzione f2

plt.clf()

plt.title('CPI f2() - Confronto tempo parallelo e non parallelo - Nome e cognome - data ')
plt.grid()
plt.xlabel('Intervalli')
plt.yscale('log')
plt.ylabel('tempo')
plt.plot(cpi_2.iter,cpi_2.t_np,'r-o',label='Tempo non parallelo')
plt.plot(cpi_2.iter,cpi_2.t_p,'g-o',label='Tempo parallelo')
plt.legend(shadow=True)
plt.savefig('cpi_scaling_tempo_f2.png')

plt.clf()

plt.title('CPI f2() - Confronto quota percentuale e non percentuale - Nome e cognome - data ')
plt.grid()
plt.xlabel('Intervalli')
plt.yscale('log')
plt.ylabel('quote')
plt.plot(cpi_2.iter,cpi_2.t_np/(cpi_2.t_p+cpi_2.t_np),'r-o',label='Quota non parallela')
plt.plot(cpi_2.iter,cpi_2.t_p/(cpi_2.t_p+cpi_2.t_np),'g-o',label='Quota parallela')
plt.legend(shadow=True)
plt.savefig('cpi_scaling_quote_f2.png')

plt.clf()
# Plot confronfo tra f1() e f2()
plt.title('Confronto funzioni f1() e f2() - CPI Seriale - Nome e cognome – data’)
plt.grid()
plt.xscale('log')
plt.xlabel('Intervalli')
plt.ylabel('tempo')
plt.plot(cpi_1.iter,cpi_1.t_np,'r-o',label='f1() - T non parallelo')
plt.plot(cpi_2.iter,cpi_2.t_np,'b-o',label='f2() - T non parallelo')
```

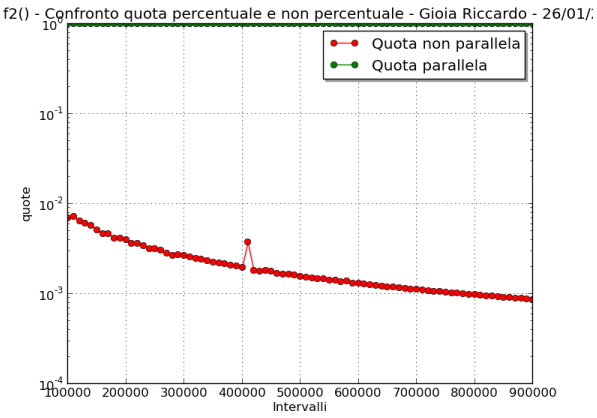
```
plt.legend(shadow=True,loc='upper left')
plt.savefig('confronto_Tseriale.png')

plt.clf()

plt.title('Confronto funzioni f1() e f2() - CPI Parallelo - Nome e cognome - data ')
plt.grid()
plt.xscale('log')
plt.xlabel('Intervalli')
plt.ylabel('tempo')
plt.plot(cpi_1.iter,cpi_1.t_p,'g-^',label='f1() - Tempo parallelo')
plt.plot(cpi_2.iter,cpi_2.t_p,'y-^',label='f2() - Tempo parallelo')
plt.legend(shadow=True,loc='upper left')
plt.savefig('confronto_Tparallelo.png')

plt.clf()

plt.title('Confronto funzioni f1() e f2() - CPI Quota percentuale - Nome e cognome - data ')
plt.grid()
plt.xscale('log')
plt.xlabel('Intervalli')
plt.ylabel('quota')
plt.plot(cpi_1.iter,cpi_1.t_p/(cpi_1.t_p+cpi_1.t_np),'g-^',label='f1() - Q parallela')
plt.plot(cpi_2.iter,cpi_2.t_p/(cpi_2.t_p+cpi_2.t_np),'y-^',label='f2() - Q parallela')
plt.legend(shadow=True,loc='upper left')
plt.savefig('confronto_Qp.png')

plt.clf()

plt.title('Confronto funzioni f1() e f2() - CPI Quota percentuale - Nome e cognome – data')
plt.grid()
plt.xscale('log')
plt.xlabel('Intervalli')
plt.ylabel('quota')
plt.plot(cpi_1.iter,cpi_1.t_np/(cpi_1.t_p+cpi_1.t_np),'r-o',label='f1() - Quota non parallela')
plt.plot(cpi_2.iter,cpi_2.t_np/(cpi_2.t_p+cpi_2.t_np),'b-o',label='f2() - Quota non parallela')
plt.legend(shadow=True,loc='upper left')
plt.savefig('confronto_Qnp.png')
```

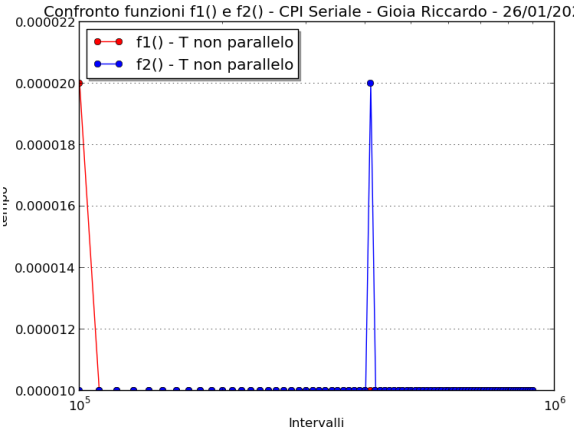
Ecco i grafici generati :

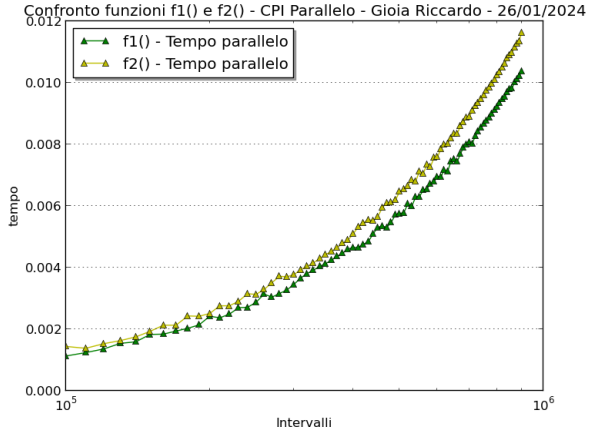


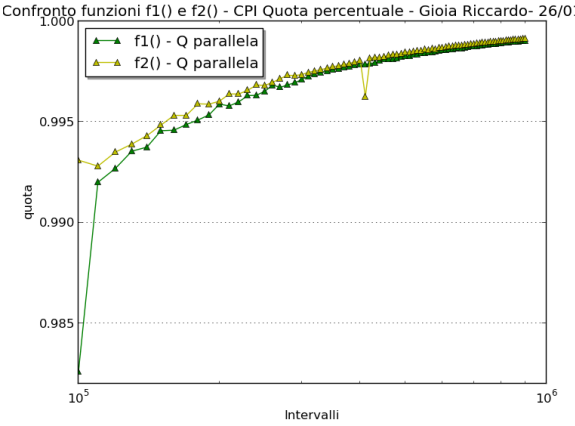


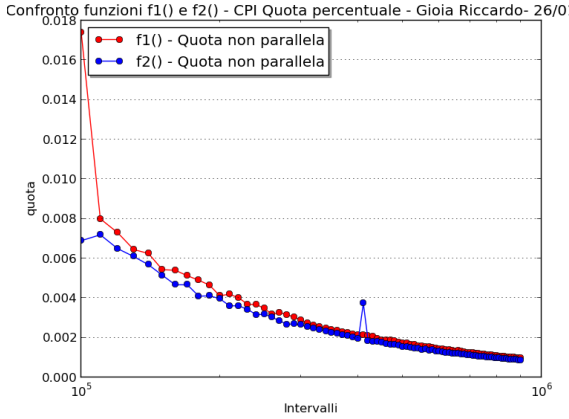












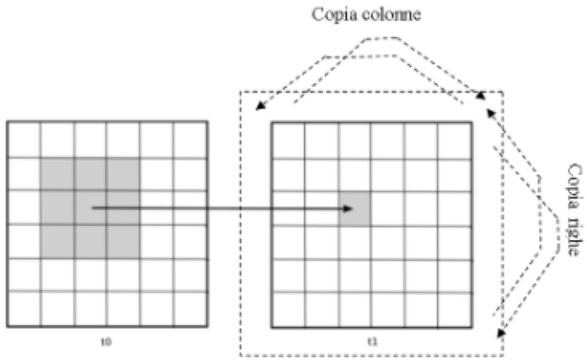
Da questi grafici, concludo che questo programma è più efficiente quando è parallelizzato e che è la funzione f1() è più efficiente di f2().

Esercizio heat:

Consegna :

- Il programma heat.c simula la propagazione del calore su una superficie rettangolare con alcune parti mantenute a temperatura costante.

Individuate nel programma:
- le funzioni Init_center() Init_left() e Init_top() che mantengono la temperatura nelle corrispondenti parti del rettangolo (parte centrale, bordo in alto e bordo a sinistra).
- le funzioni copy_rows() e copy_cols() realizzano le condizioni periodiche al contorno nelle due direzioni, ovvero mettono a contatto termico i bordi alto/basso e destra/sinistra.
- la funzione jacobi() determina la nuova temperatura allo step t1 come media della temperatura nei punti vicini allo step t0 (in questo caso solo i 4 vicini, nord, sud, est e ovest)



Compilare ed eseguire il programma :
gcc -O2 heat.c -o heat

Il programma scrive su stdout i valori della temperatura finale nei punti della superficie e su stderr i parametri e tempi di esecuzione.
Possiamo salvare la mappa del calore in un file heatmap.out:
./heat > heatmap.out

Utilizzando la funzione imsave di python possiamo generare una color map in formato png partendo dall'output del programma.

Il programma heatmap_plot.py esegue il programma e determina heatmap.png.

Modificare il programma in modo da ottenere qualche immagine della propagazione facendo variare le zone a temperatura costante con e senza condizioni periodiche (commentare/decommentare il codice e ricompilare), Verificare il tempo di calcolo della simulazione al crescere della dimensione della superficie.

Il programma heatmap_animation.py determina diverse heatmap al crescere del numero di iterazioni che poi utilizza per generare una animazione in formato gif. Le singole immagini png vengono incluse nel file gif in ordine alfabetico.

Provare a generare una animazione significativa.

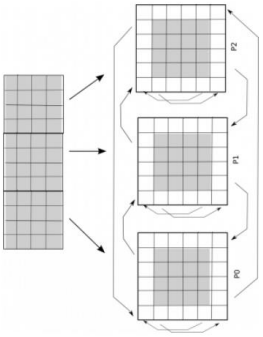
Considerazioni sulla parallelizzazione di heat.c :
Parallelizzazione: .decomposizione dominio: la funzione Jacoby, che ad ogni iterazione aggiorna la temperatura di tutti i punti della superficie, può essere parallelizzata partizionando la superficie tra i processi/thread. Anche in questo caso possiamo lavorare a grana grossa, ad esempio suddividendo la superficie in P strisce, oppure a grana fine se lavoriamo con i thread.
Ovviamente il ciclo for esterno for(iter=0; iter<MAX_ITER; iter=iter+1) non può essere parallelizzato poiché rappresentata la progressione temporale.
In prima approssimazione consideriamo tutto il resto non parallelizzabile (principalmente l'aggiornamento della temperatura nei punti caldi).
Occorre analizzare i tempi Tp e Tnp per la legge di Amdahl.

Overhead:
- **Comunicazione**.

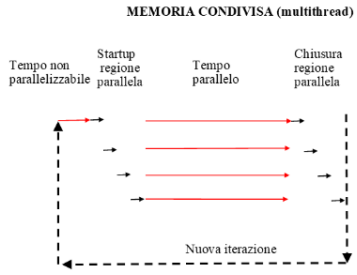
Se la memoria è distribuita partizioniamo la superficie, ad esempio, in P righe (1D) e distribuiamo un sottodominio ad ogni processo. Ad ogni iterazione ogni processo aggiorna il proprio sottodominio. Al termine delle iterazioni i sottodomini aggiornati devono essere ricomposti inviandoli al processo principale.

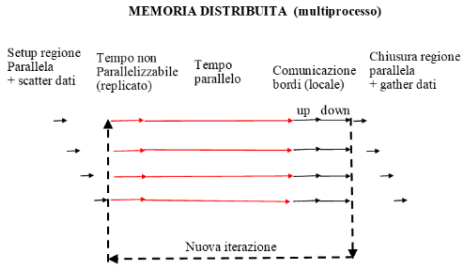
I sottodomini devono essere inizialmente distribuiti dal processo principale verso i P-1 altri processi (scatter), O(P). Al termine occorre fare l'operazione opposta (gather), O(P).
Inoltre, ad ogni iterazione le righe di bordo devono essere scambiate tra processi adiacenti.

Se abbiamo I iterazioni e ogni riga contiene N punti ogni processi scambia 4 messaggi di N float ad ogni iterazione. Tcomm= 4 * I * Tlat * (N/Band)



Se utilizziamo il modello a memoria condivisa i thread aggiornano parti diverse della stessa superficie. In questo caso possiamo operare "fine grain" ovvero distribuiamo i cicli for della funzione Jacobi tra i thread disponibili.
- **Startup**: Nel modello a memoria distribuita vengono creati P processi indipendenti all'inizio del programma e cancellati al termine O(P).
Nel modello a memoria condivisa i thread vengono creati ed eliminati ad ogni iterazione O(P*I)
- **Sincronismo**: barriera (implicita o esplicita) al termine di ogni iterazione prima di inviare i messaggi.
Se il dominio è decomposto in sottodomini della stessa dimensione non abbiamo tempi di Idle.
Il seguente diagramma riporta la profilazione temporale nell'esecuzione parallela. Per semplicità è riportato lo scambio di un solo messaggio.





Svolgimento:

- Andare nella cartella di heat : “cd ../heat”.
- Utilizzare questo comando : “gcc -O2 heat.c -o heat”.
- Eseguire il programma con il nuovo comando heat generato “heat” per verificarne il funzionamento.
- Modificare il file “heat_scaling_plot.py” mettendo nei titoli dei grafici i propri dati e la data di oggi.
- Eseguire lo script SLURM “sbatch heat_scaling.slurm” per aggiornare il file “heat_scaling.csv”.
- Ora che avete aggiornato il CSV, generate il grafico con il comando : “python heat_scaling_plot.py”.
- Generare anche la gif con il comando “python heatmap_animation.py”.

Grafico confronto tra tempo percentuale e tempo non percentuale :

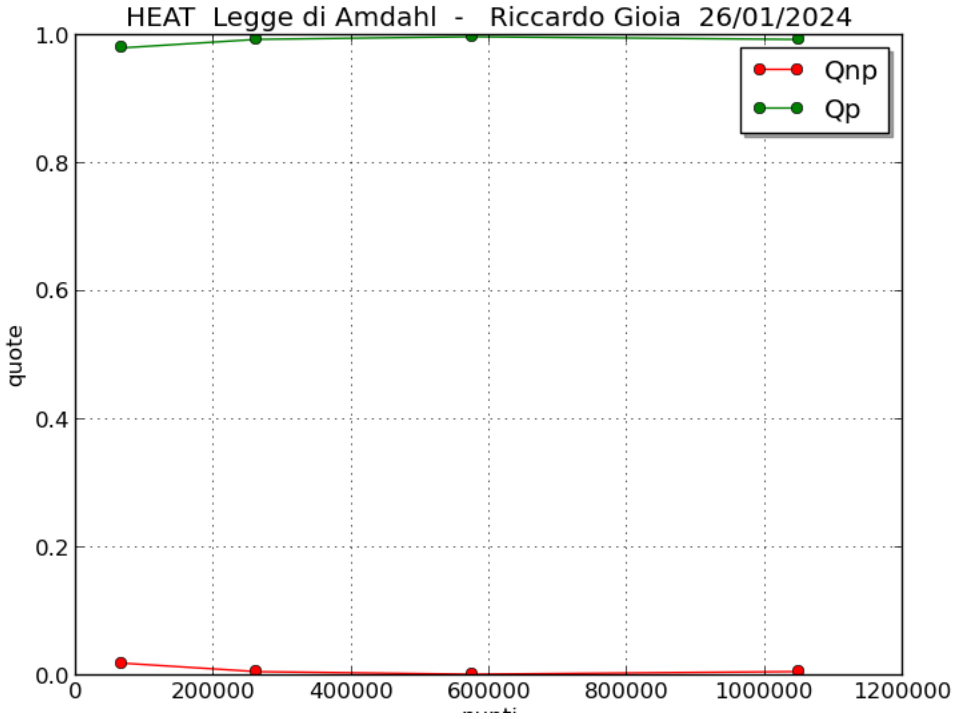
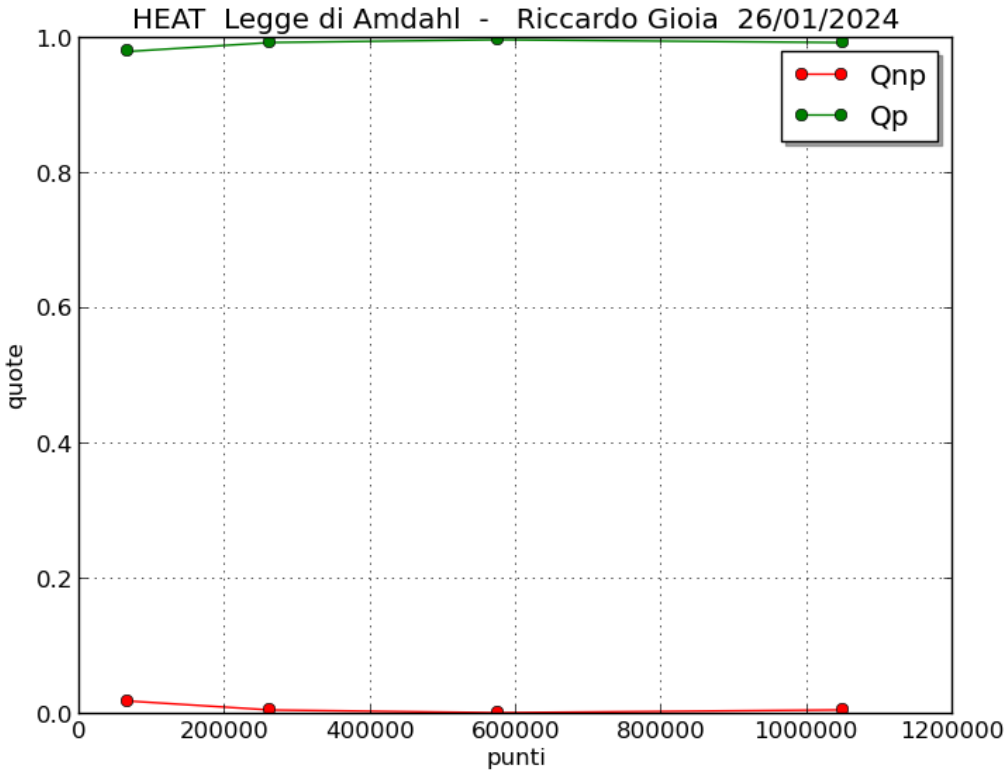


Grafico confronto tra quota percentuale e quota non percentuale :



Esercizio Fattorizzazione:

Consegna :

Il programma factorize.c utilizza la libreria BN (Big Numbers) di openSSL per determinare due numeri primi p e q, calcolarne il prodotto (modulo). Il modulo M fa parte delle chiavi pubblica e privata dell'algoritmo crittografico RSA, che può essere violato se si riesce a scomporlo nei fattori primi p e q.

Il programma, dopo aver calcolato il Modulo prova a fattorizzarlo a forza bruta, ovvero calcola progressivamente i numeri primi x e divide Modulo/x .

- Analizzare ed eventualmente realizzare miglioramenti dell'algoritmo
- Definire la tecnica ottimale di scomposizione del dominio e il relativo impatto della legge di Amdahl, e degli overhead introdotti dalla parallelizzazione (comunicazione, bilanciamento del carico, ecc…) a memoria condivisa e distribuita.
- Scrivere nel documento fattorizzazione.txt una relazione riguardo i punti indicati.

Svolgimento :

- Andare nella cartella di factorize : “cd ../ factorize”.
- Copiare il contenuto dalla cartella old alla cartella attuale con questo comando ‘cp -r "/hpc/home/${USER}/HPC2223/serial/factorize/old/"/* .’
- Compilare i tre programmi C con i seguenti comandi e poi avviarli e testarli :
  - “gcc factorize0.c -o factorize0 -lcrypto”
  - “gcc factorize1.c -o factorize1 -lcrypto -lm”
  - “gcc factorize2.c -o factorize2 -lcrypto -lm”

- Ora dobbiamo calcolare il tempo dei tre programmi tramite lo script SLURM, io ho preferito crearne uno nuovo per ciascun programma con i seguenti comandi copiandolo dal file factorize.slurm :
  - “cp -r factorize.slurm factorize0.slurm”
  - “cp -r factorize.slurm factorize1.slurm”
  - “cp -r factorize.slurm factorize2.slurm”
- Modificare i tre nuovi file e adeguarli a ciascuno dei tre programmi in C e aggiungere la funzione tipe per calcolare il tempo per vedere quale e il più efficiente :
  echo "#SLURM_JOB_NODELIST: $SLURM_JOB_NODELIST"
  gcc factorize0.c -o factorize0 -O2 -lcrypto -lm
  time ./factorize0 24
- Avviare i tre script slurm e visualizzare i risultati una volta che tutti finiscono
- Dai risultati, il programma più efficiente è factorize1.c

Scrivere infine il file di testo factorize.txt in cui scrivete tutti i punti richiesti alla fine della consegna.
Queste sono le cose che ho aggiunto io :
Come migliorare l'algoritmo : Un modo per migliorare l'algoritmo è quello di parallelizzare il programma task differenti. Cioè si potrebbe dividere il dominio del problema in più sottodomini e creare un software parallelo che possa eseguire la fattorizzazione in modo più efficiente su più task differenti (OpenMP o MPI). Si potrebbe utilizzare con OpenMP o MPI anche un meccanismo master slave.
Tecnica ottimale di scomposizione del dominio : La tecnica ottimale sarebbe quella di dividere il dominio in tanti sottodomini quanti sono i processori quindi tramite una grana grossa che permetterebbe di diminuire il tempo sprecato con l'overhead e di aumentare l'efficienza del programma in base alla legge di Amdahl (Tp + Tnp)/(Tn/P + Tp + Toh).
Legge di Amdahl e Overhead : La parte di ricerca è parallelizzabile, mentre le parti non parallelizzabili no perché' semplici. Inoltre non richiede comunicazioni di alcun tipo dato che ogni blocco è indipendente dagli altri.

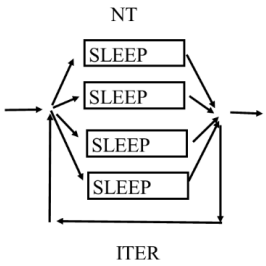
Inviare tutto alla propria macchina linux con questo comando : “rsync -av --chmod=D755,F644 ~/HPC2223/serial ${USER}@studenti.unipr.it@didattica-linux.unipr.it:html/HPC2223/”

Laboratorio: openMP base

omp_overhead.c :

Consegna :

Il programma omp_overhead valuta l'overhead dovuto all'avvio e chiusura di un pool di NT thread.
Il pool viene attivato con un ciclo for di NT iterazioni; ogni thread attende un tempo "SLEEP", quindi il pool si chiude e l'operazione viene ripetuta ITER volte. Il tempo di esecuzione T = SLEEP x ITER + Toverhead



Variare il numero di thread (NT) e di usleep (SLEEP) in modo da valutare il tempo di overhead al crescere del numero di thread.

Svolgimento :

- Avviare i seguenti comandi per creare la cartella e copiarne il contenuto ed entrarci:
  - mkdir ~/HPC2223/openMP
  - rsync -av /hpc/group/T_2022_HPCPROGPAR/openMP/base/ ~/HPC2223/openMP/base/
  - cd openMP/base
- Avviare il di questo programma con il seguente comando : “sbatch omp_overhead.slurm”

Dai risultati del JOB che avvia il programma più volte cambiando il numero di thread :

