Modello di sviluppo della popolazione: Matrice di Leslie

April 24, 2007

1 Scopo del progetto

Lo scopo è quello di creare un programma parallelo in grado di fare una stima di quale sarà la popolazione in Italia in un certo istante di tempo.

Per fare ciò ci siamo appoggiati ad un modello di sviluppo della popolazione: la matrice di Leslie.

2 Definizione del modello

Costruiamo un modello che possa tracciare gruppi di età nel loro sviluppo futuro.

Pecularità del modello: vengono prese in considerazione solo le donne. Si escludono gli uomini per tradizione demografica e per semplificare il modello.

Dividiamo la popolazione femminile in n categorie di età

$$[0,\Delta],[\Delta,2\Delta],...,[(n-1)\Delta,n\Delta]$$

dove Δ è l'intervallo d'età.

Dato n come parametro, calcoliamo Δin modo che $n\cdot\Delta=100$. Dato che una quantità trascurabile di donne sopravvive oltre i 100 anni si assume che nessuna donna possa sopravvivere oltre i 100 anni.

La variabile t misurerà il tempo in Δ anni e $t=0$ rappresenta oggi. Il modello ci darà i valori dei gruppi d'età in istanti di tempo discreti distanziati di Δ; $t=0, t=\Delta, t=2\Delta,\ldots$

Sia $F_i(t)$ il numero di femmine al tempo t nell'i-esimo gruppo d'età, cioè con età compresa nell'intervallo $[i\Delta,(i+1)\Delta]$; sia $F_i(t)$ il vettore colonna:

$$F(t)=\begin{bmatrix} F_0(t) \\ F_1(t) \\ \vdots \\ F_{n-1}(t) \end{bmatrix}$$

che chiameremo vettore di distribuzione d'età.

$F(0)$ è la distribuzione d'età corrente che si suppone nota. Il nostro compito sarà calcolare $F(\Delta), F(2\Delta), \ldots$. A tale scopo avremo bisogno di un coefficienti d_i= 'coefficiente di mortalità dell'i-esimo gruppo' = frazione della popolazione dell'i-esimo gruppo che non sarà più presente nel gruppo $(i+1)$-esimo fra Δ anni perchè morta (non consideriamo l'emigrazione o l'immigrazione).

La frazione che sopravvive sarà $1-d_i = p_i$. Cosideriamo p_i attivo in tutti i periodi futuri della simulazione. Si avrà quindi

$$F_{i+1}(t+\Delta) = p_i \cdot F_i(t) \ \forall t \tag{1}$$

A questo punto bisogna definire un altro coefficiente m_i= 'coefficiente di natalità dell'i-esimo gruppo' = contributo di bambini che le femmine dell'i-esimo gruppo daranno al gruppo iniziale ($[0,\Delta]$) nel periodo $[t, t+\Delta]$.

Si assume che m_i sia costante nel tempo, quindi il numero di neoanati al tempo $t+\Delta$ è

$$F_0(t+\Delta) = \sum_{i=0}^{n-1} m_i \cdot F_i(t) \tag{2}$$

Secondo questo schema il calcolo della popolazione nelle varie fascie del tempo $t+\Delta$ dipende dai valori al tempo t eslusa la fascia F_0 che eredita i nuovi nati dalle fascie che sono fertili al tempo t. Le equazioni (1)e (2) possono essere scritte in forma matriciale

$$\begin{bmatrix} F_0(t+\Delta) \\ F_1(t+\Delta) \\ \vdots \\ F_{n-1}(t+\Delta) \end{bmatrix} = \begin{bmatrix} m_0 & m_1 & m_2 & \cdots & m_{n-1} \\ p_0 & 0 & 0 & \cdots & 0 \\ 0 & p_1 & 0 & \cdots & 0 \\ \vdots & & & & \\ 0 & 0 & \cdots & p_{n-2} & 0 \end{bmatrix} \cdot \begin{bmatrix} F_0(t) \\ F_1(t) \\ \vdots \\ F_{n-1}(t) \end{bmatrix}$$

in forma matriciale compatta

$$F(t+\Delta) = M \cdot F(t) \ t = 0, 1, \ldots$$

Proprietà:

$$t = 0 \quad F(\Delta) = M \cdot F(0)$$

$$t = \Delta \quad F(2\Delta) = M \cdot F(\Delta)$$

e sostituendo si ha

$$F(2\Delta) = M \cdot M \cdot F(0) = M^2 \cdot F(0)$$

analogamente

$$F(3\Delta) = M \cdot M^2 \cdot F(0) = M^3 \cdot F(0)$$

e quindi in generale

$$F(k\Delta) = M^k \cdot F(0) \; k = 0, 1, 2, \ldots$$

3 Spiegazione del sorgente

Prima di cominciare la spiegazione bisogna sottolineare una cosa. A causa di problemi nell'invio di matrici allocate dinamicamente abbiamo di deciso di considerare ogni matrice come fosse un'array. In questo modo una matrice M 5X5 corrisponde ad avere un array V di 25 elementi e l'elemento $M[i][j]$ corrisponde all'elemento $V[i \cdot 5 + j]$.

Cominciamo, quindi, dagli include e dalle variabili globali:

```
#include <mpi.h>
#include <stdio.h>

int n,size,rank;
```

`#include <mpi.h>` rende disponibili le funzioni delle MPI mentre `#include <stdio.h>` è necessaria per utilizzare alcune funzioni che ci serviranno per la gestione di file. Abbiamo dichiarato `n`,`size` e `rank` di tipo intero e hanno il segue significato:

- `n` è il numero di intervalli d'età da considerare
- `size` è il numero di processi che sono stati attivati
- `rank` è il il rank del processo

Queste variabili sono state dichiarate globali per comodità, avremo potuto benissimo dichiararle nel main ma questo avrebbe aumentato notevolmente i parametri da passare alla maggior parte delle funzioni.

Passiamo ora al main in particolare alle variabili dichiarate.

```
int    i,k,r;
float  *F;
float  *M,*M2;
double t0,t1;
FILE   *dati;
```

La variabile `i` è una semplice variabile di tipo intero che utilizzeremo per effettuare alcuni cicli. L'intero k è il numero di unità temporali da calcolare. Infine `r` è una variabile che useremo per verificare se alcune funzioni sono state eseguite correttamente.

`F` è un puntatore ci servirà per memorizzare i valori della popolazione iniziale e alla fine del programma la stima della popolazione. `M` è un puntatore che conterrà i coefficienti di mortalità e natalità. `M2` è un puntatore che utilizzeremo per il calcolo di M^k.

`t0 e t1` sono delle variabili che verranno utilizzate per il calcolo del tempo di esecuzione.

Passiamo ora alle seguenti istruzioni:

```
r = MPI_Init (&argc, &argv);
if (r != MPI_SUCCESS){
 printf (”Errore di inizializzazione\n”);
 MPI_Abort (MPI_COMM_WORLD, r);
}
r = MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if (r != MPI_SUCCESS){
 printf (”Errore nell’esecuzione di MPI_Comm_rank\n”);
 MPI_Abort (MPI_COMM_WORLD, r);
}
r = MPI_Comm_size (MPI_COMM_WORLD, &size);
if (r != MPI_SUCCESS){
 printf(”Errore nell’esecuzione di MPI_Comm_size\n”);
 MPI_Abort (MPI_COMM_WORLD, r);
}
t0 = MPI_Wtime ();
dati = fopen (“dati”,”r”);

if (dati == NULL){
 if (rank == 0)
  printf (“File ’dati’ non trovato”);
 MPI_Finalize ();
 return 0;
}
```

La funzione `MPI_Init` serve a inizializzare la libreria mpi e a pulire gli argomenti passati da riga di comando da opzioni che riguardavano sole le mpi. Il valore restituito dalla funzione viene assegnato a `r`. L'`if` seguente controlla se `r` è diverso da `MPI_SUCCESS`, in questo caso infatti il programma deve essere terminato perchè significa che c'è stato qualche problema.

