Corso di High Performance Computing

Esercitazione MPI del 30/10/2019

Moreno Marzolla

Ultimo aggiornamento: 2019-11-01

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)

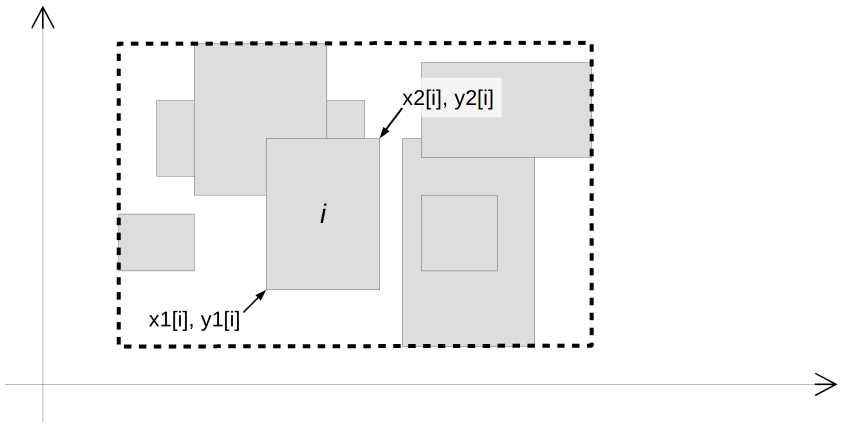


Figura 1: Bounding box di un insieme di rettangoli

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.

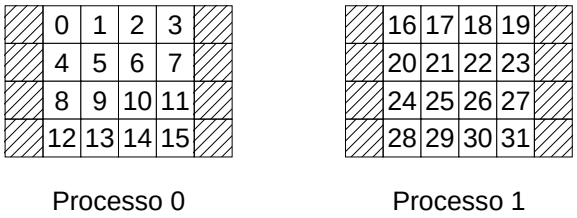
3. MPI Datatype

Quando si deve decomporre un dominio rettangolare di dimensione $R \times C$ tra un insieme di processi MPI si è soliti adottare una decomposizione a blocchi per righe (Block, *): il primo processo elabora le prime R/P righe; il secondo le successive R/P righe, e così via. Infatti, nel linguaggio C le matrici sono memorizzate per righe, e una decomposizione di tipo (Block, *)

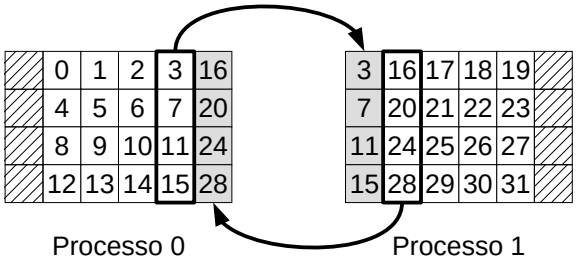
consente di avere le partizioni adiacenti in memoria semplificando l'invio dei dati tramite le funzioni MPI apposite. In questo esercizio consideriamo una decomposizione sulle colonne allo scopo di prendere familiarità con gli `MPI_Datatype`.

Scopo dell'esercizio è la realizzazione di un programma MPI in cui i processi 0 e 1 mantengono ciascuno una matrice di interi di dimensioni $SIZE \times SIZE$; le matrici includono una colonna a sinistra e a destra di *ghost cells* (dette anche *halo*), quindi la loro dimensione effettiva è $SIZE \times (SIZE + 2)$. Nello scheletro di programma `mpi-send-col.c` si pone $SIZE = 4$, ma il programma deve funzionare con qualunque valore.

I processi inizializzano le proprie matrici come segue (caso $SIZE = 4$)



Il processo 0 deve inviare l'ultima colonna della propria matrice al processo 1, che la inserisce nell'*halo* di sinistra; analogamente, il processo 1 deve inviare la prima colonna della propria matrice al processo 0, che la inserisce nell'*halo* di destra.



Per completare lo scambio delle colonne sul bordo il processo 0 invia ora la prima colonna della propria matrice al processo 1, che la inserisce nell'*halo* di destra; il processo 1 invia l'ultima colonna (quella di destra) della propria matrice al processo 0, che la inserisce nell'*halo* di destra.

4. Automa cellulare della "regola 30"

Quando abbiamo parlato dei pattern per la programmazione parallela abbiamo introdotto gli automi cellulari (CA) come semplice esempio di computazioni di tipo *stencil*. In questo esercizio consideriamo l'automa cellulare prodotto dalla "regola 30". L'automa è costituito da un array $x[N]$ di N interi, ciascuno dei quali può avere valore 0 oppure 1. Lo stato dell'automa evolve durante istanti discreti nel tempo: lo stato di una cella al tempo t dipende dal suo stato e da quello dei due vicini al tempo $t - 1$. Assumiamo un dominio ciclico, per cui i vicini della cella $x[0]$ sono $x[N - 1]$ e $x[1]$ e i vicini della cella $x[N - 1]$ sono $x[N - 2]$ e $x[0]$.