```
# NT=28,  ITER=1000, SLEEP=1000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  1.00        1.20        0.20        202.95
 # NT=8,  ITER=1000, SLEEP=1000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  1.00        1.12        0.12        118.64
 # NT=4,  ITER=1000, SLEEP=1000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  1.00        1.11        0.11        114.82
 # NT=2,  ITER=1000, SLEEP=1000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  1.00        1.11        0.11        108.96
 # NT=1,  ITER=1000, SLEEP=1000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  1.00        1.11        0.11        105.06
 # NT=28,  ITER=1000, SLEEP=2000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  2.00        2.14        0.14        144.06
 # NT=8,  ITER=1000, SLEEP=2000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  2.00        2.12        0.12        120.98
 # NT=4,  ITER=1000, SLEEP=2000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  2.00        2.11        0.11        113.44
 # NT=2,  ITER=1000, SLEEP=2000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  2.00        2.11        0.11        108.06
 # NT=1,  ITER=1000, SLEEP=2000
# Tsleep(s)    Treal(s)  Toverhead(s)   Toverheadperiter(micros)
  2.00        2.10        0.10        101.07
```

Possiamo concludere che all’aumentare del numero di thread, aumenta anche l’overhead.

omp_balancing.c :

Consegna :

Il programma simula un carico di lavoro parallelizzabile ma sbilanciato.

```
for(i=0; i<n; i++) {
   x = drand48();
   usleep(x*100000);
   sum += x;
}
```

ottimizzare il bilanciamento del carico e i tempi di calcolo di sum.

Svolgimento :

```
#pragma omp parallel for schedule (dynamic,1)
 for(i=0; i<n; i++) {
    rank= omp_get_thread_num();
    x =  rand();
    printf ("i:%d r:%d x:%.4f \n", i, rank, x);
    usleep(x*100000);
    #pragma critical
    sum += x;
 }
```

Andando ad aggiungere una direttiva **#pragma** in testa al for, come nell'esempio, succede che si hanno tanti thread che in parallelo iterano.

#pragma omp parallel for schedule(static,1)

Inserendo una direttiva come questa si ha che ogni thread esegue un chunk, sempre con lo stesso ordine, la rotazione è quindi statica

#pragma omp parallel for schedule(dynamic,1)

Al contrario, in questo caso la rotazione è dinamica.

In questo caso questa soluzione è migliore, non vincola ogni thread a dover eseguire un chunk determinato a priori, dato che, ad esempio, il primo thread che esegue potrebbe essere l'ultimo a terminare, andando quindi a bloccare gli altri.

Utilizzato uno scheduler dinamico si ottiene un bilanciamento del carico migliore.

Il termine "**chunk**" si riferisce alla porzione di lavoro suddivisa e assegnata a ciascun thread in un ambiente di programmazione parallela, come nel contesto di OpenMP.

Nella programmazione parallela, il chunk rappresenta il numero di iterazioni del ciclo che vengono assegnate a ciascun thread quando viene utilizzata una clausola di scheduling.

Con static viene assegnato ad ogni thread una porzione fissa di iterazioni, con dynamic variabile.

#pragma omp parallel for schedule(dynamic,1)

È meglio di

#pragma omp parallel for schedule(dynamic,100)

Perché I tempi sono variabili e ci ho aggiunto sulla variabile sum una **#pragma omp critical** per evitare una race condition.

omp_cpi.c :

Consegna :

rsync -av /hpc/group/T_2022_HPCPROGPAR/openMP/cpi/ ~/HPC2223/openMP/cpi/

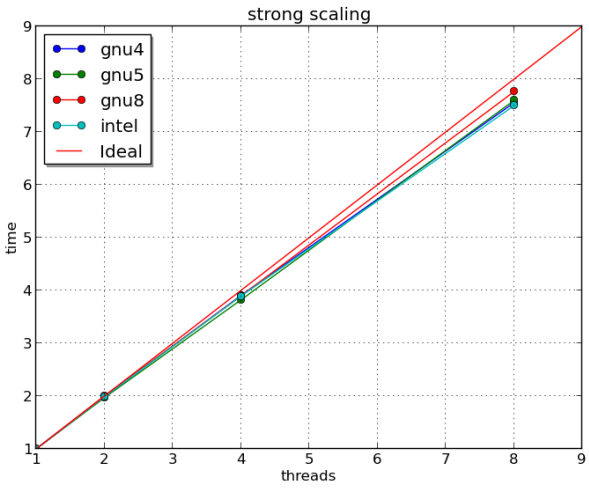
Eseguire lo Strong Scaling e generare il plot.

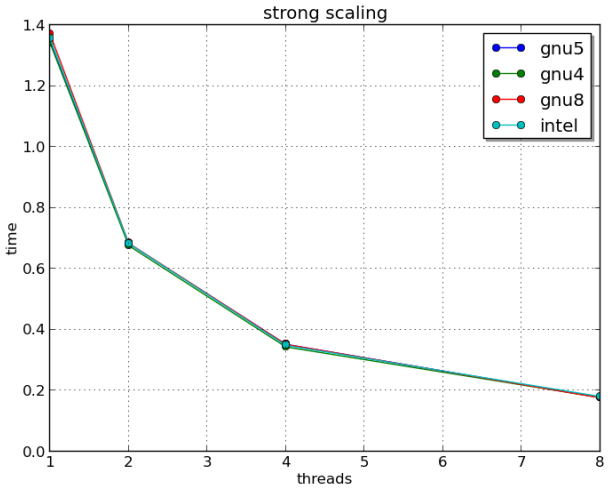
Svolgimento :

Qui ci ho aggiunto solo un #omp critical per evitare una race condition.