Lo stesso discorso vale per le funzioni `MPI_Comm_rank` e `MPI_Comm_size`. `MPI_Abort` termina il programma, `MPI_Comm_rank` ritorna nella variabile `rank` il rank del processo corrente e infine `MPI_Comm_size` carica nella variabile `size` il numero di processi attivati. Da notare che `rank` va da 0 a `size`-1.

`MPI_Wtime` serve per specificare da quale momento del programma vogliamo far partire la misurazione del tempo di esecuzione.

Poi il file contenente i dati di partenza viene aperto in modalità lettura con la funzione `fopen` e viene controllato il valore della variabile `file`. Se `file` è uguale a `NULL` significa che il file non esiste, quindi viene segnalato all'utente e viene terminato il programma.

```
fscanf (dati, ” %d”, &n);
fscanf (dati, ” %d”, &k);
F  = Alloca (n);
M  = Alloca (n*n);
M2 = Alloca (n*n);
```

```
CaricaDati (file, F, M, M2);
```

Viene letto il valore di `n` che ci servirà per allocare l'array `F` e le matrici `M` e `M2`. Subito dopo, infatti, vengono allocate con la funzione `Alloca` che analizzeremo più avanti.

La funzione `CaricaDati`, invece, legge il file e sistema i dati in `F`, `M` e `M2`.

```
if (n%size != 0){
 if (rank == 0)
  printf ("Il numero di processi deve essere divisore");
  printf (" dimensione della matrice cioè %d\n", n);
 MPI_Finalize ();
 return 0;
}

for (i = 0; i < k-1; i++)
 M2 = MoltMatPar (M2,M);
```

Il primo `if` controlla che il numero dei processi sia un divisore di `n` e in caso contrario segnala l'errore e termina il programma. Questo perchè essendo `M` una matrice `n`X`n` e che verrà divisa in `size` parti, se `size` non è un divisore di `n` ci sarebbero dei problemi durante l'esecuzione.

Il `for` sucessivo calcola M^kattraverso `k` moltiplicazioni successive. La funzione `MoltMatPar` esegue, appunto, la moltiplicazione tra `M` e `M2` in parallelo e ne restituisce il risultato. Infine le ultime istruzioni del main:

```
t1 = MPI_Wtime ();

if (rank == 0){
 F = MoltMatVet (M2, F);
 for (i = 0; i < n; i++)
   printf ("%f", F[i]);
  printf ("Tempo:  f\n", t1-t0);
}

MPI_Finalize ();
return 0;
}
```

Viene richiamata la funzione `MPI_Wtime` pre fermare la misurazione del tempo di esecuzione.

L'`if` controlla che il `rank` sia uguale a 0 per stampare solo una volta i risultati.

`MolMatVet` esegue la moltiplicazione tra `M2` e `F` e torna il risultato in `F`. Questo è il risultato finale cioè la popolazione divisa per classi d'età al tempo `k`.

Infine vengono stampati i risultati e viene stampato il tempo di esecuzione.

Pasiamo ora ad analizzare le funzioni.

```
void StampaRisultati (float *F, int k){

 int i;
 int delta;

 delta = 100/n;
 printf ("  eta'   | Femmine dopo\n        |   %d anni\n", k*delta);
 printf ("---------|-------------\n");
 for (i = 0; i < n; i++)
  printf ("%-2d - %-3d |    %-2.2f\n", delta*i, delta*(i+1), F[i]);

}
```

La funzione `StampaRisultati`, come si può facilmente intuire, stampa a video i risultati, cioè la popolazione femminile divisa per classi d'età. Con un minino di conoscienze di C si può capire il suo funzionamento.

Passiamo a qualche funzione più interessante.

```
float *Alloca (int Dim){

 float *vet;

 vet = (float *) calloc (Dim * sizeof (float));
 if (vet == NULL){
  printf ("Memoria esaurita\n");
  MPI_Abort (MPI_COMM_WORLD, -1);
 }

 return (vet);
}
```

Questa funzione alloca un'area di memoria di tanti `float` quanti specificati in `Dim` e ritorna il puntatore ad essa.

