Corso di High Performance Computing

Esercitazione MPI del 13/11/2018

Moreno Marzolla

Ultimo aggiornamento: 2018-11-12

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 http://www.moreno.marzolla.name/teaching/HPC/ex2-mpi.zip
unzip ex2-mpi.zip
cd ex2-mpi/
```

Alcuni degli esercizi producono immagini in formato PPM (*Portable Pixmap*) che le macchine Windows dei laboratori non sono in grado di visualizzare. È necessario convertire tali immagini in un formato diverso (ad esempio, PNG) dando sul server il comando:

```
convert image.ppm image.png
```

per poi copiare il file `image.png` sul proprio PC usando il programma Winscp (già installato).

1. Prodotto scalare

Il file `mpi-dot.c` calcola il prodotto scalare tra due array `a[]` e `b[]` di uguale lunghezza *n*. Ricordo che il prodotto scalare *s* di due array `a[]` e `b[]` di lunghezza *n* è:

$$s=\sum_{i=0}^{n-1} a[i]\times b[i]$$

Il programma implementa una soluzione seriale in quanto solo il master esegue la computazione. Scopo di questo esercizio è di parallelizzare il calcolo del prodotto scalare, distribuendo gli array `a[]` e `b[]` tra i processi usando la funzione `MPI_Scatter`. Ciascun processo calcola il prodotto scalare della porzione di array ricevuti; il master usa la funzione `MPI_Reduce` per sommare i prodotti scalari parziali, determinando il valore di *s*.

Assumere inizialmente che *n* sia un multiplo esatto del numero di processi MPI. Realizzare successivamente una versione del programma che funzioni correttamente con lunghezze *n* arbitrarie. La soluzione più semplice consiste nel far gestire al processo master i dati in eccesso. Una soluzione alternativa è di usare `MPI_Scatterv` per distribuire l'input ai vari processi.

2. Calcolo del bounding box di un insieme di rettangoli

Scopo di questo esercizio è il calcolo del *bounding box* di un insieme di rettangoli. Il bounding box è il rettangolo di area minima che contiene tutti i rettangoli dati; un esempio è mostrato nella Figura 1 (il bounding box è quello tratteggiato)

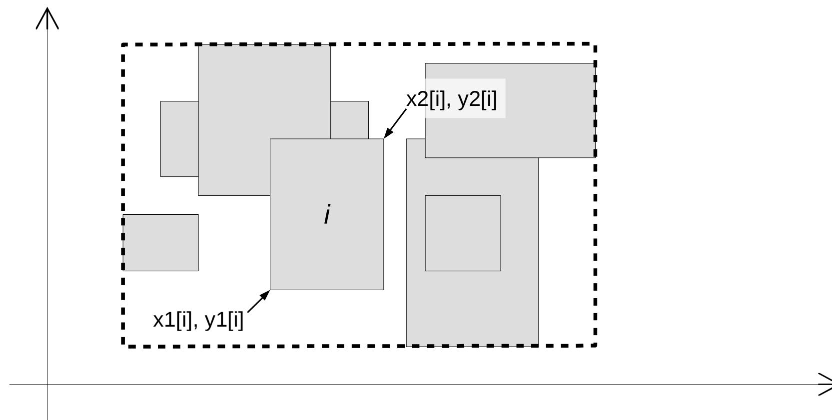

Figura 1: Bounding box di un insieme di rettangoli

Le coordinate dei rettangoli sono indicate in un file di testo, con il formato seguente. La prima riga contiene il numero *N* di rettangoli; seguono *N* righe, ciascuna composta da quattro valori `x1[i]` `y1[i]` `x2[i]` `y2[i]` di tipo `float`, separati da spazi. Le righe rappresentano le coordinate degli angoli opposti di ciascun rettangolo: (`x1[i]`, `y1[i]`) sono le coordinate dell'angolo in basso a sinistra dell'*i*-esimo rettangolo, mentre (`x2[i]`, `y2[i]`) sono quelle dell'angolo in alto a destra. Il riferimento sono gli assi cartesiani.

Viene fornito un programma `mpi-bbox.c` che risolve il problema in modo sequenziale, dato che solo il processo master effettua le computazioni. Scopo di questo esercizio è di parallelizzare il programma in modo che *P* processi MPI cooperino per il calcolo del bounding box. Si suggerisce di strutturare il programma in base ai passi seguenti:

1. Il master legge i dati dal file di input, inserendo le coordinate negli array `x1[]`, `y1[]`, `x2[]`, `y2[]`; si può inizialmente assumere che il numero di rettangoli *N* sia un multiplo del numero *P* di processi MPI.
2. Il master comunica il valore *N* ai processi (usando `MPI_Bcast`), e distribuisce le coordinate dei rettangoli tra i processi MPI usando `MPI_Scatter`; in questo modo ogni processo riceve i dati di *N*/*P* rettangoli (assumere inizialmente che *N* sia multiplo di *P*).
3. Ciascun processo calcola il bounding box dei rettangoli a lui assegnati.
4. Il master usa `MPI_Reduce` per calcolare i minimi/massimi delle coordinate dei bounding box parziali, usando gli operatori di riduzione `MPI_MIN` e `MPI_MAX`. Al termine di questa fase il master avrà determinato il bounding box di tutti i rettangoli.

Nell'archivio dell'esercitazione è fornito un programma `bbox-gen.c` che può essere usato per generare dei file di input composti da rettangoli generati casualmente; le istruzioni d'uso sono contenute nel sorgente.

Dopo aver risolto il problema assumendo che *N* sia multiplo di *P*, modificare il codice per funzionare correttamente per un numero *N* arbitrario di rettangoli.

3. 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, in modo che ogni processo calcoli una porzione dell'immagine, come schematizzato nella Figura 2.

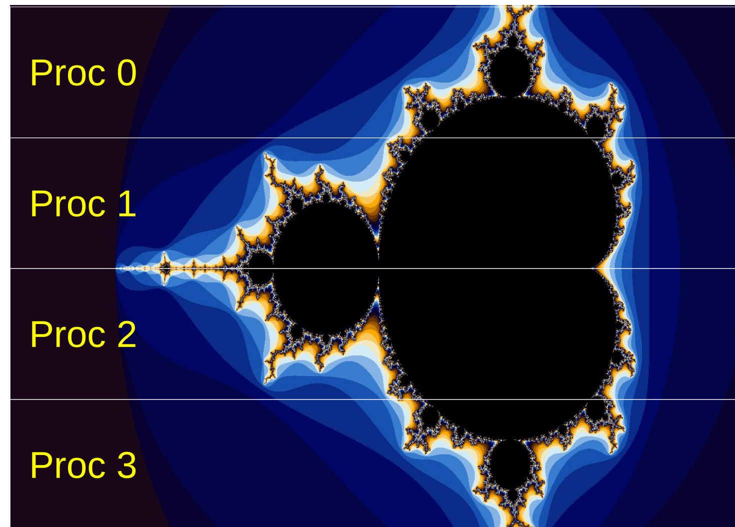

Figura 2: Esempio di suddivisione del dominio per il calcolo dell'insieme di Mandelbrot con 4 processi MPI

Ciascun processo alloca e calcola una porzione di immagine di dimensione $xsize \times (ysize / P)$, 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 $(xsize \times ysize \ / \ P \times 3)$ elementi di tipo `MPI_BYTE` (ogni pixel è composto da 3 byte). Si assuma inizialmente che *ysize* sia un multiplo di *P*; una volta ottenuto un programma corretto, modificarlo per funzionare con una dimensione verticale arbitraria, ad esempio delegando al processo 0 (il master) il calcolo della porzione di immagine corrispondente alle ultime (`ysize % P`) righe. In alternativa si può usare la funzione `MPI_Gatherv()` per assemblare porzioni di altezza non uniforme.

Suggerisco di tenere da parte la versione seriale fornita come punto di partenza; essa potrà tornare utile per verificare la correttezza della versione parallela realizzata confrontando le immagini prodotte (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.

4. Esercizi extra (se avanza tempo)

Svolgere l'esercizio `mpi-sum.c` della precedente esercitazione MPI usando le funzioni per comunicazione collettiva MPI viste fino ad oggi. Gestire correttamente il caso in cui la dimensione *n* dell'array non sia multipla del numero di processi MPI.

Svolgere l'esercizio `mpi-pi.c` della precedente esercitazione MPI usando la funzione `MPI_Reduce()` per effettuare la riduzione.