```
#pragma critical //Aggiunto per evitare race condition
#pragma critical
sum += f(x);
```

Dopo di che eseguire tutti script SLURM e gli script python per poter generare i grafici.





Laboratorio: applicazioni openMP

Consegna :

Parallelizzare con openMP le applicazioni heat e factorize.
Creare le directory HPC2223/openMP/heat e HPC2223/openMP/factorize copiare i sorgenti seriali, rinominarli con prefisso omp_ e aggiungere il supporto openMP.
Aggiungere l'intestazione #include <omp.h>
Individuare li cicli parallelizzabili, aggiungere la direttiva #pragma omp for aggiustando se necessario le clausole. Dove necessario determinare e stampare i tempi di calcolo. Esempio:
t1=omp_get_wtime();
…
t2=omp_get_wtime();
Stampare anche il rank dei thread. Esempio:
printf("# Thr:%d / %d ", omp_get_thread_num(), omp_get_num_threads());
Attenzione alla sovrapposizione delle stampe nelle regioni parallele. Può essere utile l'accesso esclusivo:

```
#pragma omp critical
{
printf (..);
}
```

Compilazione: aggiungere -fopenmp
gcc -O2 omp_heat.c -fopenmp -o omp_heat
Esempio di esecuzione:
OMP_NUM_THREADS=8 ./omp_heat > omp_heatmap.out

Heat: Analisi dello speedup ed eventuali ottimizzazioni.
Realizzare lo script slurm omp_heat_scaling.slurm e lo script python omp_heat_plot.py per il plot dello strong scaling per superfici 2048x2048.
Esempio di script slurm:

```
#!/bin/bash
#SBATCH --output=%x.o%j
##SBATCH --error=%x.e%j
#SBATCH --partition=cpu_guest
#SBATCH --qos=cpu_guest
#SBATCH --nodes=1
```

#SBATCH --cpus-per-task=8
#SBATCH --time=00-23:00:00
for T in 1 2 4 8
do
..
done
Valutare l'importanza del compilatore (esempio gnu4, gnu8, intel) e delle opzioni di ottimizzazione
https://www.rapidtables.com/code/linux/gcc/gcc-o.html#optimization
Scrivere nel file heat.txt una breve relazione sull'attività svolta

Factorize: analisi dello speedup e eventuali ottimizzazioni
Realizzare lo script slurm omp_factorize_scaling.slurm
Determinare il numero di core-hours necessario per fattorizzare un numero di 128 bit.
Analizzare ed eventualmente implementare una procedura di check pointing basata sul modello master-slave e implementato con la direttiva sections.
Scrivere nel file factorize.txt una breve relazione sull'attività svolta

Svolgimento :

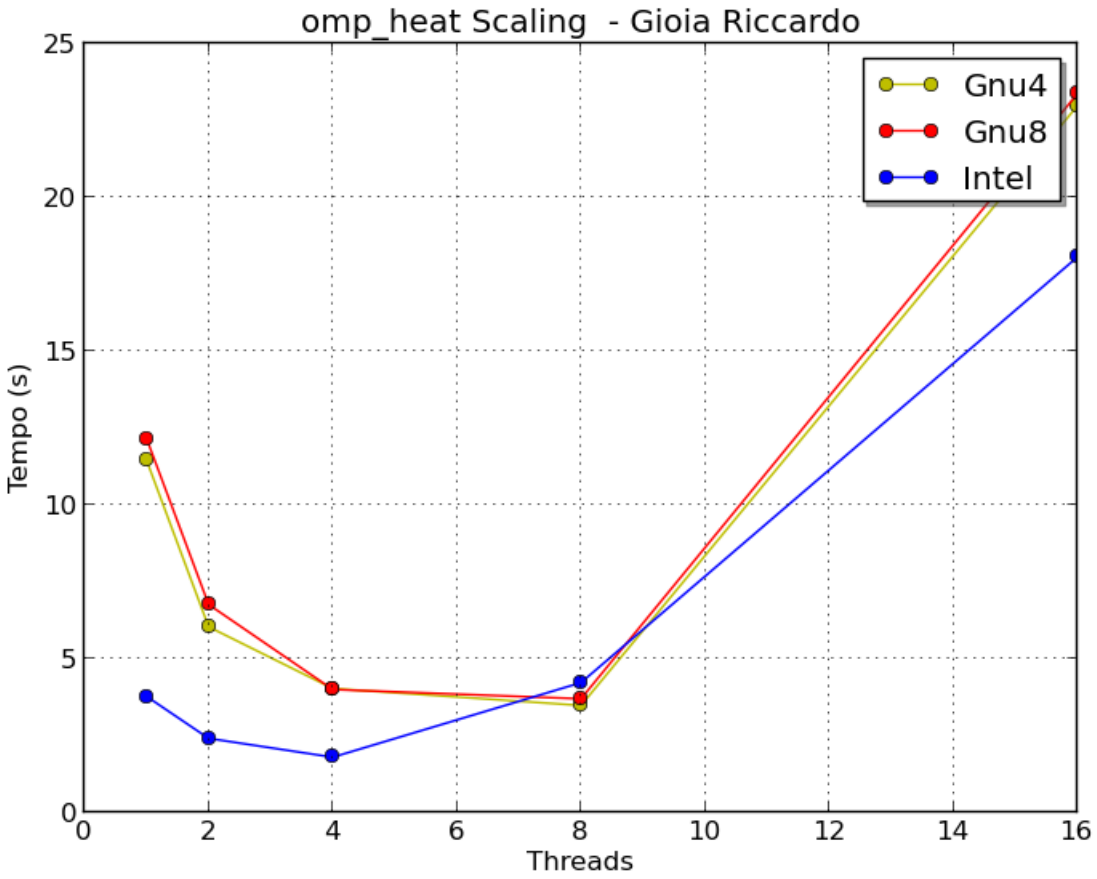
Heat :

Modificato solo sto metodo :

```
void Jacobi_Iterator_CPU(float * __restrict T, float * __restrict T_new, const int NX, const int NY) {
	int i,j;
	#pragma omp parallel for private (i) shared(num_threads) //righe j in parallelo
	for(j=1; j<NY-1; j++)
	{
		#pragma omp critical
		{
			printf("# Thr:%d / %d on row %d\n", omp_get_thread_num(), omp_get_num_threads(), j);
			num_threads = omp_get_num_threads();
		}
		for(i=1; i<NX-1; i++) {
			float T_E = T[(i+1) + NX*j];
			float T_W = T[(i-1) + NX*j];
			float T_N = T[i + NX*(j+1)];
			float T_S = T[i + NX*(j-1)];
			T_new[NX*j + i] = 0.25*(T_E + T_W + T_N + T_S);
		}
	}
}
```

La direttiva #pragma omp parallel for private(i) nel codice indica che il ciclo for successivo è parallelizzato, con la variabile i dichiarata come privata per ciascun thread parallelo. In altre parole, ogni thread parallelo avrà una propria copia locale della variabile i, evitando conflitti di dati e garantendo che ogni thread abbia la propria variabile di loop i indipendente. Inoltre, c'è #pragma omp critical per evitare race condition.
Inoltre è presente la direttiva #pragma omp critical per evitare la race condition nel printf e nella variabile num_threads.
Dopo fatto questo :

- Avviare il batch : "heat_scaling.slurm"
- Avviare lo script python "heat_scaling_plot.py" (ovviamente modificatelo mettendo i vostri dati).
- Ecco il grafico generato :



Factorize :

Ho parallelizzato il ciclo for che chiama block_factorize attraverso la direttiva "#pragma omp parallel for". Ho poi calcolato il tempo di esecuzione del programma tramite l'utilizzo di omp_get_wtime() e del comando time.
Ho usato il tempo totale per trovare il numero di core hours necessari a fattorizzare il numero. Per farlo lo ho moltiplicato per il numero di core e diviso il tutto per 3600.
printf("CORE HOURS: %lf",(t_tot*8)/3600); ho gestito le stampe all'interno di block_factorize attraverso la direttiva "#pragma omp critical".

Per la parte di master slave ho fatto così :

```
#pragma omp parallel private(i) num_threads(2)

{
#pragma omp sections
{
    #pragma omp section // master
     {  while(response)
       {
        omp_set_lock(&master);
        response --;
        #pragma omp critical (print)
        printf("#MASTER: received request from %d  resp: %d \n", request, response);
        omp_unset_lock(&slave);
        omp_test_lock(&master);
      }
      master_stop=1;
      printf ("#MASTER STOP \n");
    }
    #pragma omp section //slave
```

```
    {
    #pragma omp parallel  num_threads(thr)
     {
      int block_num=0;
      float x;
      int t=omp_get_thread_num();
     while(! master_stop)
      {
        #pragma omp critical (print)
        printf ("#SLAVE %d: calling request of work.. \n",t);
        #pragma omp critical
        block_num=request_new_work(t);
        block_factorize (block_num);
      } // end while
     } // end parallel
    printf ("#SLAVE STOP \n");
   } // end section
 } // end sections
} // end parallel
```

- #pragma omp section // master: Inizia una sezione parallela per il master. All'interno di questa sezione, il codice è eseguito solo da un thread, il master.
- omp_set_lock(&master): Imposta un blocco sulla lock associata al master.
- #pragma omp critical (print): Garantisce che solo un thread alla volta possa eseguire la sezione di codice contenuta nel blocco, in questo caso, la stampa su console.
- omp_unset_lock(&slave): Rimuove il blocco sulla lock associata allo slave.
- omp_test_lock(&master): Testa se il master può acquisire la lock. Questa chiamata è potenzialmente errata, perché sembra non essere utilizzata correttamente (manca un parametro).
- #pragma omp section //slave: Inizia una sezione parallela per gli slave. All'interno di questa sezione, viene lanciata una nuova regione parallela con un numero di thread specificato dalla variabile thr.
- #pragma omp parallel num_threads(thr): Inizia una nuova regione parallela con un numero di thread pari al valore di thr.
- #pragma omp critical (print): Garantisce che solo un thread alla volta possa eseguire la sezione di codice contenuta nel blocco di stampa su console.
- printf("#SLAVE %d: calling request of work.. \n",t): Stampa un messaggio che indica che uno slave sta richiedendo lavoro.
- #pragma omp critical: Garantisce che solo un thread alla volta possa eseguire la sezione di codice successiva.

Laboratorio: mpirun e mpicc

Consegna :

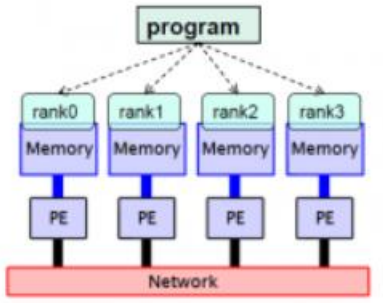
mpirun e mpicc sono i comandi principali per l'utilizzo del modello a memoria distribuita con MPI nei sistemi HPC.
Connettersi a login.hpc.unipr.it, create la directory di lavoro
mkdir -p ~/HPC2223/mpi/mpirun/
in cui copiate gli esercizi da qui: /hpc/group/T_2022_HPCPROGPAR/mpi/mpirun/ e svolgete le esercitazioni proposte.
rsync -av /hpc/group/T_2022_HPCPROGPAR/mpi/mpirun/ ~/HPC2223/mpi/mpirun/

MPI sul cluster

Sul cluster HPC sono installate le seguenti implementazioni di MPI-1 MPI-3 MPI-4 in openMPI e Intel-MPI:

MPI	Compiler	module load
openmpi (1.10.7)	gcc5 (5.4.0)	gnu openmpi
openmpi3 (3.1.4)	gcc7 (7.3.0)	gnu7 openmpi3
openmpi3 (3.1.4)	gcc8 (8.3.0)	gnu8 openmpi3
intelmpi3.1 (2019.5.281)	intel-compiler (2019.5)	intel impi
openmpi3 (3.1.4)	intel-compiler (2019.5)	intel openmpi3

mpirun
mpirun è il comando che si occupa della creazione e dell'esecuzione dei processi.
La gestione dei processi sui diversi nodi avviene tramite demoni (Orted in openMPI) che comunicano tra loro via ssh (o altri protocolli di rete).
mpirun realizza il modello SPMD (Single Program Multiple Data): lo stesso programma viene eseguito su istanze multiple dette task.
Ogni task ha la propria memoria e i propri dati. Ogni task è identificato da un numero intero denominato rank nel range [0, N-1].



Il numero di task e la loro distribuzione sui nodi è definita attraverso le opzioni di mpirun.
Le opzioni non sono standardizzate e possono variare tra le diverse implementazioni e versioni.
mpirun può essere integrato in un eventuale Job Manager, come Slurm, che può prendere il controllo dell'esecuzione.

Opzioni principali di mpirun :

- -np -n (openmpi e intel) : numero di processi da attivare
- -ppn (intel) -npernode (openmpi) : processi per nodo
- - host (intel e openmpi) : lista dei nodi su cui attivare i processi
- - hostfile (openmpi) -machine (intel) : file con l'elenco degli host e il numero di slot per host

Il **modello MPMD** è comunque possibile, ad esempio:
mpirun -np 4 hostname : -np 2 date
Il rankfile di openmpi consente di legare processi a specifici core o gruppi di core. Questo è particolarmente utile nelle architetture NUMA.
Analizzare ed eseguire **hostname.sh**
Esecuzione di mpirun via SLURM
Grazie all'integrazione tra Slurm e mpirun, le risorse selezionate da Slurm vengono automaticamente comunicate a mpirun
tramite dei wrapper che adattano la sintassi di Slurm alle diverse implementazioni MPI.
Principali opzioni Slurm per la selezione delle CPU:
https://www.hpc.unipr.it/dokuwiki/doku.php?id=calcoloscientifico:userguide#main_slurm_options
L'elenco delle risorse assegnate può essere visualizzato consultando le variabili di ambiente.
Le variabili utili per l'esercizio sono:
$SLURM_JOB_NODELIST (elenco dei nodi assegnati da Slurm)
$SLURM_JOB_CPUS_PER_NODE (numero di cpu assegnate per nodo)

Analizzare ed eseguire lo script **hostname.slurm**
Sottomettere il job:

sbatch hostname.slurm
Verificare lo stato in coda:
hpc-squeue # mostra tutti i job in coda o esecuzione
hpc-squeue -u $USER # mostra solo i miei job

mpicc
MPI fornisce una libreria per lo scambio di messaggi tra i processi.
Per utilizzare le primitive della libreria si utilizza un compilatore specifico che è un wrapper del compilatore di base.

Linguaggio	Default Compiler	MPI wrapper
Gnu C	gcc	mpicc
Gnu C++	g++	mpicxx o mpic++
fortran	gfortran	mpifc
Intel C	icc	mpiicc
Intel C++	icpc	mpiicpc

Compilazione ed esecuzione di mpi_latency.c
Rivediamo il job mpi_latency.c che abbiamo utilizzato per la valutazione delle performance di rete.
Compiliamo il programma con i compilatori GNU e Intel ed eseguiamo il programma utilizzando le opzioni di mpirun per la selezione delle risorse.
Vedi lo script mpi_lat.sh

Adattiamo lo script per la sua esecuzione attraverso il job manager Slurm. Vedi lo script mpi_lat.slurm
Sottomette lo script:
sbatch mpi_lat.slurm
hpc-squeue -u $USER

Consegna

Dopo aver completato l'esercitazione copiare i file su didattica-linux, quindi verificate la visibilità della directory via http.

Svolgimento:

Comando mpirun:

Serve a mettere in esecuzione i programmi che deve essere messo in esecuzione su più istanze con il modello SPMD (Single Program Multiple Data).
Si ha un programma unico da mettere in esecuzione su diversi processi in modo che ogni processo abbia a disposizione la propria memoria.
Ogni task ha la propria memoria e i propri dati.
Ogni task è identificato da un numero intero denominato rank nel range $[0, N-1]$.
mpicc serve per avviare programmi che usano l'API mpi.

Analisi file hostname.sh : È uno script bash dove attraverso tre cicli while avvia con il comando "mpirun" avvia i nodi con a sua volta sono specifica i numeri di task dopo il carattere ":" contenuti nei file : "machine.txt", "hostfile.txt" e "rankfile.txt". Per esempio dal primo file viene eseguito tra i vari comandi : "mpirun wn54:2", che significa eseguire due processi mpi sul nodo wn52.

hostname.slurm : Eseguirlo con il comando "sbatch hostname.slurm" e verificare lo stato del batch avviato con il comando "hpc-squeue -u $USER" (così mostro solo i miei job).
Analizzando questo file viene avviato il comando "mpirun hostname", che mostra i nomi di host associati a ciascun nodo coinvolto nell'esecuzione parallela.

mpi_latenci.c :
Il programma viene eseguito con diverse istanze e, tramite MPI, le varie istanze comunicano tra loro.
Bisogna includere la libreria