Viene dichiarato un puntatore a `float` e a cui viene assegnato il risultato della funzione `calloc`. La funzione `calloc` alloca un'area di memoria che viene specifica (in byte) e la inizializza a 0. Per questo motivo viene utilizzata la funzione `sizeof` che ritorna il numero di byte del tipo di variabile specificato e viene moltiplicato per `Dim`.

Siccome la funzione `calloc` ritorna un puntatore a `void` viene eseguita un operazione di cast, `(float *)`, cioè il puntatore a `void` viene convertito in un puntatore a `float`.

L'`if` che segue controlla se vet è uguale a `NULL`. In questo caso il programma deve essere terminato perchè significa che non c'è più memoria disponibile.

```
void CaricaDati (FILE *dati, float *F, float *M, float *M2){
```

```
 int i;

 for (i = 0; i < n; i++)
  fscanf (dati, " %f", &F[i]);

 for (i = 0; i < n; i++)
  fscanf (dati, " %f", &M[i]);

 for (i = 0; i < n - 1, i++)
  fscanf (dati, " %f", &M[(i+1)*n+1]);

 for (i = 0; i < n * n; i++)
  M2[i] = M[i];

 fclose (file);
}
```

La funzione `CaricaDati` va a leggere il file 'dati' e sistema i valori appena letti nella matrice `M` e nell'array `F`.

I primi tre `for` leggono rispettivamente i valori delle varie classi d'età, i coefficienti di natalità, i coefficienti di mortalità che vengono sistemati nella matrice `M`.

L'ultimo `for` copia la matrice `M` in `M2` e infine la funzione `fclose` chiude il file.

```
float *MoltMatVet (float *M, float *F){

 float *ris;
 int   i,j;

 ris = Alloca (n);

 for (i = 0; i < n; i++){
  ris[i] = 0;
  for (j = 0; j < n; j++)
   ris[i] = ris[i] + F[j] * M[i*n+j];
 }

 free (F);
 return (ris);
}
```

Questa funzione esegue la moltiplicazione tra la matrice `M` e l'array contenente i valori di popolazione di ciascuna classe cioè `F`.

Viene dichiarato un puntatore di tipo float (`ris`) che conterrà il risultato, viene allocato e poi viene eseguita la moltiplicazione con due semplici for concatenati.

Infine viene liberata l'area di memoria occupata da `F` con la funzione `free` e viene restituito il puntatore `ris`.

```
float *MoltMatPar (float *M2, float *M){

 int   i,j,k,r;
 float *ris;

 ris = Alloca (n*n);
 r = MPI_Scatter (M2, n*n/size, MPI_INT,
                  M2, n*n/size, MPI_INT, 0, MPI_COMM_WORLD);
 if (r != MPI_SUCCESS){
  ptrintf ("Errore nell'esecuzione di MPI_Scatter");
  MPI_Abort (MPI_COMM_WORLD, r);
 }

 for (i = 0; i < n/size; i++)
  for (j = 0; j < n; j++){
   ris[i*n+j] = 0;
   for (k = 0; k < n; k++)
    ris[i*n+j] += M2[i*n+k] * M[k*n+j];
  }

 r = MPI_Gather (ris, n*n/size, MPI_INT,
                 ris, n*n/size, MPI_INT, 0, MPI_COMM_WORLD);
 if (r != MPI_SUCCESS){
  printf ("Errore nell'esecuzione di MPI_Gather\n");
  MPI_Abort (MPI_COMM_WORLD, r);
 }

 free(a);
 return (ris);
}
```

`MolMatPar` esegue la moltiplicazione in paralello tra le due matrici M2, M. Prima di tutto vengono dichiarate quattro variabili `i,j,k,` che serviranno per eseguire i tre cicli annidati, e `ris` che conterrà il risultato.

Subito dopo, `ris` viene allocato con la solita funzione `Allocca`. Ora attraverso la funzione `MPI_Scatter` il processo 0 invia a ciascuno degli altri processi un certo numero di righe della matrice `M2`.

