

Lab 5a - openMP base

<https://elly2023.smfi.unipr.it/mod/page/view.php?id=5002>

OBIETTIVO

I seguenti esercizi mostrano le caratteristiche di base principali di openMP.

Attività svolte

Aggiornamento moduli

- *Compilatore Intel:*
 - o "module load intel" carico l'ultima versione del compilatore intel
 - o "icc -v" verifico che la versione sia la 19.0.5 - openMP v4.5 (vedi qui: <https://www.openmp.org/resources/openmp-compilers-tools/>)
- *Compilatore GNU:*
 - o "echo | cpp -fopenmp -dM | grep -i open" OPENMP release 201107 - OpenMP v3.1 (gcc 4.8.5)
 - o "module purge" carico l'ultima versione del compilatore purge
 - o "module load gnu8" carico l'ultima versione del compilatore gnu8
 - o "echo | cpp -fopenmp -dM | grep -i open" OPENMP release 201511 - OpenMP v4.5 (gcc 8.3)
 - o N.B. (corrispondenza tra le data di rilascio e la specifica OpenMP: <https://www.openmp.org/specifications/>)

Esercizio guida HPC (omp_hello.c + omp_hello.bash)

- Ho copiato il programma omp_hello.c e tramite lo script omp_hello.bash (dove ho modificato la partizione utilizzata in "cpu_guest" □ "#SBATCH --partition=cpu_guest") l'ho compilato ed eseguito ottenendo il seguente output

```
[martina.genovese@ui01 base]$ ./omp_hello
Hello World from thread = 2
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 3
```

Esercizi Base

- Ho copiato gli esercizi di base tramite "cp /hpc/home/roberto.alfieri/SHARE/openMP/base/*."
- Li ho compilati tramite il comando "gcc -fopenmp program.c -o program" e poi eseguiti con "./program"

o omp1_hello.c

In this simple example, the master thread forks a parallel region.

All threads in the team obtain their unique thread number and print it.

The master thread only prints the total number of threads. Two OpenMP library routines are used to obtain the number of threads and each thread's number.

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp1_hello.c -o omp1_hello
[martina.genovese@ui01 base]$ ./omp1_hello
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 1
```

o omp2_race.c

Il programma crea un numero di thread pari al numero di processori disponibili sul sistema (di default). Ogni thread esegue un ciclo di un milione di iterazioni, incrementando la variabile j in modo sicuro attraverso l'uso di una sezione critica. Pertanto, al termine dell'esecuzione del programma, j dovrebbe essere uguale a 1.000.000 moltiplicato per il numero di thread, poiché ogni thread effettua un milione di incrementi.

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp2_race.c -o omp2_race
[martina.genovese@ui01 base]$ ./omp2_race
running 3
running 0
running 2
running 1
ran 1
ran 0
ran 2
ran 3
4000000
```

o omp3_time.cpp

Il programma è un esempio di utilizzazione di OpenMP in C++ per misurare quanto tempo ci vuole per eseguire un'operazione (in questo caso, una pausa di 2 secondi).

```
[martina.genovese@ui01 base]$ g++ -fopenmp omp3_time.cpp -o omp3_time
[martina.genovese@ui01 base]$ ./omp3_time
Start timer
Stop timer
time: 2.00008
```

o omp4_for.c

Il programma parallelizza un ciclo for suddividendo le iterazioni tra diversi thread (usando la direttiva "#pragma omp for e la clausola schedule") ed esegue una sezione di codice in modalità single (usando la direttiva "#pragma omp single"), che garantisce che solo un thread esegua il blocco di codice all'interno.

In sintesi, il programma suddivide un ciclo in diversi thread usando OpenMP, fa eseguire a ogni thread un certo numero di iterazioni del ciclo e poi fa eseguire una sezione di codice a un solo thread.

L'output del programma mostra le esecuzioni parallele dei thread, con messaggi che indicano quale thread esegue quale iterazione. Inoltre, uno dei thread, quello che entra nella sezione single, stampa un messaggio specifico e poi attende 2 secondi prima di completare. Infine, ogni thread stampa un messaggio di completamento.