Dati i valori correnti *pqr* di tre celle adiacenti, il nuovo valore q' della cella centrale è determinato in base alla tabella seguente (■ = 1, □ = 0):

Configurazione corrente (*pqr*)	■■■	■■□	■□■	■□□	□■■	□■□	□□■	□□□
Nuovo stato della cella centrale (q')	□	□	□	■	■	■	■	□

(si noti che la sequenza □□□■■■■□ = 00011110, che si legge sulla seconda riga della tabella

precedente, rappresenta il numero 30 in binario, da cui il nome "regola 30").

Il file `mpi-rule30.c` contiene lo scheletro di una implementazione seriale dell'algoritmo che calcola l'evoluzione dell'automa. Inizialmente tutte le celle sono nello stato 0, ad eccezione di quella in posizione $N/2$ che è nello stato 1. Il programma accetta sulla riga di comando la dimensione del dominio N e il numero di passi da calcolare (se non indicati, si usano dei valori di default). Al termine dell'esecuzione il processo 0 produce un file `rule30.pbm` contenente una immagine simile alla Figura 2.

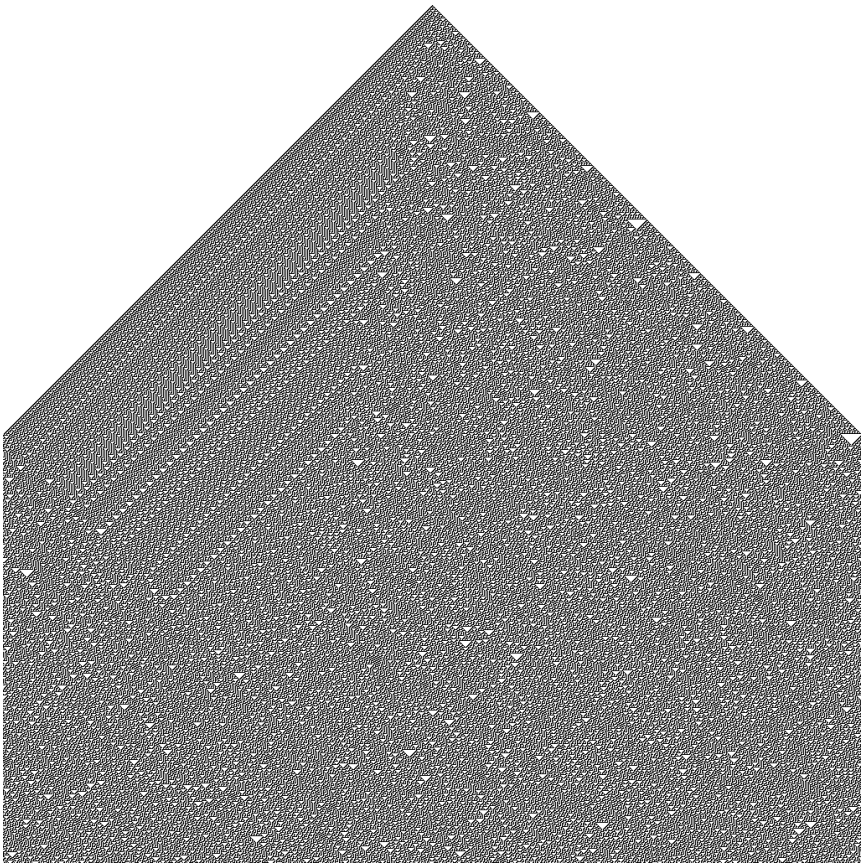


Figura 2: Evoluzione dell'automa cellulare della "regola 30" partendo da una singola cella attiva

Ogni riga dell'immagine rappresenta lo stato dell'automa in un istante di tempo; il colore di ogni pixel indica lo stato di ciascuna cella (nero = 1, bianco = 0). Il tempo avanza dall'alto verso il basso, quindi la prima riga indica lo stato al tempo 0, la seconda riga lo stato al tempo 1 e così via.

Lo scopo di questo esercizio è di sviluppare una versione realmente parallela del programma, in cui il calcolo di ogni riga dell'immagine venga realizzato in parallelo da tutti i processi MPI. Il programma deve operare secondo i passi seguenti (si faccia riferimento alla Figura 3).

