Corso di High Performance Computing

Esercitazione MPI del 27/10/2020

Moreno Marzolla

Ultimo aggiornamento: 2020-10-24

Per svolgere l'esercitazione è possibile collegarsi al server `isi-raptor03.csr.unibo.it` tramite ssh, usando come *username* il proprio indirizzo mail istituzionale completo, e come password la propria password istituzionale (cioè quella usata per accedere alla casella di posta o ad AlmaEsami). Sulla macchina è installato il compilatore `gcc` e alcuni editor di testo per console: `vim`, `pico`, `joe`, `ne` e `emacs`. Per chi non è pratico suggerisco `pico`, che è semplice da usare e richiede poche risorse. Chi ha un portatile con Linux può lavorare localmente, dopo aver installato il compilatore.

Per scaricare l'archivio con i sorgenti di questa esercitazione è possibile usare i comandi:

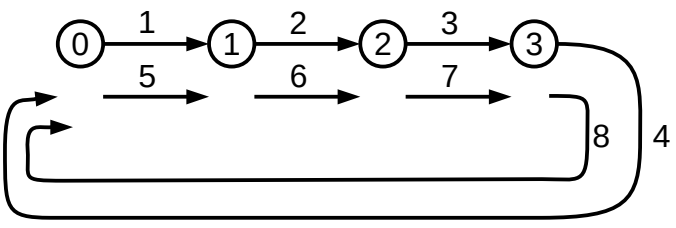
```
wget https://www.moreno.marzolla.name/teaching/HPC/ex1-mpi.zip
unzip ex1-mpi.zip
cd ex1-mpi/
```

1. Comunicazione ad anello

Realizzare un programma MPI chiamato `mpi-ring.c` che effettua una comunicazione ad anello tra i processi. Più in dettaglio, detto P il numero di processi MPI (da specificare con il comando `mpirun`; si deve avere $P > 1$) si richiede di realizzare le funzionalità seguenti:

- Il programma riceve sulla riga di comando un valore intero K, che rappresenta il numero di "giri" dell'anello che devono essere effettuati ($K \geq 1$). Ricordo che tutti i processi MPI hanno accesso ai valori passati sulla riga di comando, quindi tutti possono conoscere il valore di K senza bisogno di comunicazioni esplicite.
- Il processo 0 (il master) invia al processo 1 un intero, il cui valore iniziale è 1.
- Ciascun processo p (incluso il master) rimane in attesa di ricevere un valore v dal processo $p - 1$; una volta ricevuto, il processo p invia il valore $(v + 1)$ al processo $p + 1$. Poiché la comunicazione si considera ad anello, il predecessore del processo 0 è $(P - 1)$, mentre il successore del processo $(P - 1)$ è il processo 0.
- Il master stampa il valore ricevuto dopo la K-esima iterazione e la computazione termina; dati il numero P di processi e il valore di K, quale valore deve stampare il master?

Ad esempio, se $K = 2$ e ci sono $P = 4$ processi, la comunicazione da realizzare deve essere la seguente (i cerchi sono i processi MPI; le frecce rappresentano messaggi il cui contenuto è il numero indicato sopra). I messaggi percorrono $K = 2$ "giri" dell'anello composto da tutti i processi; al termine dell'esecuzione il processo 0 riceve e stampa il valore 8.



2. Broadcast tramite comunicazioni punto-punto

Lo scopo di questo esercizio è di implementare la funzione

```
void my_Bcast(int *v)
```

La funzione deve svolgere operazioni diverse a seconda che venga eseguita dal master (processo 0) o da un altro processo. Per fare ciò ogni processo determina il proprio rango p e il numero P di processi MPI attivi.

Il processo 0:

- invia il valore `*v` ai processi $(2p + 1)$ e $(2p + 2)$ (purché i destinatari esistano)

Ogni altro processo $p > 0$:

- riceve un valore dal processo $(p – 1)/2$ e lo memorizza nella locazione di memoria `v`;
- invia il valore ricevuto ai processi $(2p + 1)$ e $(2p + 2)$ (purché i destinatari esistano).

Ad esempio, nel caso $P = 14$ si otterrebbe lo schema di comunicazione illustrato nella Figura 1; le frecce indicano comunicazioni punto-punto; i numeri indicano il rango dei processi coinvolti. Si noti che il procedimento descritto in precedenza funziona correttamente qualunque sia il numero P di processi.

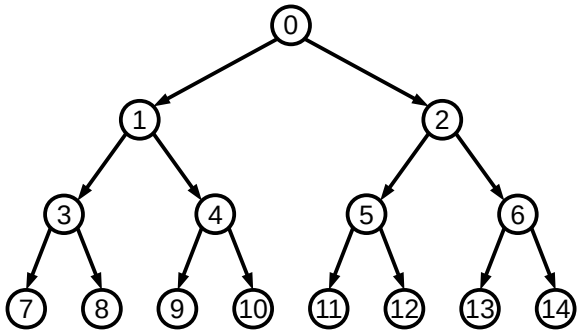


Figura 1: Schema d comunicazione con P = 14 processi

Il file `mpi-my-bcast.c` contiene lo scheletro della funzione `my_Bcast()` cui manca il corpo. Completare il corpo della funzione `my_Bcast()` usando `MPI_Send()` e `MPI_Recv()`.