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp4_for.c -o omp4_for
[martina.genovese@ui01 base]$ ./omp4_for
Esecuzione del thread 0: i=0
Esecuzione del thread 0: i=4
Esecuzione del thread 0: i=8
Esecuzione del thread 0: i=12
Esecuzione del thread 0: i=16
Esecuzione del thread 0: i=20
Esecuzione del thread 3: i=3
Esecuzione del thread 2: i=2
Esecuzione del thread 2: i=6
Esecuzione del thread 2: i=10
Esecuzione del thread 2: i=14
Esecuzione del thread 2: i=18
Esecuzione del thread 2: i=22
Esecuzione del thread 2: i=26
Esecuzione del thread 2: i=30
Esecuzione del thread 2: i=34
Esecuzione del thread 2: i=38
Esecuzione del thread 2: i=42
Esecuzione del thread 2: i=46
Esecuzione del thread 1: i=1
Esecuzione del thread 1: i=5
Esecuzione del thread 1: i=9
Esecuzione del thread 1: i=13
Esecuzione del thread 1: i=17
Esecuzione del thread 1: i=21
Esecuzione del thread 1: i=25
Esecuzione del thread 1: i=29
Esecuzione del thread 1: i=33
Esecuzione del thread 1: i=37
Esecuzione del thread 1: i=41
Esecuzione del thread 1: i=45
Esecuzione del thread 1: i=49
Esecuzione del thread 0: i=24
Esecuzione del thread 0: i=28
Esecuzione del thread 0: i=32
```

```

Esecuzione del thread 0: i=36
Esecuzione del thread 0: i=40
Esecuzione del thread 0: i=44
Esecuzione del thread 0: i=48
Esecuzione del thread 3: i=7
Esecuzione del thread 3: i=11
Esecuzione del thread 3: i=15
Esecuzione del thread 3: i=19
Esecuzione del thread 3: i=23
Esecuzione del thread 3: i=27
Esecuzione del thread 3: i=31
Esecuzione del thread 3: i=35
Esecuzione del thread 3: i=39
Esecuzione del thread 3: i=43
Esecuzione del thread 3: i=47
Esecuzione single del thread 2
2 ha finito
1 ha finito
3 ha finito
0 ha finito

```

○ `omp5_for-schedule_2d.c`

Il programma crea 5 thread che eseguono un ciclo annidato. Ogni thread stampa l'ID del thread e il valore corrente di `j` e `i` durante l'esecuzione dei cicli, infine stampa un messaggio che indica la fine del lavoro.

```

[martina.genovese@ui01 base]$ gcc -fopenmp omp5_for-schedule_2d.c -o omp5_for-schedule_2d
[martina.genovese@ui01 base]$ ./omp5_for-schedule_2d
Esecuzione del thread 0: j=0 i=0
Esecuzione del thread 0: j=0 i=1
Esecuzione del thread 0: j=0 i=2
Esecuzione del thread 0: j=0 i=3
Esecuzione del thread 0: j=0 i=4
Esecuzione del thread 4: j=4 i=0
Esecuzione del thread 4: j=4 i=1
Esecuzione del thread 4: j=4 i=2
Esecuzione del thread 4: j=4 i=3
Esecuzione del thread 4: j=4 i=4
Esecuzione del thread 1: j=1 i=0
Esecuzione del thread 2: j=2 i=0
Esecuzione del thread 2: j=2 i=1
Esecuzione del thread 2: j=2 i=2
Esecuzione del thread 2: j=2 i=3
Esecuzione del thread 2: j=2 i=4
Esecuzione del thread 3: j=3 i=0
Esecuzione del thread 3: j=3 i=1
Esecuzione del thread 3: j=3 i=2
Esecuzione del thread 1: j=1 i=1
Esecuzione del thread 1: j=1 i=2
Esecuzione del thread 1: j=1 i=3
Esecuzione del thread 3: j=3 i=3
Esecuzione del thread 3: j=3 i=4
Esecuzione del thread 1: j=1 i=4
0 ha finito
2 ha finito
3 ha finito
4 ha finito
1 ha finito

```

○ `omp6_single_master.c`

Il programma dimostra l'uso di sezioni `master`, `single`, e `critical` in OpenMP per sincronizzare l'esecuzione di diversi thread e registrare/stampare i tempi relativi.

Osservazioni:

- **Sincronizzazione delle Sezioni:**

La sezione `master` viene eseguita solo dal thread master (`tid=0`), e durante il suo sonno di 1 secondo, gli altri thread continuano a lavorare. Dopo che il master termina la sua parte, ogni thread stampa quanto tempo è passato dall'inizio.

- **Sezione `single`:**

Un solo thread selezionato esegue la sezione `single`, dormendo per 1 secondo. Durante questo tempo, gli altri thread aspettano che la sezione `single` sia completata.

- **Sezioni Critiche:**

Le sezioni `critical` servono a garantire che l'output dei thread non si sovrapponga, quindi ogni thread stampa il proprio messaggio in modo ordinato.