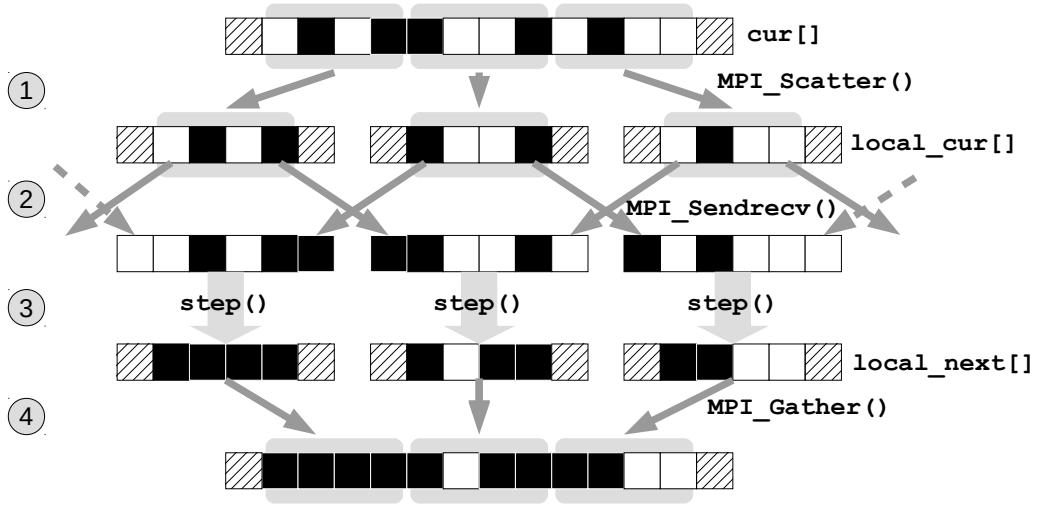


Figura 3: Schema di calcolo dell'evoluzione di un automa cellulare usando MPI

1. Il dominio viene distribuito ai *P* processi MPI usando `MPI_Scatter`; si assuma che *N* sia multiplo di *P*. Ciascuna partizione deve includere due *ghost cells* (una a sinistra e una a destra, mostrate tratteggiate in figura) che sono necessarie per calcolare lo stato successivo, come discusso a lezione.
2. Ogni processo comunica ai vicini i valori delle celle sul proprio bordo usando `MPI_Sendrecv`. Questa fase serve per riempire le *ghost cell* delle partizioni locali.
3. Ciascun processo calcola la configurazione successiva della propria porzione di dominio, sfruttando i valori delle *ghost cells*. Si può fare riferimento alla funzione `step()` inclusa nello scheletro di programma.
4. Ciascun processo invia la propria porzione di dominio, contenente la nuova configurazione, al master che le assembla usando `MPI_Gather` per poi salvarla sul file di output.

Al termine del passo 4 si riparte dal passo 2; dato che il nuovo stato dell'automa è già presente nella memoria di ciascun processo, non serve rifare `MPI_Scatter` ogni volta. Per i dettagli si rimanda ai lucidi in cui abbiamo discusso i pattern per la programmazione parallela. Il file `mpi-rule30.c` va usato come indicazione di massima su come procedere; sarà necessario modificarne pesantemente la struttura per poter realizzare lo schema parallelo descritto sopra.

I punti critici di questo esercizio sono: (*i*) l'uso di `MPI_Scatter()` e `MPI_Gather()` per distribuire e concatenare i sottodomini, e (*ii*) l'uso di `MPI_Sendrecv()` per consentire ai processi lo scambio delle ghost cells.

Per quanto riguarda il punto (*i*), ciascun sottodominio dovrà essere composto da (*N*/comm_sz + 2) elementi (i due elementi aggiuntivi rappresentano le ghost cells). Indichiamo con `cur[]` il dominio completo usato dal master e con `local_cur[]` il sottodominio con (*N*/comm_sz + 2) elementi di ciascun processo. Sebbene non sia necessario per la versione MPI, supponiamo che `cur[]` sia esteso con due ghost cells. Allora la distribuzione di `cur[]` avrà la forma:

```
MPI_Scatter( &cur[1],           /* sendbuf      */
             N/comm_sz,         /* sendcount    */
             MPI_INT,           /* datatype     */
             &local_cur[1],     /* recvbuf      */
             N/comm_sz,         /* recvcount    */
             MPI_INT,           /* datatype     */
             0,                 /* root         */
```

```
        MPI_COMM_WORLD
        );
```

L'istruzione precedente distribuisce *N*/comm_sz elementi di `cur[]`, in modo ciclico, tra i processi MPI; ciascuno di essi riceve il blocco a partire dall'indirizzo di memoria `&local_cur[1]`, lasciando quindi invariato il valore della ghost cell di sinistra e di destra. Si noti che se l'evoluzione dell'automa viene calcolata in parallelo dai processi MPI, non servono le ghost cells sul dominio completo `cur[]` ma solo su `local_cur[]`.