```
#include "mpi.h"
   //...
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if (rank == 0 && numtasks != 2) {
      printf("Number of tasks = %d\n",numtasks);
      printf("Only need 2 tasks - extra will be ignored...\n");
   }
   MPI_Barrier(MPI_COMM_WORLD); //si va aventi solo se tutti sono arrivati alla
                                //barriera
```

1) Inizializzazione MPI:
Il programma inizia con l'inizializzazione di MPI utilizzando la funzione MPI_Init.
Attenzione di informazioni sulle dimensioni e il rango del comunicatore:

Viene ottenuto il numero totale di processi (numtasks) e il rango del processo corrente (rank) all'interno del comunicatore predefinito (MPI_COMM_WORLD).

Controllo del numero di processi: Se il rango è 0 e il numero totale di processi non è esattamente 2, viene stampato un messaggio informativo. Questo programma è progettato per funzionare con esattamente due processi, quindi qualsiasi processo aggiuntivo viene ignorato.

Sincronizzazione dei processi: Tutti i processi si sincronizzano utilizzando MPI_Barrier per garantire che tutti siano pronti prima di iniziare la misurazione della latenza.

Invio di un messaggio

```
dest = 1;
source = 1;
for (n = 1; n <= reps; n++) {
   T1 = MPI_Wtime();     /* start time */
   /* send message to worker - message tag set to 1.  */
   /* If return code indicates error quit */
   rc = MPI_Send(&msg, 1, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
   /* Now wait to receive the echo reply from the worker  */
   /* If return code indicates error quit */
   rc = MPI_Recv(&msg, 1, MPI_BYTE, source, tag, MPI_COMM_WORLD,
                 &status);
   T2 = MPI_Wtime();
}
//...
MPI_Finalize();
```

Misurazione del tempo:
Prima e dopo l'invio del messaggio, il programma utilizza la funzione MPI_Wtime per ottenere il tempo corrente in secondi, permettendo la misurazione del tempo trascorso durante la comunicazione.

1. `dest`e`source` vengono entrambi impostati su 1. Questi rappresentano i ranghi dei processi MPI a cui il messaggio viene inviato e da cui viene ricevuto. In questo caso, il messaggio viene inviato e ricevuto tra i processi con rango 1.

2. Il ciclo `for` viene eseguito `reps` volte, dove `reps` è il numero di iterazioni del

test di latenza.

3. `MPI_Wtime`viene utilizzato per misurare il tempo iniziale (`T1`) prima dell'invio del messaggio.

4. Viene inviato un messaggio di 1 byte (`&msg`) al processo con rango `dest` utilizzando `MPI_Send`. Il tag del messaggio è impostato su 1 (`tag`). La funzione restituisce un codice di errore (`c`), e se questo codice indica un errore, il programma termina immediatamente.

5. Successivamente, il processo attende la ricezione di una risposta dal processo con rango 'source' utilizzando `MPI_Recv`. Anche in questo caso, la funzione restituisce un codice di errore (`c`), e il programma termina in caso di errore.

6. Infine, `MPI_Wtime` viene utilizzato per misurare il tempo finale (`T2`) dopo la ricezione della risposta.

Infine, avviare lo script “mpi_lat.slurm” per avviare il programma su più nodi e visualizzare i risultati, vedrete che verranno misurate le velocità per ciascun nodo avviato nel batch.

Laboratorio: MPI base

Consegna :

Riferimenti: slide mpi.pdf
Copiare in mpi/base il contenuto della directory: /hpc/group/T_2022_HPCPROGPAR/mpi/base/
rsync -av /hpc/group/T_2022_HPCPROGPAR/mpi/base/ ~/HPC2223/mpi/base/
La directory contiene alcuni esercizi dimostrativi dei principali schemi di comunicazione MPI, tra cui tutti gli esercizi riportati nelle slide MPI.pdf.
Compilare gli esercizi ed eseguirli con mpirun. Esempio:
module load intel impi
mpiicc 1-helloworld.c -o 1-helloworld
mpirun 1-helloworld

Esercizi:
Copiare i programmi *_todo.c in *.c (rimuovendo quindi _todo) e completarli con le primitive di comunicazione MPI.
Verificare il funzionamento.
mpi_cpi.c
Copiare in mpi/cpi/ i file per il calcolo di PI Greco con MPI:
rsync -av /hpc/group/T_2022_HPCPROGPAR/mpi/cpi/ ~/HPC2223/mpi/cpi/

Tecnica di parallelizzazione di mpi_cpi.c: scomposizione del dominio 1D in sottodomini
- L'intervallo di integrazione tra 0 e 1 viene suddiviso tra i task
- se abbiamo N intervalli e p task possiamo suddividere il dominio in p sottodomini.
Ogni sottodominio è ampio range=N/p Ogni task si occupa della gestione dei dati del sottodominio:
for (i = mystart; i <= mystart+range ; i++)
- Al termine vengono sommati i diversi contributi con MPI_Reduce()
Compilare e testare il programma quindi determinare strong e weak scaling utilizzando il job manager Slurm.
Generate i file dati e i plot adattando Nome Cognome e Data.
Copiare tutti i file su didattica-linux. Ad esempio con :
rsync -av --chmod=D755,F644 ~/HPC2223/ ${USER}@studenti.unipr.it@didattica-linux.unipr.it:html/HPC2223/

Svolgimento :

Il primo programma da analizzare è un semplice Hello, World! eseguito su più nodi. Questo programma ci permette di vedere un esempio semplice di inizializzazione MPI.

```
#include <stdio.h>
#include "mpi.h"
#define MASTER 0

int main (int argc, char **argv) {
        int numtasks, taskid, len;
        char hostname[MPI_MAX_PROCESSOR_NAME];
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);   MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
        MPI_Get_processor_name(hostname, &len);
        printf ("Hello from task %d on %s!\n", taskid, hostname);
        if (taskid == MASTER)
          printf("MASTER: Number of MPI tasks is: %d\n", numtasks);
        MPI_Finalize();
}
```

MPI_COMM_WORLD è un comunicatore predefinito in MPI (Message Passing Interface) che rappresenta il gruppo di tutti i processi MPI che partecipano a un programma. In altre parole, è il comunicatore di default che include tutti i processi che sono stati avviati nel contesto MPI.
MPI_Comm_size fornisce il numero totale di processi nel comunicatore.
MPI_Comm_rank fornisce l'ID (o rank) del processo corrente nel comunicatore.

Testare il programma con i seguenti comandi :
module load intel impi
mpiicc 1-helloworld.c -o 1-helloworld
mpirun 1-helloworld

Adesso entrare nella cartella TODO e rimuovere da tutti i file di estensione .c la scritta "_todo" e completarli con le direttive delle comunicazioni in MPI

mpi_DataMining.c :

- il Task 0 invia a tutti l'intero x tra 0 e 10 (esempio 7)
- Tutti i processi generano ogni secondo un intero random tra 0 e 10
- chi trova il numero x invia un messaggio agli altri con il numero trovato e termina
- tutti gli altri processi ricevono e stampano il valore ricevuto e il mittente; quindi, terminano

```
int main (int argc, char *argv[]) {
  MPI_Status status;
  MPI_Request request;
  int  MPIrank, MPIsize;
  int r, i, x, flag,  received, found = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &MPIrank);
  MPI_Comm_size(MPI_COMM_WORLD, &MPIsize);
  srand((unsigned)time(NULL)*MPIrank); // inizializzazione del seme
  flag=0;
  if (MPIrank==0){
 x = 7;
  }  printf("Task %d/%d : RECEIVING x\n",MPIrank,MPIsize);
  MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
  // attiva la ricezione asincrona non bloccante di x
  MPI_Irecv(&received, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
  while(!flag)  {
    sleep(1);
    MPI_Test(&request, &flag, &status);
    r = rand() % 11 + 1;  // Genera un numero casuale tra 1 e 10
    printf ("Task %d/%d - x:%d  r:%d \n", MPIrank, MPIsize,x, r);
    if (r == x) {
      for(i = 0; i < MPIsize; i++)
        MPI_Send(&r, 1, MPI_INT, i, 42, MPI_COMM_WORLD);
      found = 1;
      printf ("Task %d/%d - FOUND %d, exiting ...\n", MPIrank, MPIsize, r);
      break;
    }
  }
  if(!found)
    printf ("Task %d/%d - RECEIVED %d from %d, exiting ...\n", MPIrank, MPIsize, received,  status.MPI_SOURCE);
  MPI_Finalize();
  return 0;
}
```

- Inizializzazione dell'ambiente MPI.

- Ogni processo ottiene il proprio rank (MPIrank) e il numero totale di processi (MPIsize).
- Il processo con rank 0 genera un numero casuale x (nel tuo caso, 7) e lo invia a tutti gli altri processi utilizzando MPI_Bcast.
- Ogni processo attiva la ricezione asincrona non bloccante di un numero da un qualsiasi processo utilizzando MPI_Irecv.
- Ogni processo entra in un ciclo while che continua finché un flag (flag) diventa vero. All'interno del ciclo, ogni processo genera un numero casuale r compreso tra 1 e 10.
- Ogni processo stampa il proprio rank, il numero generato r, e il valore di x.
- Se il numero casuale r è uguale a x, il processo condivida questo valore con tutti gli altri processi utilizzando MPI_Send e imposta found a 1. Successivamente, il processo esce dal ciclo.
- Se il flag è ancora falso, significa che nessun processo ha trovato il numero cercato. In tal caso, il processo stampa un messaggio indicando che ha ricevuto un valore da un altro processo e poi esce.

Testare il tutto con i comandi :

- mpiicc mpi_DataMining.c -o mpi_Datamining
- mpi_Datamining

mpi_sendrecv_2 :
I processi 0 e 1 si scambiano il rank. Utilizzare MPI_Sendrecv() :

```
if (rank == 0) {
  data_send = rank;
  MPI_Sendrecv(&data_send, 1, MPI_INT, 1, 33, &data_recv, 1, MPI_INT, 1, 33, MPI_COMM_WORLD, &status);
  printf("Process 0 receives %d from process 1.\n", data_recv);
 }
 if (rank == 1) {
  data_send = rank ;
  sleep(1);
  MPI_Sendrecv(&data_send, 1, MPI_INT, 0, 33, &data_recv, 1, MPI_INT, 0, 33, MPI_COMM_WORLD, &status);
  printf("Process 1 receives %d from process 0.\n", data_recv);
 }
```

Testare il tutto con i comandi :

- mpiicc mpi_sendrecv_2.c -o mpi_sendrecv_2
- mpi_sendrecv_2

mpi_sendrecv_n.c :
I processi 0 e 1 si scambiano il rank. Utilizzare MPI_Sendrecv() :

```
MPI_Sendrecv(&data_send, 1, MPI_INT, next_rank, 666, &data_recv, 1, MPI_INT, prev_rank, 666, MPI_COMM_WORLD, &status );
```

Testare il tutto con i comandi :

- mpiicc mpi_sendrecv_n.c -o mpi_sendrecv_n
- mpi_sendrecv_n

mpi_cpi :
Copiare in mpi/cpi/ i file per il calcolo di PI Greco con MPI:
rsync -av /hpc/group/T_2022_HPCPROGPAR/mpi/cpi/ ~/HPC2223/mpi/cpi/
Dopo aver visto i metodi base di MPI, si può già adattare uno dei primi programmi paralleli che abbiamo visto per operare in memoria distribuita. In questo caso adattiamo il programma del calcolo di π da

approssimazione numerica.
La parallelizzazione, come per il caso a memoria condivisa, avviene suddividendo il dominio delle iterazioni in più sottodomini, 1 per task. Dovendo utilizzare il paradigma SPMD, ogni task individua il proprio sottodominio in base al proprio rank.

```
// MPI INIT
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&mpiTasks);
MPI_Comm_rank(MPI_COMM_WORLD,&mpiRank);
MPI_Get_processor_name(hostname,&namelen);

// DEFINE SUBDOMAIN
long long int rangeLocal=n/(mpiTasks);
long long int startLocal=rangeLocal*mpiRank;
```

Ognuno dei task effettua quindi l'approssimazione numerica all'interno del proprio sottodominio (e per motivi di testing misuriamo il tempo):

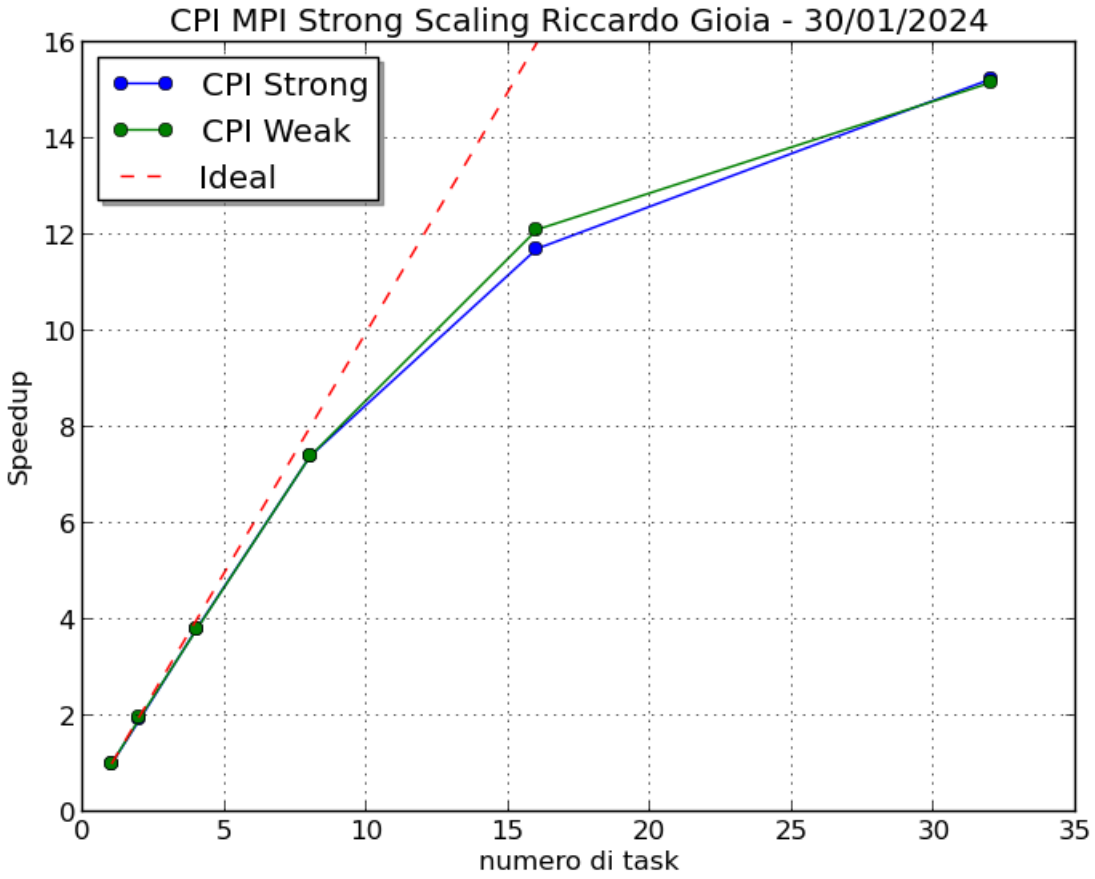
```
startwtime = MPI_Wtime();  // START TIMER
h   = 1.0 / (double) n;
sum = 0.0;
for (i = startLocal; i <= startLocal+rangeLocal ; i++){
                x = h * ((double)i - 0.5);
                 sum += f1(x);
}
piLocal = 4 * h * sum;
endwtime = MPI_Wtime();  // STOP TIMER
```

A questo punto si possono sommare le approssimazioni su tutti i sottodomini tramite riduzione. Si noti che il task eletto responsabile della parte finale del programma è quello di rank 0, perciò la riduzione avverrà su di lui e quindi tutti i task comunicheranno con lui la propria approssimazione.
Infine si stampa il risultato del programma.