```
[martina.genovese@ui01 base]$ ./omp6_single_master
THR-0 MASTER entra sleep 1
3 dopo master 0.000080
2 dopo master 0.000080
1 dopo master 0.000080
THR-3 SINGLE entra sleep 1
THR-0 MASTER esce
0 dopo master 1.000191
THR-3 SINGLE esce
0 dopo single 1.000249
3 dopo single 1.000260
2 dopo single 1.000265
1 dopo single 1.000271
```

o `omp7_reduction.c`

Il programma parallelizza un algoritmo di Monte Carlo per la stima del pi-greco, sfruttando la riduzione per aggregare i risultati parziali dei thread.

L'output indica che su 100 punti generati 84 sono caduti all'interno del cerchio, portando a una stima di pi-greco di circa $3.36000e+00$ con l'utilizzo di 4 thread (NT: 4).

Metodo Monte Carlo: algoritmo per stimare il valore di pi-greco. Genera punti casuali all'interno di un quadrato di lato 2 e conta quanti di questi punti cadono all'interno di un cerchio di raggio 1. La frazione di punti all'interno del cerchio rispetto al totale dei punti generati è proporzionale a $\pi/4$.

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp7_reduction.c -o omp7_reduction
[martina.genovese@ui01 base]$ ./omp7_reduction
INSIDE: 84 PI: 3.36000e+00 NT: 4
```

o `omp8_sections.c`

Nel programma vengono utilizzate le direttive:

- `"#pragma omp sections"` che permette di suddividere il lavoro in blocchi indipendenti (section), che possono essere eseguiti in parallelo da thread diversi disponibili
- `"#pragma omp parallel private(i)"` che crea un'area parallela in cui i thread eseguono il codice all'interno. (Nota: ogni thread ha la propria copia privata della variabile `i` grazie alla clausola `"private(i)"`)

Il programma crea tre thread ognuno dei quali esegue una delle sezioni, di cui le prime due contengono un ciclo for e la terza contiene un ciclo for e una pausa di 4 secondi.

L'output mostra i messaggi stampati dai diversi thread che eseguono una delle sezioni parallele, quindi l'id del thread che è in esecuzione, l'iterazione del for che sta eseguendo e il blocco, cioè quale delle tre sezioni, sta eseguendo.

