

Corso di High Performance Computing

Esercitazione MPI del 20/4/2017

Moreno Marzolla

Ultimo aggiornamento: 2017/04/20

Per svolgere l'esercitazione è possibile collegarsi al server `disi-hpc.csr.unibo.it` tramite `ssh`, usando come `username` il proprio indirizzo mail istituzionale completo, e come password la propria password istituzionale (cioè quella usare 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/ex1-mpi.zip
unzip ex1-mpi.zip
cd ex1-mpi/
```

1. Comunicazione ad anello

Realizzare un programma MPI che effettua una comunicazione “ad anello” tra i processi. Più in dettaglio, detto P il numero di processi MPI utilizzati (da specificare con il comando `mpirun`; si deve avere $P > 1$), il programma deve operare come segue:

- Il programma accetta 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 .
- 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 invia il valore $(v + 1)$ al processo $p + 1$. Poiché la comunicazione si considera “ad anello”, il predecessore del processo 0 è il processo $(P - 1)$, 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.

Suggerimento: usare `MPI_Send()`/`MPI_Recv()` prestando attenzione ad evitare deadlock.

2. 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[]` è:

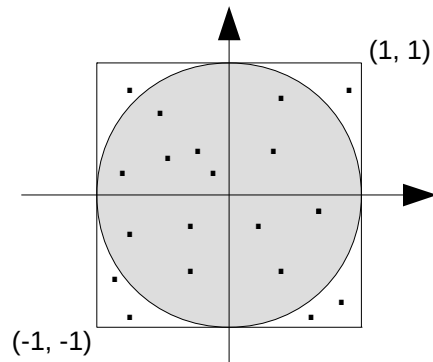
$$s = \sum_{i=0}^{n-1} a[i] \times b[i]$$

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 potrà quindi usare 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 accetta valori di n arbitrari; la soluzione più semplice consiste nel far processare gli elementi in eccesso al master.

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 pi greco. Il principio di funzionamento dell'algoritmo è molto semplice:



- 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 il valore di π come $(4x / N)$

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 utilizzati):

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. Il master calcola la somma x di tutti i valori x_p usando `MPI_Reduce`. A questo punto il master stampa il valore approssimato di π come $(4x / N)$.

4. Distanza di Levenshtein con OpenMP

Il file `omp-levenshtein.c` contiene una implementazione seriale di un algoritmo, basato sulla programmazione dinamica, per il calcolo della distanza di Levenshtein tra due stringhe. La distanza di Levenshtein è una misura di similarità tra due stringhe (più la distanza è alta, più le stringhe sono diverse), e indica il numero minimo di operazioni elementari necessarie per trasformare una delle due stringhe nell'altra. Per operazione elementare si intende: (i) inserimento di un carattere; (ii) cancellazione di un carattere; (iii) sostituzione di un carattere con uno diverso.

La distanza di Levenshtein può essere calcolata mediante la programmazione dinamica; chi è interessato ai dettagli può consultare https://en.wikipedia.org/wiki/Levenshtein_distance oppure i lucidi da 65 in poi all'indirizzo <http://www.moreno.marzolla.name/teaching/algoritmi-e-strutture-dati/2015-2016/L04-Programmazione-Dinamica.pdf>. Conoscere i dettagli di funzionamento dell'algoritmo non è comunque essenziale per svolgere questo esercizio, perché il programma fornito dovrebbe già effettuare il calcolo corretto. Date due stringhe $s[n]$, $t[m]$, indichiamo con $L[i][j]$ la distanza di Levenshtein tra i prefissi composti dai primi i caratteri di s e j caratteri di t ; la distanza complessiva tra s e t è data da $L[n+1][m+1]$. Il calcolo di L viene effettuato da due cicli "for" annidati, ma le dipendenze tra le iterazioni danno origine ad uno stencil a tre punti simile a quella illustrata nel lucido 31 di L05-parallelizing-loops.odp; tale struttura non può essere parallelizzata banalmente, ma può diventarlo se si riempie la matrice L "in diagonale", realizzando una computazione di tipo *wavefront* come mostrato nel lucido 33.

Scopo di questa esercitazione è quello di riscrivere i cicli "for" annidati evidenziati nei commenti della funzione `levenshtein()` per realizzare una visita in diagonale della matrice L , in modo da poter poi applicare un costrutto `#pragma omp parallel for` al ciclo più interno.