Abbiamo visto che MPI mette a disposizione la funzione `MPI_Bcast()` per realizzare una comunicazione di tipo *broadcast*, che va sempre preferita a soluzioni “fatte in casa” come quella descritta sopra. Lo scopo di questo esercizio è di vedere una possibile implementazione della funzione di broadcast MPI.

3. Calcolo di Pi greco

Il file `mpi-pi.c` contiene una implementazione seriale di un algoritmo di tipo Monte Carlo per il calcolo del valore approssimato di π (pi greco). L'algoritmo è già stato descritto nella prima esercitazione OpenMP, e viene richiamato nuovamente per comodità (si faccia riferimento alla Figura 2).

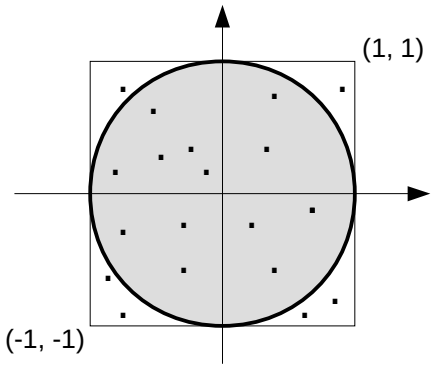


Figura 2: Calcolo del valore di π con metodo Monte Carlo

- Vengono generati *N* punti casuali all'interno del quadrato di vertici opposti (-1, -1) e (1, 1);
- Si definisce *x* il numero di punti che cadono all'interno del cerchio di raggio unitario e centro nell'origine degli assi;
- Il rapporto *x* / *N* approssima il rapporto tra l'area del quadrato (che vale 4) e l'area del cerchio inscritto in esso. Poiché sappiamo che l'area di tale cerchio è π, possiamo stimare π come (4*x* / *N*)

Modificare il file `mpi-pi.c` per parallelizzare il calcolo del valore di π. Sono possibili diverse strategie; si consiglia di usare quella seguente che ha il vantaggio di essere abbastanza semplice da realizzare (*P* rappresenta il numero di processi MPI attivi):

1. Ciascun processo ottiene il valore di *N* dalla riga di comando; si può inizialmente assumere che *N* sia multiplo di *P*, e successivamente rilassare questo requisito per fare funzionare il programma con qualsiasi valore di *N*.
2. Ciascun processo *p*, incluso il master, genera *N*/*P* punti casuali e tiene traccia del numero x_p di punti all'interno del cerchio;
3. Ciascun processo *p* diverso dal master invia al master il proprio valore x_p utilizzando operazioni send/receive punto-punto.
4. Il master riceve i valori x_p (per ogni $p = 1, \ldots P - 1$; il valore x_0 già ce l'ha) e calcola la loro somma *x* stampando il valore approssimato di π come (4*x* / *N*).

Realizzare il passo 3 mediante operazioni send/receive punto-punto. Questo approccio non è efficiente: vedremo nella prossima lezione come i passi 3-4 possano (debbano!) essere realizzati mediante la funzione MPI che realizza una riduzione.

4. Insieme di Mandelbrot

Il file `mpi-mandelbrot.c` contiene lo scheletro di una implementazione MPI dell'algoritmo che calcola l'insieme di Mandelbrot; non si tratta di una versione realmente parallela, in quanto il processo master è l'unico che esegue computazioni. Il programma accetta come parametro opzionale la dimensione verticale dell'immagine, ossia il numero di righe (default 1024). La risoluzione orizzontale viene calcolata automaticamente dal programma in modo da includere l'intero insieme. Il programma produce un file `mandebrot.ppm` contenente una immagine in formato PPM (*Portable Pixmap*) dell'insieme di Mandelbrot.

Scopo di questo esercizio è la realizzazione di una versione realmente parallela del programma, in cui tutti i processi MPI cooperano al calcolo dell'immagine. In particolare, si richiede di partizionare l'immagine a blocchi per righe, dove il numero di blocchi è uguale al numero di

processi MPI in modo che ogni processo calcoli una porzione dell'immagine come schematizzato nella Figura 3.