```
// MPI COMMUNICATION
MPI_Reduce(&piLocal, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
// MAIN MPI TASK PRINT RESULTS
if (mpiRank==0) {

              printf("#Funct  Inter  pi    error   time numtasks  hostname \n");
               printf("%d %lld  %.10f %.10e %.4f  %d # %s \n", nf, n, pi, fabs(pi-PI),endwtime-startwtime, mpiTasks ,hostname);

}

// MPI FINALIZE
MPI_Finalize();
```

Ora bisogna eseguire questi comandi per avviare e testare il programma e per avviare il batch e generare il grafico :

- mpiicc mpi_cpi.c -o mpi_cpi
- mpi_cpi
- sbatch mpi_cpi_scaling.slurm
- python mpi_cpi_scaling.py (Modificare prima questo file con i propri dati e la propria data).



Inviare tutto alla macchina linux con il seguente comando “rsync -av --chmod=D755,F644 ~/HPC2223/ ${USER}@studenti.unipr.it@didattica-linux.unipr.it:html/HPC2223/”.

Laboratorio: Applicazioni MPI

Consegna :

Parallelizzare con MPI le applicazioni heat e factorize.
Creare le directory HPC2223/mpi/heat e HPC2223/mpi/factorize e , partendo dalla versione seriale, creare la versione parallela con MPI. Aggiungere
- l'intestazione #include <mpi.h>
- l'inizializzazione dell'ambiente MPI: MPI_Init(&argc, &argv);
- la scomposizione di dominio
- le comunicazioni MPI

- la ricomposizione del dominio
- la chiusura dell'ambiente MPI: MPI_Finalize();
- Dove necessario determinare e stampare i tempi di calcolo. Esempio: t1=MPI_Wtime();
- Valutare se le prestazioni ottenute corrispondono alle attese.

HEAT
Esempio di scomposizione di dominio: scomposizione delle righe in sottodomini 1D
Il numero complessivo di righe che possiamo chiamare WNY (Whole NY) viene suddiviso per il numero di task (+2 per le lo scambio dei bordi tra task adiacenti). Ogni task gestisce quindi un numero di righe NY pari a:

```
NY = WNY/mpi_size+2;          // NY: numero righe per task
```

Ogni task esegue le stesse operazioni del programma seriale, ma su un numero inferiore di righe.
Comunicazione MPI:
Al termine di ogni iterazione è necessario che ogni task scambi le righe di bordo con i task adiacenti.
Al termine della simulazione occorre ricomporre il dominio sul rank 0.
Realizzare i plot per lo strong scaling, con righe da 2048 punti e 1000 iterazioni.
Eventualmente confrontare le prestazioni quando i task sono allocati su un singolo nodo o distribuiti su due nodi. Quali sono i risultati attesi?

Esempi di script per l'esecuzione delle scaling e per il plotting:

rsync -av /hpc/group/T_2022_HPCPROGPAR/mpi/heat/ ~/HPC2223/mpi/heat/

Valutare ed eventualmente implementare le seguenti estensioni:
Overhead di comunicazione: riduzione attraverso la sovrapposizione comunicazione /computazione.
Checkpointing. Ogni simulazione può disporre di un tempo di calcolo limitato. Può essere utile la possibilità di salvare sul sistema di storage lo stato di avanzamento della simulazione in modo da poterlo riprendere in tempi successivi.
FACTORIZE
Il programma determina uno dei 2 fattori primi del modulo m noto.
È possibile utilizzare una versione da completare disponibile qui:
rsync -av /hpc/group/T_2022_HPCPROGPAR/mpi/factorize/ ~/HPC2223/mpi/factorize/

Il programma mpi_factorize_todo.c (da copiare in mpi_factorize.c) contiene una versione MPI che funziona con un solo task, ma può essere completato con le primitive MPI necessarie per il parallelismo.

Il dominio dei possibili numeri primi da testare viene suddiviso in blocchi di dimensione fissa; ogni blocco e' identificato da un indice (block_idx). I numeri di un blocco vengono testati dalla funzione block_factorize

(block_idx), che ritorna 1 se trova il numero , altrimenti torna 0.
Ad ogni task viene assegnata una sequenza contigua di blocchi (da firstBlock a lastBlock):

```
for (block_idx=lastBlock; block_idx >= firstBlock ; block_idx--)
   if ( block_factorize (block_idx) )
    {
    // stampa il numero trovato
    break;
    }
```

La ricerca nei blocchi può avvenire in qualsiasi ordine e quindi in parallelo su diversi task, ma è necessario programmare un sistema di comunicazione in cui il task che trova il numero avvisa gli altri task. Possibile modello:

```
mpi_Irecv( ...., MPI_ANY_SOURCE, .. ); // ogni task attiva la ricezione asincrona da qualsiasi altro task
for (block_idx=lastBlock; block_idx >= firstBlock ; block_idx--)
{
 MPI_Test (...);  // ogni task prima di analizzare un nuovo blocco testa l'eventuale ricezione del messaggio
 if (messaggio arrivato) break;
 if ( block_factorize (block_idx))
  {
  // Invia un messaggio a tutti i task
  // stampa in numero trovato
  break;
  }
}
```

Realizzare e testare il programma.
Al termine copiare programmi e dati su didattica-linux.

Svolgimento :

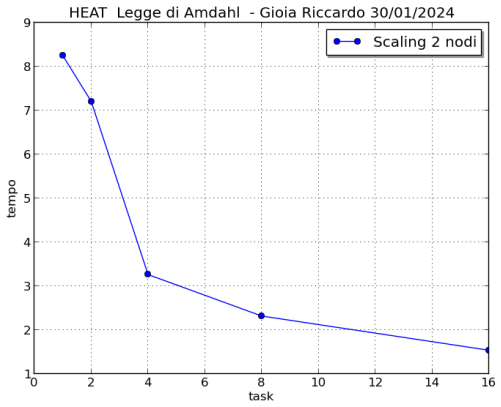
Heat :

Come abbiamo fatto sulla analoga versione di openMP precedente, il dominio viene suddiviso in diversi intervalli di righe con dimensioni uniformi.
Analogamente al caso openMP, la prima e l'ultima riga dell'intervallo fungono da buffer per i task adiacenti.
Dopo aver suddiviso il dominio, ciascun task esegue i calcoli in modo simile a quanto visto nel caso seriale.
Le fasi del programma sono :

- Inizializzazione del MPI: MPI_Init viene utilizzato per inizializzare l'ambiente MPI. I rank MPI e la loro dimensione vengono ottenuti utilizzando MPI_Comm_rank e MPI_Comm_size.
- Scomposizione del dominio: Il dominio di simulazione viene suddiviso tra i rank MPI. La variabile mpi_rank rappresenta l'identificatore del rank corrente, e mpi_size rappresenta il numero totale di rank. La scomposizione delle righe viene gestita assegnando un numero di righe NY a ciascun rank.
- Inizializzazione dei buffer: Vengono allocati e inizializzati gli array h_T_old e h_T_new che contengono le temperature al passo temporale corrente e successivo rispettivamente. Viene allocato anche l'array h_T_whole per contenere l'intero dominio gestito dal rank 0.
- Iterazioni di Jacobi con scambio di dati MPI: Il cuore del programma è l'iterazione di Jacobi implementata nella funzione Jacobi_Iterator_CPU. Le iterazioni avvengono per un numero massimo specificato (MAX_ITER). Viene effettuato uno scambio periodico delle righe di confine tra i rank MPI utilizzando MPI_Sendrecv. Le operazioni di Jacobi vengono eseguite su porzioni del dominio assegnate a ciascun rank.
- Misurazione del tempo: Vengono misurati il tempo totale di esecuzione della simulazione e alcuni parametri della simulazione. Nel rank 0, il tempo totale e altri parametri vengono stampati su standard output.

- Raccolta dei dati da parte del rank 0: Dopo le iterazioni, i dati dei singoli rank vengono raccolti nel rank 0 utilizzando MPI_Gather, e il risultato viene visualizzato tramite la funzione print_colormap se il rank è 0.
- Liberazione della memoria e chiusura di MPI: Gli array allocati vengono liberati dalla memoria, e MPI viene chiuso tramite MPI_Finalize.

Lo strong scaling del programma al crescere dei task è mostrato a seguito (con matrice fissa 2048*2048): Successivamente avviare lo script slurm e dal file di estensione .dat generato, generare il grafico con lo script python presente :



Da questo grafico possiamo dire che al crescere dei task, diminuisce il tempo.

Factorize :

In questo programma, il dominio è suddiviso in base al numero di task MPI e ad ognuno di essi viene assegnato un intervallo di blocchi in cui effettuare la ricerca. Prima di entrare nel ciclo for, viene richiamato MPI_Irecv, che avvia una ricezione non bloccante, destinata a interrompere il compito nel momento in cui viene individuato uno dei primi. A ogni iterazione del ciclo for, pertanto, viene verificata la possibile ricezione di un messaggio mediante MPI_Test.

```
MPI_Irecv(&block_found, 1, MPI_INT, MPI_ANY_SOURCE, 69, MPI_COMM_WORLD, &request);
t1=MPI_Wtime();
for (block_idx=lastBlock; block_idx >= firstBlock ; block_idx--)
{
    //Check for messages
    MPI_Test(&request, &flag, &status);
    //Modulus has been found by another rank
    if(flag){
        break;
    }
    //Modulus is found by the current rank
    if ( block_factorize (block_idx) )
    {
        block_found = 1;
        for(i = 0; i < MPIsize; i++)
            MPI_Isend(&block_found,1,MPI_INT,i,69,MPI_COMM_WORLD,&request);

        printf("Task %d/%d - block %d - FOUND: ", MPIrank, MPIsize, block_idx);
        BN_print_fp(stdout,F);
        printf("\n");
        break;
    }
    else
        printf("Task %d/%d - block %d - NOT found\n",  MPIrank, MPIsize, block_idx );
}
```

Infine copiamo tutto sulla macchina linux con il comando : “rsync -av --chmod=D755,F644 ~/HPC2223/ ${USER}@studenti.unipr.it@didattica-linux.unipr.it:html/HPC2223/”

Laboratorio: OpenMP + MPI

Consegna :

Programmazione del parallelismo multilivello

I moderni sistemi di calcolo ad alte prestazioni sono composti da architetture eterogene ottenute combinando diversi livelli di parallelismo hardware (architetture a memoria condivisa, distribuita, istruzioni vettoriali, acceleratori come le GPU, ecc).
Generalmente per ogni livello di parallelismo hardware esiste almeno una specifica metodologia di programmazione (linguaggio, libreria, set di istruzioni dedicate, ecc),
Alcune iniziative nello sviluppo di strumenti per la programmazione parallela hanno introdotto il supporto a diversi livelli di parallelismo Hardware.
Ad esempio openMP:
Nella versione 4 di openMP , a fianco della direttiva parallel, per la programmazione parallela di CPU a memoria condivisa, è stato introdotta la direttiva target per la programmazione di GPU e la direttiva SIMD per la programmazione delle istruzioni vettoriali. Vedi ad esempio:

https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf

https://www.ibm.com/docs/en/xl-c-and-cpp-linux/16.1.0?topic=parallelization-pragma-omp-target

https://www.ibm.com/docs/en/xl-c-and-cpp-linux/16.1.0?topic=pdop-pragma-omp-simd

OpenMP non prevede il supporto del message passing per la programmazione a memoria distribuita.
La metodologia di programmazione più comunemente utilizzata per la programmazione multilivello combina in uno stesso programma le diverse metodologie, specifiche di ogni livello e prende il nome di MPI+X, poiché generalmente viene inclusa almeno la programmazione a memoria distribuita con MPI, mente X sta per openMP, CUDA, o altro. Vedi ad esempio:

https://www.hpcwire.com/2014/07/16/compilers-mpix/

Vediamo come possiamo integrare MPI e openMP nelle applicazioni che abbiamo visto.

Programmazione ibrida mpi+openMP: heat

Nella programmazione ibrida (MPI + openMP) viene generalmente attivato un solo task MPI per nodo (o per socket) il quale si occupa della comunicazione con gli altri task, mentre il calcolo all'interno del nodo (o del socket) viene parallelizzato con openMP. Questa architettura consente di sfruttare la scalabilità multimodo di MPI ma riducendo al minimo l'overhead di comunicazione.

Creare la directory di lavoro ~/HPC2223/mpi+omp in cui copiate gli script

/hpc/group/T_2022_HPCPROGPAR/mpi+omp/heat/

e copiare le diverse versioni del programma heat: heat.c omp_heat.c mpi_heat.c .

Eseguire via slurm (heat.slurm) con un solo thread / processo per verificare la correttezza delle diverse implementazioni. Verificate che le condizioni di esecuzione siano le stesse: NX, NY (esempio 2048x2048) , Iterazioni (esempio 1000) e inizializzazione delle zone a temperatura costante (esempio init_top).

Dalle versioni MPI e OpenMP realizziamo la versione ibrida mpi+openMP:
- Creare il file mpi+omp_heat.c partendo da mpi_heat.c e aggiungete il parallelismo OpenMP:
Come nella versione openMP pura occorre distribuire su più thread le iterazioni sulle righe (j) della funzione di Jacobi:
Jacobi Iterator:

```
#pragma omp parallel for private (i)
 for(j=1; j<NY-1; j++)
   for(i=1; i<NX-1; i++) { ...}
```

Inoltre sistemare la stampa dei tempi al termine delle iterazioni:

```
if(mpi_rank == 0)  {
    #pragma omp parallel
    #pragma omp single
    fprintf(stderr,"%d %d %d %f %d %d \n", NX, NY, MAX_ITER, t2-t1, mpi_size, omp_get_num_threads() );
 }
```

La stampa viene fatta solo dal master (mpi_rank==0). Il master attiva la regione parallela (altrimenti la funzione omp_get_num_threads() ritornerebbe 1) ma la stampa è fatta da un solo thread (omp single),

Compilazione:

```
module load intel impi
mpicc -O2 mpi+omp_heat.c -o mpi+omp_heat  -fopenmp
```

Esempio di esecuzione a linea di comando con 2 processi e 4 thread per processo:

```
OMP_NUM_THREADS=4  mpirun -np 2  mpi+omp_heat -r 2048 -c 2048 1> /dev/null
```

Esecuzione batch:

```
sbatch  mpi+omp_heat.slurm
```

Il seguente script confronta le prestazioni delle diverse versioni, seriali e parallele, del programma heat.