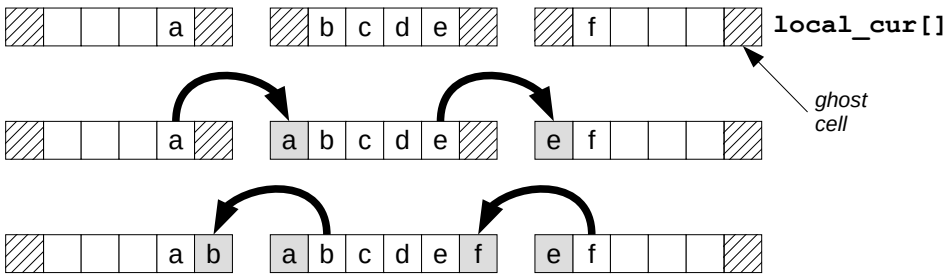


Figura 4: Uso di MPI_Sendrecv() per lo scambio delle ghost cells

Per quanto riguarda il punto (*ii*), lo scambio di celle sul bordo tra processi adiacenti mediante `MPI_Sendrecv()` richiede due fasi (cioè occorre usare due operazioni `MPI_Sendrecv`, si veda la Figura 4): nella prima fase ogni processo invia il valore dell'ultima cella a destra del proprio dominio locale al processo *successivo*, e riceve il valore della ghost cell di sinistra dal processo *precedente*. Di conseguenza, la chiamata `MPI_Sendrecv` deve avere come destinatario il processo successivo e come mittente il processo precedente. Nella seconda fase ogni processo invia il contenuto della prima cella al processo *precedente*, e riceve il valore della ghost cell di destra dal processo *successivo*.

5. Calcolo dell'area dell'unione di N cerchi

Il file `mpi-circles.c` contiene una implementazione seriale di un algoritmo randomizzato di tipo Monte Carlo per stimare l'area dell'unione *N* cerchi. Siano *cx*[*i*], *cy*[*i*] le coordinate del centro del cerchio *i*-esima, e *r*[*i*] il suo raggio; tutti i cerchi sono interamente contenuti all'interno del quadrato avente gli angoli opposti di coordinate (0, 0), (1000, 1000). Poiché i cerchi possono essere collocati in posizioni arbitrarie, potrebbero sovrapporsi in tutto o in parte; di conseguenza non è semplice determinare l'area della loro unione. Un metodo per stimare l'area consiste nell'utilizzare il metodo seguente:

- Si generano *K* punti distribuiti uniformemente all'interno del quadrato di coordinate (0, 0), (1000, 1000). Sia *C* il numero di tali punti che si trovano all'interno di almeno un cerchio.
- L'area *A* dell'unione dei cerchi è data dal prodotto tra l'area del quadrato che li contiene (nel nostro caso 1000×1000) e la frazione di punti che ricadono all'interno di almeno un cerchio, ossia $A = 1'000'000 \times C / S$;

La Figura 5 illustra il concetto. L'area dell'unione dei cerchi viene approssimata come il prodotto tra l'area del quadrato contenente i cerchi e il rapporto tra il numero di punti che si trovano all'interno di almeno un cerchio e il numero totale di punti casuali generati.

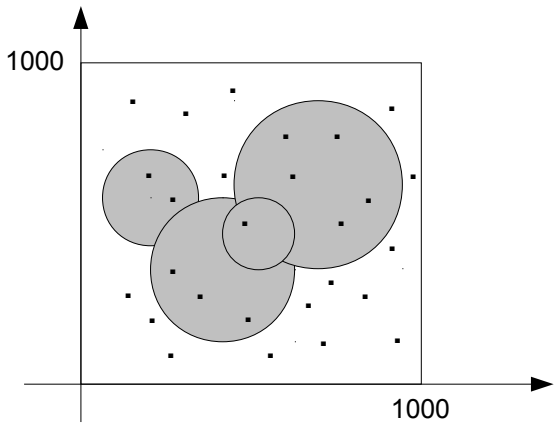


Figura 5: Calcolo dell'area dell'unione di cerchi con metodo Monte Carlo

Viene fornito il file `mpi-circles.c` contenente uno schema di soluzione seriale, in cui il processo 0 esegue tutte le operazioni. Scopo di questo esercizio è di distribuire la computazione tra tutti i processi MPI. È richiesto che solo il processo 0 legga il file di input. Questo significa che solo il processo 0 conosce il numero N di cerchi e le loro coordinate; se tali informazioni sono necessarie agli altri processi, il master le deve comunicare esplicitamente. Il programma deve funzionare correttamente per qualsiasi valore di N e K.

Suggerimento: si potrebbe essere tentati di partizionare i cerchi tra i processi MPI, in modo che ciascun processo gestisca N/P cerchi. Questa però non sarebbe una buona idea: perché?