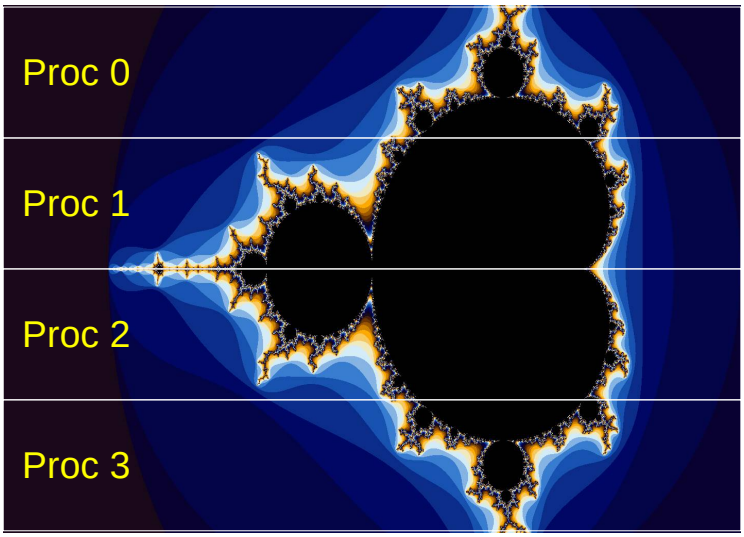


Figura 3: Esempio di suddivisione del dominio per il calcolo dell'insieme di Mandelbrot con 4 processi MPI

Più in dettaglio, ciascun processo alloca e calcola una porzione di immagine di dimensione *xsize* × (*ysize* / *P*), dove *P* è il numero di processi MPI utilizzati; per questa fase non serve alcuna comunicazione. Successivamente, il master assembla le porzioni di immagine calcolate dai vari processi mediante la funzione `MPI_Gather()`. Ciascuna porzione di immagine è un array di (*xsize* × *ysize* / *P* × 3) elementi di tipo `MPI_BYTE` (ogni pixel è composto da 3 byte). Si assuma che *ysize* sia un multiplo di *P*; una volta ottenuto un programma corretto, modificarlo per funzionare con una dimensione verticale arbitraria delegando al processo 0 il calcolo della porzione di immagine corrispondente alle ultime (`ysize % P`) righe. Vedremo più avanti come si possa usare la funzione `MPI_Gatherv()` per assemblare porzioni di dimensione non uniforme.

Suggerisco di tenere da parte la versione seriale fornita da usare come riferimento. Per verificare in modo empirico la correttezza della versione parallela realizzata, consiglio di confrontare le immagini prodotte dalla versione parallela e da quella seriale fornita (devono risultare identiche byte per byte). Per confrontare le immagini si può usare il comando `cmp` dalla shell di Linux: il comando

```
cmp file1 file2
```

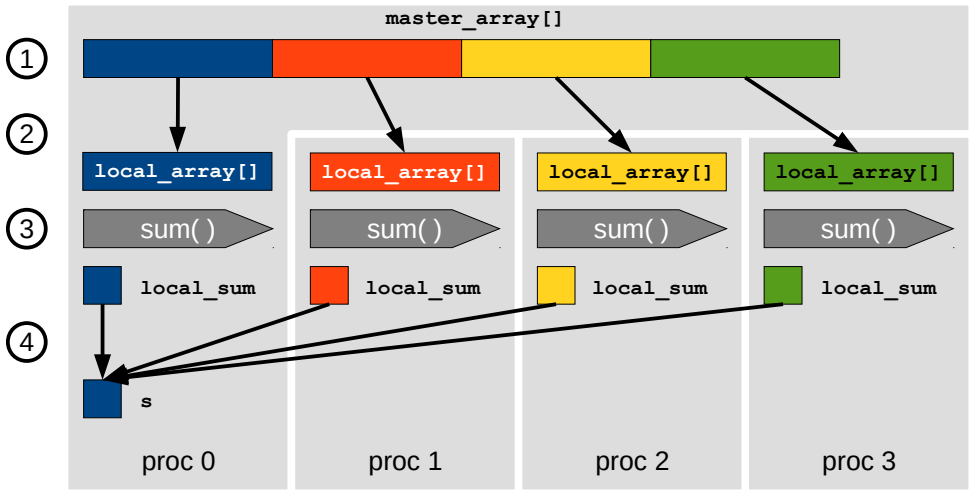
stampa un messaggio se e solo se `file1` e `file2` differiscono.

5. Somma degli elementi di un array

Il file `mpi-sum.c` contiene una implementazione essenzialmente seriale di un programma MPI che calcola la somma degli elementi di un array; nella versione fornita, il processo 0 esegue tutte le computazioni. Modificare il programma in modo da parallelizzare la somma, seguendo le seguenti indicazioni (si faccia riferimento alla figura)

1. Il processo master (quello con *rank* 0) crea e inizializza l'array `master_array[]` di cui calcolare la somma.
2. Il master distribuisce l'array `master_array[]` tra i *P* processi MPI (incluso se stesso) usando `MPI_Scatter()`.

3. Ciascun processo calcola la somma parziale della propria porzione di array; il master opera sul primo blocco dell'array globale.
4. Ciascun processo diverso dal master invia la propria somma locale al master, usando `MPI_Send()`; il master riceve le somme locali usando `MPI_Recv()` e le accumula nella somma globale `s`.



Vedremo nelle prossime lezioni come il passo 4) possa essere effettuato in modo più efficiente sfruttando le operazioni collettive.