```
sbatch  mpi+omp_heat_strong.slurm
```

Da notare che lo script slurm chiede due nodi (nodes=2) e 8 cores per nodo, poi lancia il programma con un task per nodo, mentre il numero di thread cresce da 1 fino a utilizzare tutti i cores disponibili, mantenendo fissa la dimensione del problema (strong scaling):

Realizzare il plot heat_mpi+omp_strong.png che compara lo strong scaling tra la versione MPI pura , la versione openMP e la versione ibrida con 2 nodi.

Programmazione ibrida mpi+openMP: factorize

Creare la directory di lavoro ~/HPC2223/mpi+omp/factorize in cui copiate le versioni openMP e MPI del programma e create la nuova versione mpi+omp_factorize.c :

suddividere il dominio in base al numero di task MPI, quindi i cicli di ogni task vengono distribuiti tra i thread openMP.

Analizzare ed eventualmente implementare una architettura di comunicazione asincrona in modo che il thread che trova in fattore primo distribuisca l'informazione ai thread dello stesso task e ai thread degli altri task, che, appena possibile dovranno interrompere l'attività.

Analizzare ed eventualmente implementare un modello master-slave in cui il task 0 gestisce l'elenco dei blocchi da testare e distribuisce i numero dei blocchi ai thread dei task MPI che ne fanno richiesta.

Al termine trasferire programmi e risultati su didattica-linux.

Svolgimento :

Heat :

Ho modificato il file “mpi_heat.c” e ci ho aggiunto delle direttive OpenMP nel metodo “Jacobi_Iterator_CPU” :

```
void Jacobi_Iterator_CPU(float *__restrict T, float *__restrict T_new, const int NX, const int NY) {
    int i, j;
    // --- Only update "interior" (not boundary) node points
    //Aggiunta direttiva OpenMP
    #pragma omp parallel for private(i)
    for (j = 1; j < NY - 1; j++)
        for (i = 1; i < NX - 1; i++)
        {
            float T_E = T[(i + 1) + NX * j];
            float T_W = T[(i - 1) + NX * j];
            float T_N = T[i + NX * (j + 1)];
            float T_S = T[i + NX * (j - 1)];
            T_new[NX * j + i] = 0.25 * (T_E + T_W + T_N + T_S);
        }
}
```

Per parallelizzare il ciclo for di questo metodo, inoltre ogni thread parallelo avrà la sua copia privata della variabile i per evitare conflitti di dati.

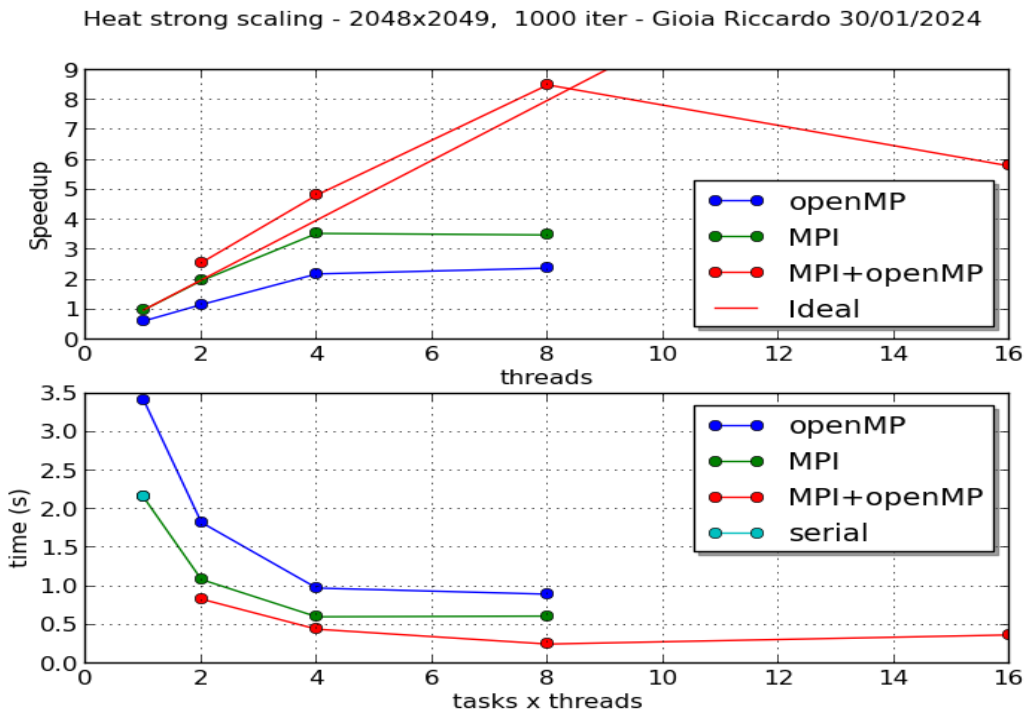
E anche in questo punto della applicazione :

```
if (mpi_rank == 0) {
  #pragma omp parallel
  #pragma omp single
  fprintf(stderr,"%d %d %d %f %d %d \n", NX, NY, MAX_ITER, t2-t1, mpi_size, omp_get_num_threads() );
}
```

- #pragma omp parallel: Questa direttiva OpenMP avvia una regione parallela. Ciò significa che il blocco di codice all'interno delle parentesi graffe successivi verrà eseguito da più thread in parallelo. La ragione per l'utilizzo di una regione parallela qui è dovuta al fatto che omp_get_num_threads() restituirebbe 1 al di fuori di una regione parallela, e l'obiettivo potrebbe essere quello di ottenere il numero effettivo di thread attivi durante l'esecuzione.
- #pragma omp single: Questa direttiva specifica che il blocco di codice successivo deve essere eseguito da un solo thread. Quindi, anche se la regione parallela è attivata, solo un thread eseguirà la stampa all'interno di fprintf.

Successivamente avviare il programma per testarlo, gli script slurm e python per generare il grafico :

- mpicc -O2 mpi+omp_heat.c -o mpi+omp_heat -fopenmp
- mpi+omp_heat
- sbatch mpi+omp_heat_strong.slurm
- python mpi+omp_heat_strong.py



Da questi grafici notiamo che il programma ibrido ha maggior speed-up ed è anche più veloce.

Factorize :

Il programma iniziale si basa sulla suddivisione dello spazio di ricerca in blocchi, come evidenziato in MPI. Si frammenta la regione di ricerca in intervalli di blocchi uniformi per i task. Questa fase viene gestita attraverso MPI. Successivamente, è possibile parallelizzare l'esplorazione dei blocchi disponibili nel task utilizzando OpenMP su più thread. In questo contesto, si verifica anche un'interazione tra le due librerie. Ciascun thread coinvolto nella ricerca del primo blocco controlla e, se necessario, comunica la sua scoperta mediante comunicazioni MPI non bloccanti.

Una volta fatto il programma, testatelo con il seguente comando e avviate lo script SLURM :

- - mpicc mpi+omp_factorize.c -o mpi+omp_factorize -lcrypto -lm -fopenmp
- mpi+omp_factorize
- sbatch mpi+omp_factorize.slurm

Infine inviare tutto alla macchina linux con il seguente comando : “rsync -av --chmod=D755,F644 ~/HPC2223/ ${USER}@studenti.unipr.it@didattica-linux.unipr.it:html/HPC2223/”.

Laboratorio: CUDA cpi

Consegna :

La consegna lo trovi nel PDF di nome “gpu-matrix” dal materiale didattico di Elly.
Scrivere un programma per calcolare cpi2 partendo dal file di esempio "cpi2/cpi2.cu".
Compilare con nvcc -arch=sm_60 cpi2.cu e lanciare con “slurm_launch_cpi2”, includere il calcolo dei tempi di esecuzione.

Svolgimento :

- Creare la cartella “gpu” con il seguente comando : “madri gpu” ed entrarci dentro : “cd gpu”
- Copiare la cartella di nome “cpi2” dentro : “cp -R /hpc/home/alessandro.dalpalu/gpu/cpi2 .”
- Entrare nella cartella ed avviare il programma “cpi2.cu” :
  - cd cpi2
  - module load cuda
  - nvcc cpi2.cu -o cpi2
  - cpi2
- Analizzando questo programma, ho constatato che calcola l’approssimazione del pi greco e ne calcola numero di blocchi, l'errore stimato, il tempo di esecuzione e la somma totale.
- Successivamente ho creato un altro programma di nome “cpi2_scaling.cu”in cui ci ho fatto le seguenti modifiche :
  1) Ho utilizzato la formula di Gregory-Leibniz per calcolare la formula approssimata di π : 4 * sqrt(1-x*x) (la vecchia funzione “f2()” che usavamo nel programma cpi.c in una delle prime consegne.
  2) il calcolo di x viene fatto in base all'indice globale del thread e al numero totale di blocchi e thread per blocco. Anziché utilizzare la posizione del thread rispetto al numero totale di rettangoli.
  3) Il mio programma accetta opzioni da riga di comando tramite la funzione options, che imposta il numero di blocchi in base all'opzione "-n". L’altro programma iniziale utilizza invece una variabile nblocks direttamente nel codice.
- Successivamente ho creato un nuovo script SLURM dove utilizzando il device della GPU con un ciclo for in cui testo il programma con un numero di blocchi di thread che va da 128 a 262144 e calcolo il risultato del mio programma a seconda del numero di blocchi dati in input e li converte in CSV:

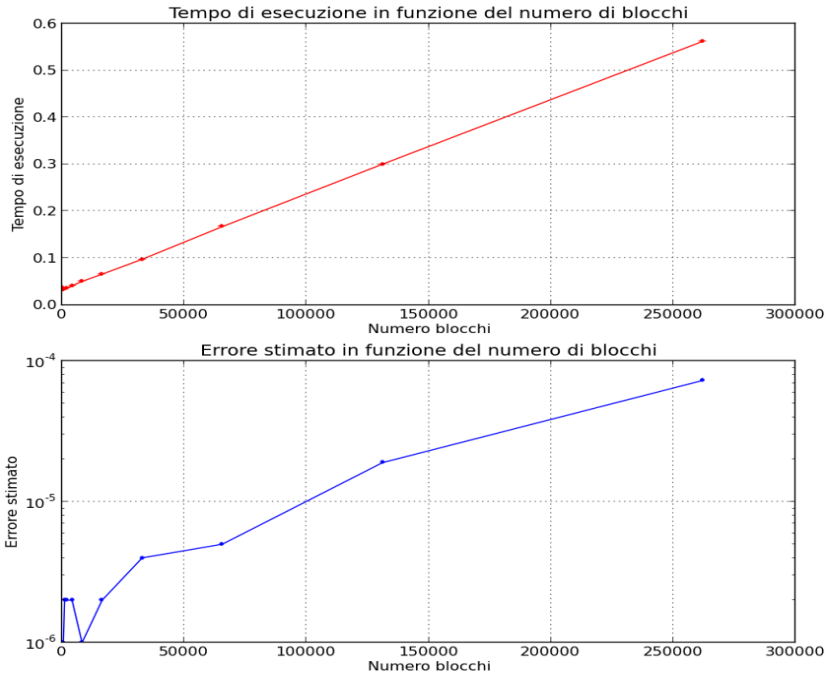
```
#!/bin/bash
#SBATCH --output=slurm_cpi2_scaling.out
#SBATCH --output=%x.o%j # Nome del file per lo standard output
##SBATCH --error=%x.e%j  # Se non specificato stderr e' rediretto su stdout
#SBATCH --partition=gpu
#SBATCH --qos=gpu
#SBATCH --nodes=1
#SBATCH --gres=gpu:a100:1
#SBATCH --time=0-00:10:00
#stampa il nome del nodo assegnato e argomenti
echo "#SLURM_JOB_NODELIST   : $SLURM_JOB_NODELIST"
echo "#CUDA_VISIBLE_DEVICES : $CUDA_VISIBLE_DEVICES"
#esegui il programma
module purge
module load cuda
nvcc -arch=sm_60 cpi2_scaling.cu -o cpi2_scaling
rm -f cpi2_scaling.dat
```

```
touch cpi2_scaling.dat
echo "Numero blocchi, Errore stimato, Tempo di esecuzione, Somma" >> cpi2_scaling.dat
for i in 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072 262144
do
   echo "# Esecuzione blocco $i"
   ./cpi2_scaling -n $i 2>> cpi2_scaling.dat
done
cat cpi2_scaling.dat | fgrep -v "#"  > output_scaling.csv
```

- Usare il seguente script python per generare il grafico :

```
import pandas as pd
import matplotlib as mlp
mlp.use('Agg')
import matplotlib.pyplot as plt
# Numero blocchi, Errore stimato, Tempo di esecuzione, somma
df = pd.read_csv("output_scaling.csv", sep=",", header=0, names=["n_blocchi", "err", "time", "sum"])
# devo effettuare 2 flot nella stessa immagine, blocchi tempo e blocchi errore
# primo plot
plt.figure(figsize=(10, 10))
plt.subplot(2, 1, 1)
plt.errorbar(df["n_blocchi"], df["time"], yerr=df["err"], marker=".", color="red")
plt.xlabel("Numero blocchi")
plt.ylabel("Tempo di esecuzione")
plt.grid(True)
plt.title("Tempo di esecuzione in funzione del numero di blocchi")
# secondo plot
plt.subplot(2, 1, 2)
plt.errorbar(df["n_blocchi"], df["err"], marker=".", color="blue")
plt.xlabel("Numero blocchi")
plt.ylabel("Errore stimato")
plt.grid(True)
plt.title("Errore stimato in funzione del numero di blocchi")
plt.yscale("log")
plt.savefig("cpi2_scaling.png")
```

- Infine ho avviato il batch e lo script python per generare il grafico con i seguenti comandi :
  - sbatch slurm_launch_cpi2.slurm
  - python cpi2_scaling.py



Da questi grafici possiamo affermare che il programma per il calcolo del pi-greco ha un ottima scalabilità perché lo speed-up rimane costante all'aumentare del numero di blocchi di thread.
Mentre l'errore stimato cresce all'aumentare dei blocchi.
Quindi in definitiva la GPU è più efficiente e più scalabile rispetto alla GPU.

Laboratorio: CUDA Matrix

Consegna :
Lanciare lo script go, i risultati saranno scritti su vari file.
Estrarre le prestazioni per CPU, Naive GPU, coalescing GPU e Tiling GPU.
Caricare i 4 files su didattica-linux. Adattare lo script in python (cpi2_scaling.py) per produrre un grafico con le 4 serie di dati.
Svolgimento :

- Scaricare il PDF di nome "gpu-matrix" dal materiale didattico di Elly.
- Copiare la cartella di nome "matrixMul" dentro gpu : "cp -R /hpc/home/alessandro.dalpalu/gpu/matrixMul ."
- Entrarci dentro : "cd matrixMul"
- Creare il file "slurm_launch_single" con il comando "cat slurm_launch_single" e inserire dentro :

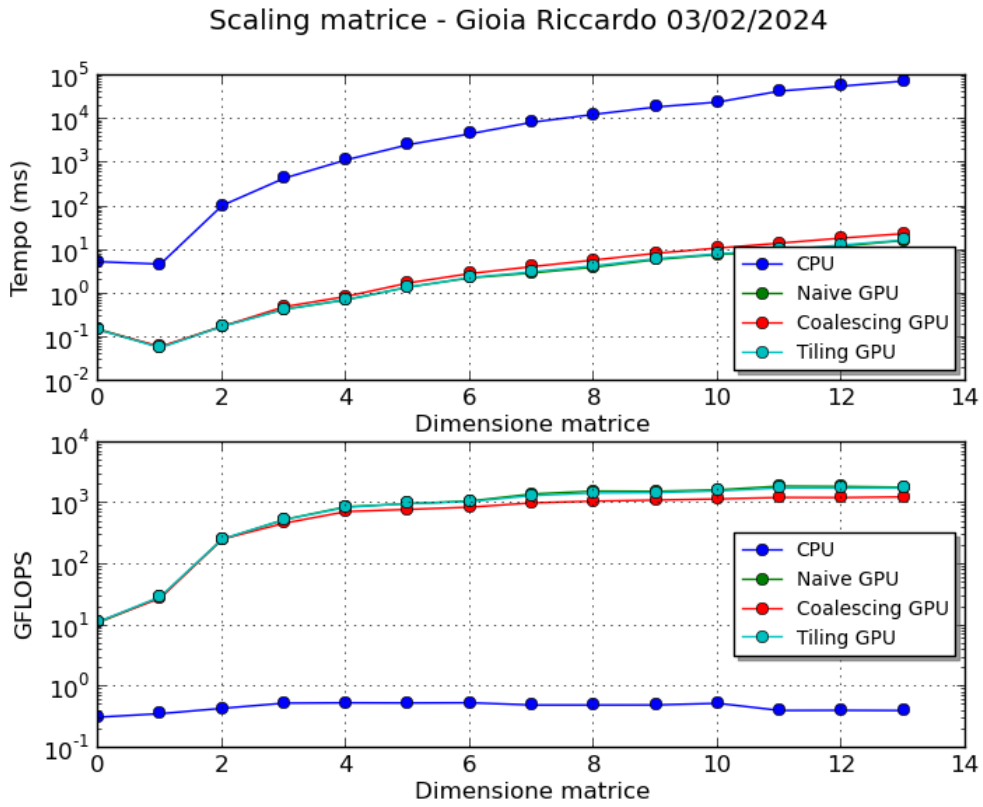
```
#!/bin/sh
# Richiedi un nodo con una gpu
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --mem=4G
#SBATCH --qos=gpu
#SBATCH --gres=gpu:a100:1
##SBATCH -–account=t_2022_hpcprogpar
##SBATCH --reservation= t_2022_hpcprogpar_20230525
# Dichiara che il job durera' al massimo 1 minuto
#SBATCH --time=0-00:01:00
#stampa il nome del nodo assegnato e argomenti
```