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp8_sections.c -o omp8_sections
[martina.genovese@ui01 base]$ ./omp8_sections
Th 0: 0 del blocco 1
Th 0: 1 del blocco 1
Th 0: 2 del blocco 1
Th 0: 3 del blocco 1
Th 0: 4 del blocco 1
Th 1: 0 del blocco 3
Th 1: 1 del blocco 3
Th 1: 2 del blocco 3
Th 1: 3 del blocco 3
Th 1: 4 del blocco 3
Th 2: 0 del blocco 2
Th 2: 1 del blocco 2
Th 2: 2 del blocco 2
Th 2: 3 del blocco 2
Th 2: 4 del blocco 2
1 ha finito
0 ha finito
2 ha finito
```

o `omp_balancing.c`

Nel programma vengono utilizzate le direttive:

- `"#pragma omp parallel for"` che per parallelizza i cicli for in modo che il carico di lavoro possa essere distribuito tra più thread
 - o `schedule(dynamic, 1)`: le iterazioni del ciclo vengono assegnate ai thread in modo dinamico, ogni volta che un thread termina l'esecuzione di un'iterazione, ne richiede immediatamente un'altra
 - o `schedule(static, 1)`: le iterazioni del ciclo vengono suddivise in blocchi fissi di dimensione 1, quindi ogni thread esegue una singola iterazione in sequenza prima di passare alla successiva

- `schedule(dynamic, 100)`: i thread ricevono blocchi di 100 iterazioni alla volta in modo dinamico, ogni volta che un thread termina tutte le iterazioni che gli sono state assegnate, può richiederne un altro blocco di 100

- `"#omp_get_wtime"` che permette di misurare il tempo d'esecuzione di sezioni di codice

Il programma utilizza tre thread per parallelizzare l'esecuzione di un ciclo for che genera numeri casuali, fa una pausa ed incrementa la somma dei numeri casuali generati. Nel frattempo viene misurato il tempo totale di esecuzione.

L'output mostra un elenco con l'iterazione del for in esecuzione (i), l'id del thread che sta eseguendo quella sezione (r) e il numero casuale generato (x); infine stampa la somma finale dei valori casuali generati (sum) e il tempo totale impiegato (time).

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp_balancing.c -o omp_balancing
[martina.genovese@ui01 base]$ ./omp_balancing
# n=20
i:0 r:3 x:0.8370
i:5 r:3 x:0.8573
i:15 r:3 x:0.9880
i:10 r:3 x:0.9880
i:1 r:0 x:0.3701
i:11 r:2 x:0.3014
i:16 r:3 x:0.7051
i:6 r:1 x:0.0378
i:2 r:0 x:0.8034
i:12 r:2 x:0.8427
i:17 r:3 x:0.6489
i:7 r:1 x:0.1037
i:18 r:3 x:0.2752
i:8 r:1 x:0.2201
i:9 r:1 x:0.0486
i:19 r:3 x:0.6515
i:3 r:0 x:0.5986
i:13 r:2 x:0.2989
i:4 r:0 x:0.4181
i:14 r:2 x:0.4352
sum:8.5301 time: 1.4337
```

○ `omp_master-slave.c`

Il programma crea un'architettura di thread master-slave (il master esegue alcune operazioni e gli slave eseguono altre operazioni in parallelo). In questo caso il thread "master" stampa un messaggio e poi genera una regione parallela "slave" che esegue un ciclo for in parallelo, creando 8 thread aggiuntivi per gestire il lavoro.

In altre parole, il programma crea il parallelismo annidato che consente di sfruttare ulteriormente le risorse di calcolo disponibili creando regioni parallele all'interno di altre regioni parallele.

Viene creata una prima regione parallela con 2 thread, uno di questi thread esegue la sezione "master", mentre l'altro esegue la sezione "slave". Nella sezione "slave", viene creata una seconda regione parallela annidata con 8 thread che eseguono il ciclo for in parallelo.

L'output mostrerà i messaggi stampati dai diversi thread, con ogni thread che esegue una delle sezioni parallele.

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp_master-slave.c -o omp_master-slave
[martina.genovese@ui01 base]$ ./omp_master-slave
Master 0/2
Thr 1/8 : 1
Thr 3/8 : 3
Thr 4/8 : 4
Thr 6/8 : 6
Thr 0/8 : 0
Thr 5/8 : 5
Thr 7/8 : 7
Thr 2/8 : 2
```

○ `omp_master-slave-lock.c`

Il programma implementa un modello di comunicazione master-slave usando lock per garantire la corretta sincronizzazione tra thread ed evitare delle race-condition. I thread slave richiedono lavoro al master, eseguono il lavoro assegnato e poi richiedono altro lavoro, il tutto in un ciclo continuo.

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp_master-slave-lock.c -o omp_master-slave-lock
[martina.genovese@ui01 base]$ ./omp_master-slave-lock
Master: received request from 1 response: block 1
Slave 1: working on block 1
Master: received request from 3 response: block 2
Slave 3: working on block 2
Master: received request from 2 response: block 3
Slave 2: working on block 3
```

```
[martina.genovese@ui01 base]$ gcc -fopenmp omp_master-slave-lock.c -o omp_master-slave-lock
[martina.genovese@ui01 base]$ ./omp_master-slave-lock
Master: received request from 1 response: block 1
Slave 1: working on block 1
Master: received request from 3 response: block 2
Slave 3: working on block 2
Master: received request from 2 response: block 3
Slave 2: working on block 3
Master: received request from 0 response: block 4
Slave 0: working on block 4
Master: received request from 1 response: block 5
Slave 1: working on block 5
Master: received request from 3 response: block 6
Slave 3: working on block 6
Master: received request from 0 response: block 7
Slave 0: working on block 7
Master: received request from 2 response: block 8
Slave 2: working on block 8
Master: received request from 1 response: block 9
Slave 1: working on block 9
Master: received request from 2 response: block 10
```