Il numero delle righe è dato da $n/size$. Analizziamo meglio il suo funzionamento partendo dalla sua dichiarazione:

```
int MPI_Scatter {
        void         *sendbuf,
        int          sendcnt,
        MPI_Datatype sendtype,
        void         *recvbuf,
        int          recvcnt,
        int          root,
        MPI_Comm     comm }
```

- `sendbuf` è l'indirizzo iniziale di una serie di dati contigua che devono essere inviati
- `sendcnt` è il numero di elementi che devono essere inviati a ciascun processo
- `sendtype` è il tipo dei dati che devono essere inviati
- `recvbuf` è l'indirizzo iniziale di dove devono essere caricati i dati ricevuti
- `recvcnt` è il numero di elementi che vengono ricevuti
- `recvtype` è il tipo di dati ricevuti
- `root` è il rank del processo che invia i dati
- `comm` è il communicatore cioè l'insieme dei processi a cui si fa riferimento

Nel nostro caso, quindi, il processo 0 invierà ad ogni processo $n \cdot n/size$ elementi della matrice.

Supponiamo, ad esempio, di avere una matrice 5X5 e 5 processi ogni processo si troverà con una riga della matrice `M2` esattamente l'i-esima riga, dove i è il rank del processo. Dopodichè si controlla, come fatto in precedenza, che non ci sia stati problemi, controllando il valore restituito.

Le istruzioni seguenti sono i tre `for` annidati che seguono la moltiplicazione. Vi chiederete come mai `M` non viene inviata a tutti processi? Torniamo un attimo al main e ridiamo un occhiata alle seguenti istruzioni:

```
for (i = 0; i < k-1; i++)
 M2 = MoltMatPar (M2,M);
```

Supponiamo di dover calcolare M^3, la prima volta che viene esguito il ciclo `M2` ed `M` sono uguali.

Viene chiamata la funzione `MoltMatPar, M2` viene inviata a tutti i processi, anche se non servirebbe, mentre `M` non viene inviata in quanto tutti i processi hanno ce l'hanno già. Viene calcolata M^2 e viene assegnata ad `M2`. Secondo ciclo, viene di nuovo invita `M2` che corrisponde ad M^2, mentre `M` non serve inviarla perchè è sempre la stessa: non cambia mai nel corso del programma! Sarebbe quindi inutile inviarla ogni volta a tutti i processi.

Analizziamo ora i tre cicli annidati. E' molto semplice da capire.

Supponiamo di avere una normale moltiplicazione tra due matrici sarebbero bastate questi tre cicli:

```
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++){
  ris[i*n+j] = 0;
  for (k = 0; k < n; k++)
   ris[i*n+j] += M2[i*n+k] * M[k*n+j];
 }
```

Nel nostro caso però ogni processo possiede una o più righe di M2, per la precisione $n/size$ e ogni processo dovrà moltiplicare le $n/size$ righe per la matrice M1 e poi sarà il compito della `MPI_Gather` ricomporre la matrice risultante. Otteniamo quindi:

```
for (i = 0; i < n/size; i++)
 for (j = 0; j < n; j++){
  ris[i*n+j] = 0;
  for (k = 0; k < n; k++)
   ris[i*n+j] += M2[i*n+k] * M[k*n+j];
 }
```

Passiamo ad analizziamo la funzione `MPI_Gather`:

```
int MPI_Gather {
        void         *sendbuf,
        int          sendcnt,
        MPI_Datatype sendtype,
        void         *recvbuf,
        int          recvcnt,
        MPI_Datatype recvtype,
        int          root,
        MPI_Comm     comm }
```

- `sendbuf` è l'indirizzo iniziale di una serie di dati contigua che devono essere inviati
- `sendcnt` è il numero di elementi che ciascun processo deve inviare
- `sendtype` è il tipo dei dati che devono essere inviati
- `recvbuf` è l'indirizzo iniziale di dove devono essere inseriti i dati ricevuti
- `recvcnt` è il numero di dati che devono essere ricevuti da ciascun processo
- `recvtype` è il tipo dei dati ricevuti
- `root` è il processo che riceve tutti i dati
- `comm` è il communicatore

Come al solito si controlla la corretta esecuzione della funzione `MPI_Gather`. Poi la matrice `ris` quindi viene ricomposta dal processo 0. Poi viene liberata l'area di memoria occupata da `M2` e viene ritornato dalla funzione `ris`.