```
echo "#SLURM_JOB_NODELIST   : $SLURM_JOB_NODELIST"
echo "#CUDA_VISIBLE_DEVICES : $CUDA_VISIBLE_DEVICES"
echo "size A= $WA X $HA,B= $WB X $HB"
#esegui il programma
module load cuda
./bin/matrixMul -wA=$WA -hA=$HA -wB=$WB -hB=$HB
```

- Premere CTRL + D per salvare il file
- Eseguire il file “go” con il comando “go” e generare i 4 file di testo contenuti nella consegna
- cat *.o* | grep -A1 "Tiling GPU" | grep time > tiling.txt
- cat *.o* | grep -A1 "Naive CPU (Golden Reference)" | grep time > cpu.txt
- cat *.o* | grep -A1 "Naive GPU" | grep time > naive-gpu.txt
- cat *.o* | grep -A1 "Global mem coalescing GPU" | grep time > coalescing.txt
- Ho creato una cartella dei risultati e ci ho spostato i file dentro, poi ci ho generato uno script python per generare i grafici :

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from numpy import *
# Lettura dati dai .txt
cpu_data = genfromtxt('cpu.txt', dtype=None, delimiter=' ', usecols=(2, 5))
naive_data = genfromtxt('naive-gpu.txt', dtype=None, delimiter=' ', usecols=(2, 5))
coalescing_data = genfromtxt('coalescing.txt', dtype=None, delimiter=' ', usecols=(2, 5))
tiling_data = genfromtxt('tiling.txt', dtype=None, delimiter=' ', usecols=(2, 5))
# Verifica che tutti i set di dati abbiano la stessa lunghezza
assert len(cpu_data) == len(naive_data) == len(coalescing_data) == len(tiling_data), "I dati devono avere la
stessa lunghezza"

# Plot con figure e subplot
figure = plt.figure()
figure.suptitle("Scaling matrice - Gioia Riccardo 03/02/2024", fontsize="large")
# Plot tempo (ms)
subplot = figure.add_subplot(211)
subplot.grid()
subplot.set_xlabel('Dimensione matrice')
subplot.set_ylabel('Tempo (ms)')
subplot.set_yscale('log')
subplot.plot(cpu_data[:, 0], '-o', label='CPU')
subplot.plot(naive_data[:, 0], '-o', label='Naive GPU')
subplot.plot(coalescing_data[:, 0], '-o', label='Coalescing GPU')
subplot.plot(tiling_data[:, 0], '-o', label='Tiling GPU')

subplot.legend(loc='lower right', shadow=True, fontsize="small", numpoints=1)

# Plot gflops
subplot = figure.add_subplot(212)
subplot.grid()
subplot.set_xlabel('Dimensione matrice')
subplot.set_ylabel('GFLOPS')
subplot.set_yscale('log')
```

```
subplot.plot(cpu_data[:, 1], '-o', label='CPU')
subplot.plot(naive_data[:, 1], '-o', label='Naive GPU')
subplot.plot(coalescing_data[:, 1], '-o', label='Coalescing GPU')
subplot.plot(tiling_data[:, 1], '-o', label='Tiling GPU')
subplot.legend(loc='best', shadow=True, fontsize="small", numpoints=1)
# Salvataggio su file
figure.savefig('Confronto_Matrice.png')
```



Da questo grafico possiamo dire che la GPU è molto più performante della CPU e che la più performante è la GPU.

Laboratorio: CUDA Heat

Consegna 1 :

Aggiungere un timer per misurare il tempo di calcolo delle iterazioni dei kernel. Aggiungere in output il tempo preceduto dal simbolo # (per permettere a grep di generare il csv).

Modificare l'inizializzazione della matrice Initialize() per creare la propria configurazione (i valori >0 vengono imposti nella simulazione)

Consegna 2 :

1) Usare parametro da riga di comando per definire la dimensione della matrice (non 256 statica) e mostrare andamento dei tempi con un grafico per N=64,128,256,512,1024,2048.

2) Introdurre uso memoria shared per leggere T_old e mostrare l'andamento dei tempi al variare della dimensione della matrice.

3) cambio dimensione del blocco BLOCK_SIZE_X (4,8,16,32) e mostrare l'andamento dei tempi al variare della dimensione della matrice.

Svolgimento consegna 1:

- Scaricare il PDF di nome "gpuheat" dal materiale didattico di Elly.
- mkdir gpuheat
- cd gpuheat
- cp -R /hpc/home/alessandro.dalpalu/gpu/heat/* .
- Compilare e testare il programma : "nvcc -O2 heat_gpu.cu -o heat_gpu" e "heat_gpu".
- Come gli altri programmi heat fatti in passato, vedo che questo programma implementa una simulazione numerica di conduzione del calore utilizzando il metodo di Jacobi. La simulazione è eseguita su una griglia bidimensionale con una distribuzione iniziale di temperatura e applica iterativamente l'operazione di Jacobi per aggiornare la temperatura in ogni punto della griglia. Inoltre questo programma ha due kernel CUDA :
  - **Jacobi_Iterator_GPU** : esegue le iterazioni di Jacobi sulla GPU. Ogni thread della GPU si occupa di un punto nella griglia e utilizza i valori dei punti adiacenti per calcolare il nuovo valore del punto corrente.
  - **copy_constant** : copia i valori dei punti a temperatura costante nella griglia. Gestisce anche le condizioni periodiche ai bordi della griglia.
- Creare un'altra cartella per la consegna 1 e inserirci gli stessi file per poter lavorarci.
- Modificare il file di nome "heat_gpu.cu" come richiesto nella consegna:
  - Ho aggiunto il tempo impiegato per copiare la memoria costante (t_sum_copy_constant) e il tempo impiegato per l'iterazione di Jacobi (t_sum_jacobi) :

```
gettimeofday(&tempo,0);
t_start_copy_constant = tempo.tv_sec+(tempo.tv_usec/1000000.0);
copy_constant<<<dimGrid, dimBlock>>>(d_T, d_T_const, NX, NY);
gettimeofday(&tempo,0);
t_end_copy_constant = tempo.tv_sec+(tempo.tv_usec/1000000.0);
gettimeofday(&tempo,0);
t_start_jacobi = tempo.tv_sec+(tempo.tv_usec/1000000.0);
Jacobi_Iterator_GPU<<<dimGrid, dimBlock>>>(d_T, d_T_old, NX, NY);
gettimeofday(&tempo,0);
t_end_jacobi = tempo.tv_sec+(tempo.tv_usec/1000000.0);
t_sum_copy_constant += (t_end_copy_constant - t_start_copy_constant);
t_sum_jacobi += (t_end_jacobi - t_start_jacobi);
```

  - Ho personalizzato la configurazione anziché usare quella di default del metodo Init_Sq()
  - Aggiunto in output la stampa dei tempi per poter generare il CSV.

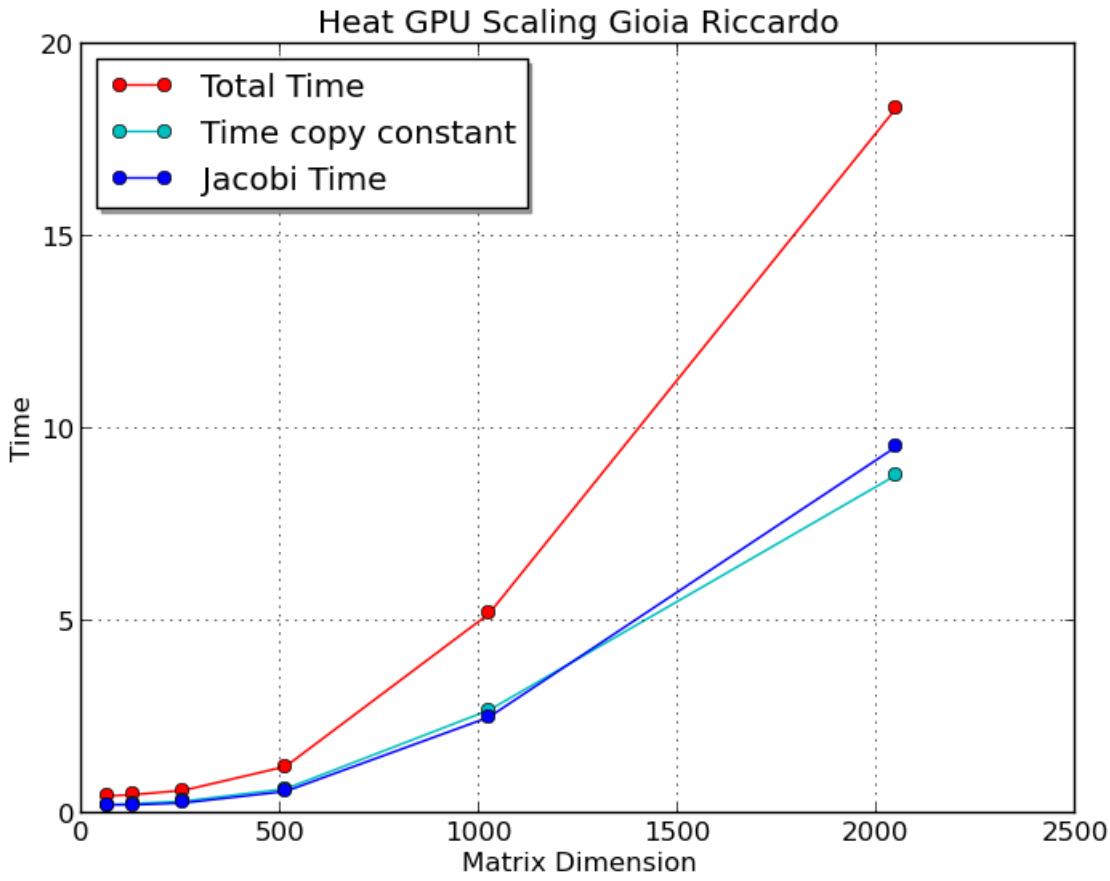
- Fatto lo script SLURM per poter generare il CSV dove valorizzo i valori di N così :
  #!/bin/sh

```
# Richiedi un nodo con una gpu
SBATCH --partition=gpu_guest
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --mem=4G
#SBATCH --qos=gpu
#SBATCH --gres=gpu:p100:1

# Dichiara che il job durera' al massimo 10 minuti
#SBATCH --time=0-00:10:00

#stampa il nome del nodo assegnato e argomenti
echo "#SLURM_JOB_NODELIST   : $SLURM_JOB_NODELIST"
echo "#CUDA_VISIBLE_DEVICES : $CUDA_VISIBLE_DEVICES"

module load cuda/11.6.0
nvcc -O2 heat_gpu.cu -o heat_gpu
#svuoto il contenuto del file ogni volta
echo -n > time_heat.csv
echo "#totale copy_constant Jacobi" >> time_heat.csv

for i in 64 128 256 512 1024 2048
do
./heat_gpu  -c $i -r $i  -s 100000 2>> time_heat.csv
done
```

- Fatto uno script python per poter generare il grafico :

```
import matplotlib
matplotlib.use('Agg')   # Backend per PNG (http://matplotlib.org/faq/usage_faq.html#what-is-a-backend )
import matplotlib.pyp

MATRIX_DIM = loadtxt("dim.txt")
data1 = loadtxt("time_heat.csv")
totale=data1[:,0]
copy_constant=data1[:,1]
jacobi=data1[:,2]
plt.title('Heat GPU Scaling Gioia Riccardo')
plt.grid()
plt.xlabel('Matrix Dimension')
plt.ylabel('Time')
plt.plot(MATRIX_DIM,totale,'ro-',label='Total Time')
plt.plot(MATRIX_DIM,copy_constant,'co-',label='Time copy constant')
plt.plot(MATRIX_DIM,jacobi,'bo-',label='Jacobi Time')
plt.legend(shadow=True,loc="best")
plt.savefig('heat_gpu_time.png')
```

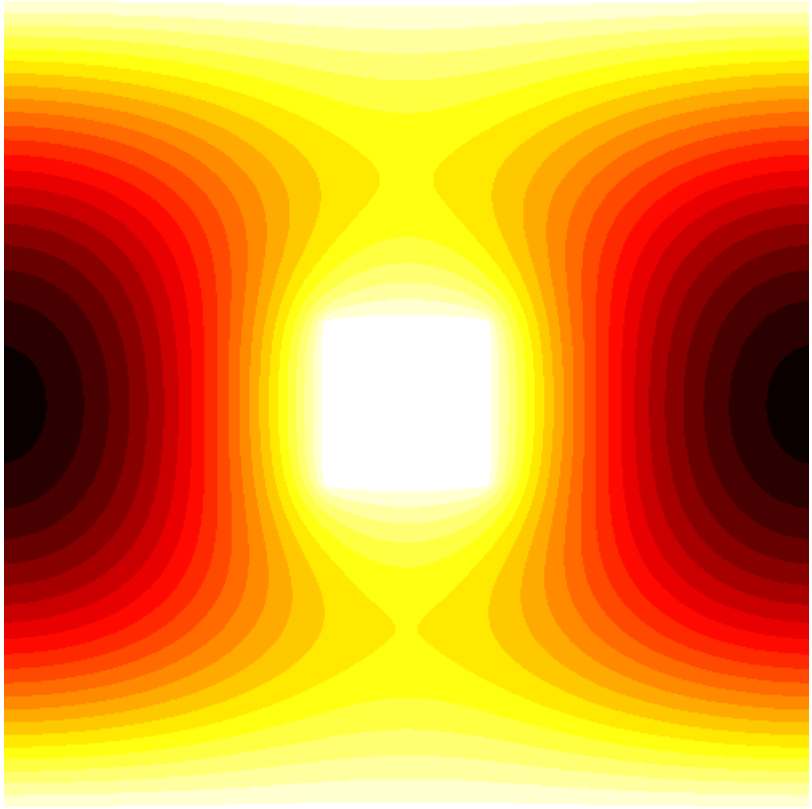
- Aggiunto un file di nome dim.txt in cui ci ho specificato i dati che vengono usati nel grafico e nello script con il seguente contenuto :
  64
  128
  256
  512
  1024
  2048
- Infine ho avviato lo script SLURM e generato il grafico :
  - sbatch heat_gpu.slurm
  - python heat_scaling.py



Da questo grafico possiamo dire che il kernel che esegue la funzionalità di Jacobi è più veloce.