Esercizio omp_cpi.c

ESERCIZIO omp_cpi.c

Copiare il sorgente cpi.c e parallelizzarlo con openMP:

```
mkdir ~/HPC2324/openMP/cpi/
```

```
cp /hpc/home/roberto.alfieri/SHARE/perf/CPU/cpi/cpi.c omp_cpi.c
```

Aggiungere l'header file: `#include <omp.h>`

Tempi: si usa la funzione `omp_get_wtime()` Esempio: `double ta,tb; ta=omp_get_wtime(); .. tb=omp_get_wtime(); printf("%.5f", tb-ta);`

Numero di thread. Esempio:

`nt=omp_get_max_threads()` // ritorna il numero di thread che verranno attivati nella regione parallela

nt deve essere stampato nel file csv, assieme a n (dimensione del problema), il tempo complessivo, tempi t1 e t2, gli errori e1 e e2 e il nome dell'Host di esecuzione:

```
fprintf(stdout,"OMP, %d, %d, %.4f, %.4f, %.4f, %.8e, %.8e, %s\n",
        nt, n, td-ta, tc-tb, td-tc, fabs(pi1-PI), fabs(pi2-PI), hostname)
```

Parallelizzazione dei cicli for di `f1()` e `f2()`. Aggiungere per entrambi la direttiva:

```
#pragma omp parallel for private(x) reduction(+:sum)
for (i = 1; i <= n; i++)
```

Compilare ed eseguire:

```
gcc -fopenmp omp_cpi.c -o omp_cpi -lm
```

```
omp_cpi
```

```
OMP_NUM_THREADS=8 omp_cpi
```

Eseguire lo Strong Scaling e generare il plot.

Obiettivo: parallelizzare il programma

Prima parte:

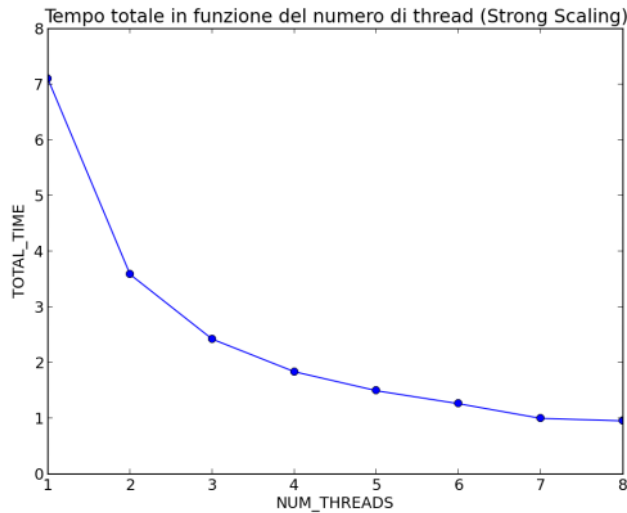
- Ho copiato il programma `cp /hpc/home/roberto.alfieri/SHARE/perf/CPU/cpi/cpi.c omp_cpi.c`
- Ho analizzato il programma:
 - o Il programma calcola approssimazioni di π -greco usando due metodi diversi:
 - F1: integra la funzione $\sqrt{1-x^2}$
 - F2: integra la funzione $1/(1+x^2)$
 - o Inoltre misura il tempo di esecuzione dell'inizializzazione e dell'esecuzione dei due metodi
 - o Infine mostra i risultati dei calcoli e dei rispettivi tempi di esecuzione
- Ho modificato il programma per parallelizzarlo con openMP, in particolare ho:
 - o parallelizzato i cicli for di `f1()` e `f2()` aggiungendo per entrambi la direttiva `"#pragma omp parallel for private(x) reduction(+:sum)"`
 - o Utilizzato la funzione `"omp_get_wtime()"` per misurare i tempi di esecuzione
 - o Aggiunto alla stampa dei risultati finali `nt` (calcolato precedentemente con la funzione `"omp_get_max_threads()"`), cioè il numero di thread che verranno attivati nella regione parallela

Seconda parte:

- Ho creato lo script `omp_cpi.slurm` per compilare ed eseguire il programma utilizzando da 1 a 8 threads ed ottenere il file di output `results.csv` che mostra l'andamento del tempo di esecuzione del programma all'aumentare del numero di thread
- Strong Scaling: capacità di un algoritmo parallelo di accelerare l'esecuzione di un problema di dimensione fissa riducendo il tempo di calcolo man mano che si aumenta il numero di processori o core utilizzati

Terza parte:

- Ho creato il programma in python `omp_cpi.py` per generare un grafico dai dati ottenuti



CODICE

`omp_cpi.c`

```
33 int main( int argc, char *argv[])
34 {
35     //time 1 (start)
36     ta = omp_get_wtime();
37
38     //inizializzazione
39     double PI = 3.14159265358979323846264338327950288 ;
40     gethostname(hostname, 100);
41     options(argc, argv); /* optarg management */
42     h = 1.0 / (double) n;
43     sleep (s); // Simulazione codice non parallellizzabile
44     nt = omp_get_max_threads();
45
46     //time 2
47     tb = omp_get_wtime();
48
49     sum1 = f1(n);
50
51     //time 3
52     tc = omp_get_wtime();
53
54     sum2 = f2(n);
55
56     //time 4 (end)
57     td = omp_get_wtime();
58
59     pi1 = 4 * h * sum1;
60     pi2 = 4 * h * sum2;
61
62     // tempo inizializzazione
63     tnp = tb - ta;
64
65     // tempo esecuzione f1
66     tp1 = tc - tb;
67
68     // tempo esecuzione f2
69     tp2 = td - tc;
70
71     fprintf(stderr, "# OMP, thread_numbers, intervals, total_time, f1_time, f2_time, f1_err, f2_err, hostname");
72     fprintf(stdout, "OMP, %d, %d, %.4f, %.4f, %.4f, %.8e, %.8e, %s\n",
73             nt, n, td-ta, tc-tb, td-tc, fabs(pi1-PI), fabs(pi2-PI), hostname);
74
75     return 0;
76 }
```

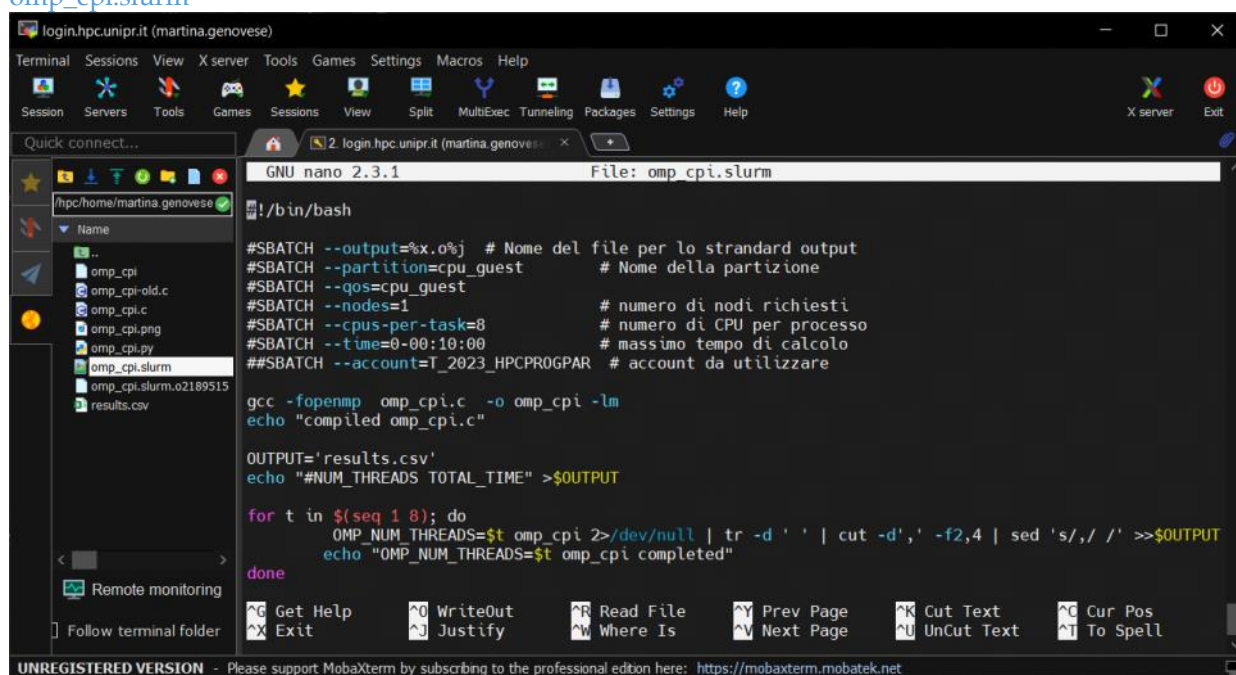


```

78 double f1 (long int n)
79 {
80     long int i;
81     double x, sum=0.0;
82
83     #pragma omp parallel for private(x) reduction(+:sum)
84     for (i = 1; i <= n; i++)
85     {
86         x = h * ((double)i - 0.5);
87         sum += sqrt(1-x*x);
88     }
89     return sum;
90 }
91
92 double f2 (long int n)
93 {
94     long int i;
95     double x, sum=0.0;
96
97     #pragma omp parallel for private(x) reduction(+:sum)
98     for (i = 1; i <= n; i++)
99     {
100         x = h * ((double)i - 0.5);
101         sum += (1.0 / (1.0 + x*x));
102     }
103     return sum;
104 }