Inoltre ho anche generato l'immagine con lo stesso script python e generato un output CSV separato nel programma per verificare se cambiando l'inizializzazione, l'immagine generata è cambiata e ne ho avuto la conferma.

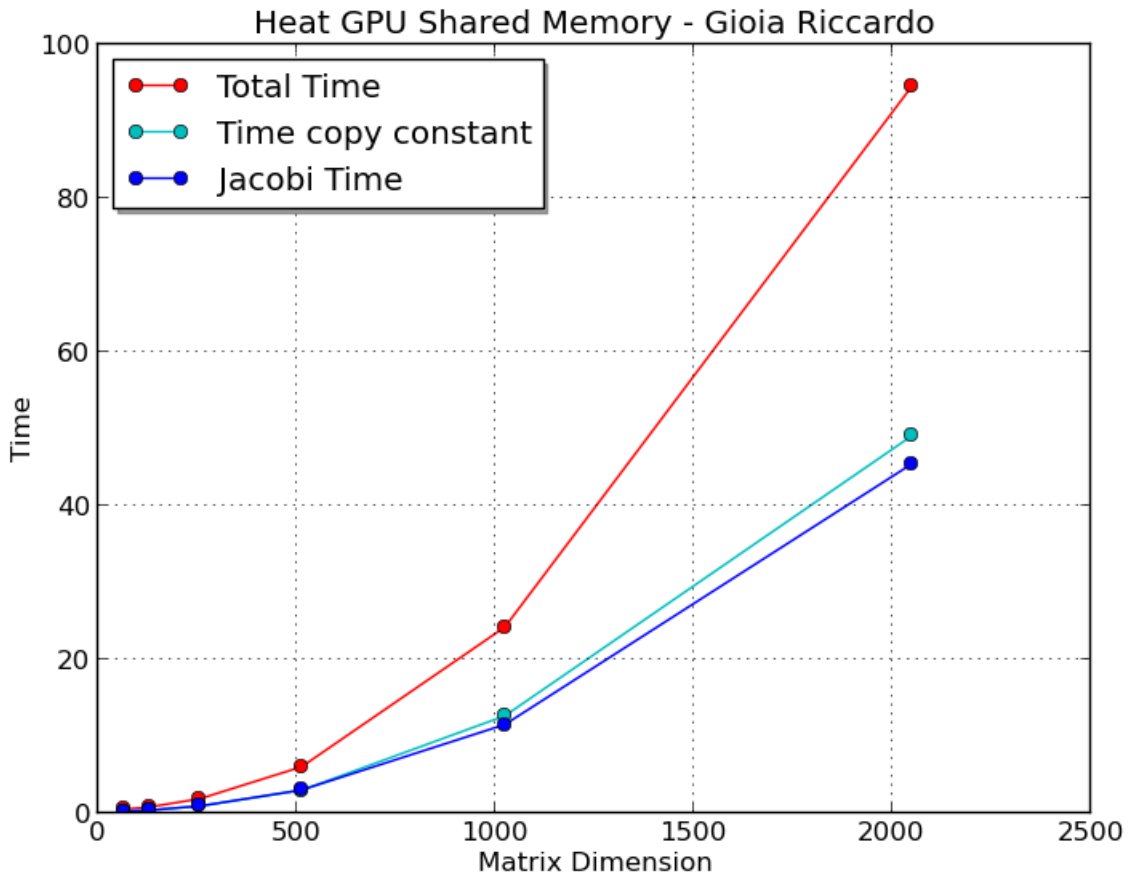
Con l’inizializzazione dei file del prof l’immagine veniva generata così :



Mentre con la mia inizializzazione l’immagine viene generata così :

Svolgimento consegna 2:

- Creare la cartella nuova ed entrarci “mkdir consegna2” e “cd consegna2”
- Copiarci i file dentro : “cp -R /hpc/home/alessandro.dalpalu/gpu/heat/* .”
- Ora modifico il metodo CUDA di JACOBI :
  1) Definisco la memoria condivisa :
  __shared__ float shr[BLOCK_SIZE_X * BLOCK_SIZE_Y];
  Serve per ridurre gli accessi alla memoria globale.
  2) Oltre agli indirizzi di memoria globale già presenti, ci ho aggiunto gli indici per la memoria condivisa.
  3) Oltre agli attributi dei punti cardinali di memoria globale già presenti, ci ho aggiunto anche quelli per memoria condivisa.
  4) Copia dell'elemento corrente dalla memoria globale alla memoria condivisa:
  shr[Cs] = T_old[Cg];
  5) Sincronizzazione di tutti i thread:
  __syncthreads(); -> Assicura che tutti i thread nel blocco abbiano completato la copia dei dati nella memoria condivisa prima di procedere con i calcoli.
  6) Vengono calcolati i valori circostanti anche tenendo conto di quando si trovano nella memoria condivisa.
  7) Calcolo la media e il risultato viene scritto nella memoria globale (T_new) :
  T_new[Cg] = (N + S + E + O) / 4;
- Ho copiato i file “dim.txt” e “heat_scaling.py” dalla consegna 1 per poter generare il grafico per l’andamento della dimensione della matrice per N=64,128,256,512,1024,2048 :



Anche in questa consegna, il kernel che esegue la funzione copy constant è più veloce.

- Adesso devo generare il grafico per verificare l'andamento del cambio dimensione del blocco BLOCK_SIZE_X (4,8,16,32).
  Ho fatto uno script SLURM e uno script python :
  SCRIPT SLURM :

```
#!/bin/sh
# Richiedi un nodo con una gpu
##SBATCH --partition=gpu_guest
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --mem=4G
#SBATCH --qos=gpu
#SBATCH --gres=gpu:p100:1

# Dichiara che il job durera' al massimo 10 minuti
#SBATCH --time=0-00:10:00

#stampa il nome del nodo assegnato e
echo "#SLURM_JOB_NODELIST   : $SLURM_JOB_NODELIST"
echo "#CUDA_VISIBLE_DEVICES : $CUDA_VISIBLE_DEVICES"
module load cuda/11.6.0

# Svuota il contenuto del file ogni volta
echo -n > time_heat_block.csv
echo "#totale copy_constant Jacobi" >> time_heat_block.csv
# Ciclo sui valori di BLOCK_SIZE_X e BLOCK_SIZE_Y
for i in 4 8 16 32
do
    # Modifica il file CUDA con i nuovi valori di BLOCK_SIZE_X e BLOCK_SIZE_Y
    sed -i "s/#define BLOCK_SIZE_X [0-9]*/#define BLOCK_SIZE_X $i/" heat_gpu_shared.cu
    sed -i "s/#define BLOCK_SIZE_Y [0-9]*/#define BLOCK_SIZE_Y $i/" heat_gpu_shared.cu

# Compila il programma CUDA
    nvcc -O2 heat_gpu_shared.cu -o heat_gpu_shared

    # Esegui il job SLURM
    ./heat_gpu_shared -c 64 -r 64 -s 100000 2>> time_heat_block.csv
done
```

Ho fatto un ciclo for in cui dinamicamente modifico gli attributi messi in define BLOCK_SIZE_X e BLOCK_SIZE_Y, visto che non potevo farlo come ho fatto negli altri modi da riga di comando tramite le option perché non si può fare con gli attributi aggiunti con la direttiva #define.

- Script python per poter generare il grafico :

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from numpy import *

MATRIX_DIM =  [4, 8, 16, 32]
data1 = loadtxt("time_heat_block.csv")
totale=data1[:,0]
copy_constant=data1[:,1]
jacobi=data1[:,2]
plt.title('Heat GPU Shared Memory - Andamento numero blocchi - Gioia Riccardo')
plt.grid()
plt.xlabel('Block size')
plt.ylabel('Time')
plt.plot(MATRIX_DIM,totale,'ro-',label='Total Time')
plt.plot(MATRIX_DIM,copy_constant,'co-',label='Time copy constant')
plt.plot(MATRIX_DIM,jacobi,'bo-',label='Jacobi Time')
plt.legend(shadow=True,loc="best")
plt.savefig('heat_gpu_time_shared_blocchi.png')
```

- Infine ho avviato lo script SLURM e generato il grafico con i seguenti comandi :
  - "sbatch heat_gpu_block.slurm"
  - “python heat_scaling_block.py”

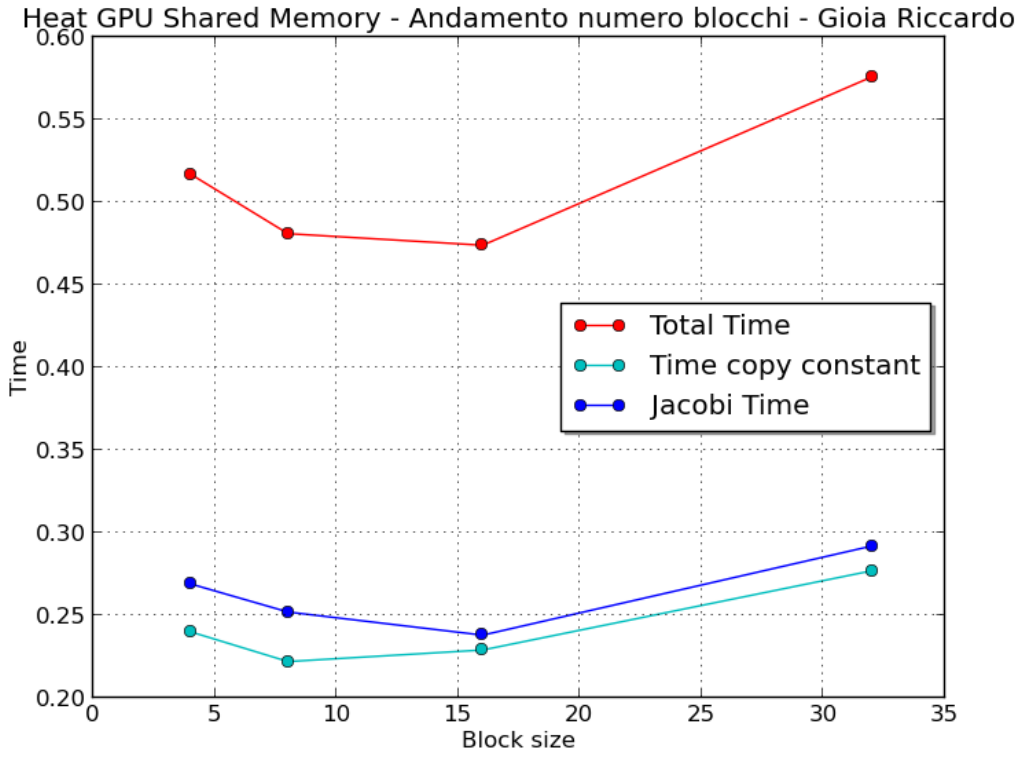


Grafico verifica andamento cambio dimensione del blocco BLOCK_SIZE_X (4,8,16,32).

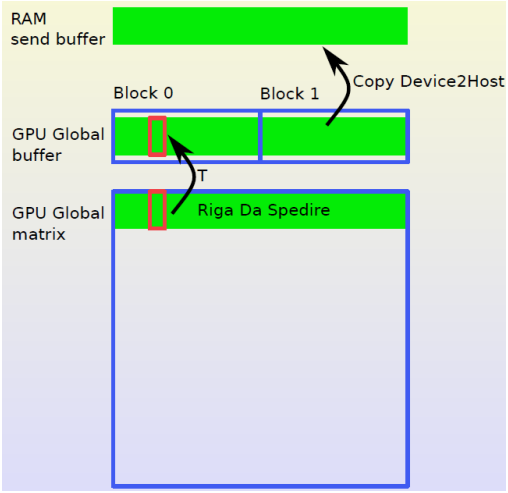
Laboratorio: Esercitazione GPU + MPI

Consegna :
Obbiettivi :

- Analisi programma GPU + MPI.
- Estensione del programma.

Programma di base :

- Heat transfer, matrice NX x 2NY
- Progettato per 2 GPU
- Comunicazione MPI per scambio dati di bordo
- Trasferimento GPU -> RAM (rank 0) -> MPI -> RAM (rank 1) -> GPU
- Output: ogni rank scrive un file
- Dataflow spostamento dati e kernel copia su buffer



Svolgimento :
Creare la cartella nuova e copiarci dentro i file :
- “mkdir gpumpiheat”, “cd gpumpiheat”, “cp -R /hpc/home/alessandro.dalpalu/gpu/mpi/* .”
Testare il programma :
“module load gnu openmpi cuda”
“nvcc main.cu -I/opt/ohpc/pub/mpi/openmpi-gnu/1.10.7/include \”, si aprirà un comando, digitare :
“-L/opt/ohpc/pub/mpi/openmpi-gnu/1.10.7/lib/ -lmpi” e premere INVIO.

Nella cartella ci sono due script SLURM :
1) “slurm_gpumpi_2x1.slurm” -> Esegue l’output del programma “main.cu” usando 2 nodi x 1 GPU con comando mpirun.
2) “slurm_gpumpi_1x2.slurm” -> Esegue l’output del programma “main.cu” usando 1 nodo x 2 GPU con comando mpirun.
- Eseguire uno degli script SLURM precedenti : “sbatch slurm_gpumpi_2x1.slurm”.
- Esportare i risultati (i file generati “test-1.txt” e “test-2.txt”) : “cat test-0.txt test-1.txt > my_file.csv”
- Avviare lo script python “python visualize.py” e visualizzare il file immagine generata :

Successivamente ho utilizzato anche l'altro script SLURM e funziona anche questo generando un immagine identica.

Analisi del programma :

È sempre lo stesso solito programma heat che simula la propagazione del calore su una superficie, però, oltre che CUDA viene utilizzato MPI.

Di seguito, spiego come avviene la comunicazione tra i processi MPI e le GPU nel codice.

1) Inizializzazione e suddivisione dei processi MPI:

- MPI viene inizializzato con MPI_Init.
- Viene ottenuto il nome del nodo e un elenco ordinato dei nodi tramite MPI_Get_processor_name e MPI_Bcast.
- Gli host vengono ordinati e i processi vengono suddivisi in base agli host con MPI_Comm_split.
- Viene calcolato il colore e il rank all'interno del nodo con MPI_Comm_rank e MPI_Comm_size.

Configurazione delle GPU:

- Il numero di GPU disponibili viene ottenuto con cudaGetDeviceCount.
- Le proprietà delle GPU vengono ottenute con cudaGetDeviceProperties.
- Viene assegnata una GPU a ciascun processo MPI nel nodo con cudaSetDevice.

Inizializzazione della temperatura e delle variabili:

- La funzione Initialize inizializza la matrice di temperatura in base al rank del processo.
- Viene allocata la memoria per le variabili utilizzate nella simulazione, sia sulla CPU che sulla GPU.
- Le matrici sulla CPU vengono copiate nella memoria GPU con cudaMemcpy.

Iterazioni di Jacobi sulla GPU:

- Le iterazioni di Jacobi vengono eseguite nella funzione Jacobi_Iterator_GPU sul dispositivo CUDA.
- In ogni iterazione, vengono eseguite operazioni di scambio dei dati tra processi MPI utilizzando MPI_Sendrecv e copy_send (kernel CUDA).
- I dati vengono quindi trasferiti tra CPU e GPU utilizzando cudaMemcpy.

Rilascio delle risorse:

- La memoria allocata viene liberata alla fine del programma con free e cudaFree.
- MPI viene chiuso con MPI_Finalize.