```

omp_cpi.slurm



```

GNU nano 2.3.1 File: omp_cpi.slurm
#!/bin/bash

#SBATCH --output=%x.o%j # Nome del file per lo standard output
#SBATCH --partition=cpu_guest # Nome della partizione
#SBATCH --qos=cpu_guest
#SBATCH --nodes=1 # numero di nodi richiesti
#SBATCH --cpus-per-task=8 # numero di CPU per processo
#SBATCH --time=0-00:10:00 # massimo tempo di calcolo
##SBATCH --account=T_2023_HPCPROGPAR # account da utilizzare

gcc -fopenmp omp_cpi.c -o omp_cpi -lm
echo "compiled omp_cpi.c"

OUTPUT='results.csv'
echo "#NUM_THREADS TOTAL_TIME" >$OUTPUT

for t in $(seq 1 8); do
    OMP_NUM_THREADS=$t omp_cpi 2>/dev/null | tr -d ' ' | cut -d',' -f2,4 | sed 's/,/ /' >>$OUTPUT
    echo "OMP_NUM_THREADS=$t omp_cpi completed"
done

```

omp_cpi.py

The screenshot shows a MobaXterm terminal window titled 'login.hpc.unipr.it (martina.genovese)'. The terminal is running GNU nano 2.3.1, editing a file named 'omp_cpi.py'. The script is a Python program that reads a CSV file 'results.csv', extracts 'NUM_THREADS' and 'TOTAL_TIME' data, and plots 'TOTAL_TIME' against 'NUM_THREADS' using Matplotlib. The plot is titled 'Tempo totale in funzione del numero di thread (Strong Scaling)'. The script also saves the figure as 'omp_cpi.png'.

```
#!/usr/bin/env python2

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

data = pd.read_csv('results.csv', delim_whitespace=True, header=0, names=['NUM_THREADS', 'TOTAL_TIME'])

num_threads = data['NUM_THREADS']
total_time = data['TOTAL_TIME']

plt.plot(num_threads, total_time, marker='o', linestyle='-')

plt.title('Tempo totale in funzione del numero di thread (Strong Scaling)')
plt.xlabel('NUM_THREADS')
plt.ylabel('TOTAL_TIME')

plt.savefig('omp_cpi.png', bbox_inches='tight', dpi=150)
plt.close()
```

The terminal window includes a sidebar with a file explorer showing the directory structure of the user's home folder, including files like 'omp_cpi', 'omp_cpi-old.c', 'omp_cpi.c', 'omp_cpi.png', 'omp_cpi.py', 'omp_cpi.slurm', 'omp_cpi.slurm.02189515', and 'results.csv'. The bottom status bar indicates 'UNREGISTERED VERSION' and provides a link to the professional edition.