LAB. DI CALCOLO PARALLELO E DISTRIBUITO – MARCELLINO

UNIVERSITÀ DEGLI STUDI DI NAPOLI
“PARTHENOPE”

Appunti a cura di
FIORENTINO MICHELE

LEZ 7:10

Es. definizione di ambiente (cod1_defAmbiente.c)

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

int main(int argc, char *argv[]){
    int menum, nproc;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    printf("Sono %d di %d\n", menum, nproc);

    MPI_Finalize();
    return 0;
}
```

Tutti i processori di MPI_COMM_WORLD stampano a video il proprio identificativo `menum` ed il numero di processori `nproc`. Ricordo che “rank” restituisce l’id del processore che chiama la funzione, “size” il numero dei processori di quell’ambiente.

Es. scambio di un numero fra due processori (cod2_scambio2p.c)

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

int main(int argc, char *argv[]){
    int menum;
    int n, tag, num;

    MPI_Status info;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &menum);

    if(menum==0){
        scanf("%d", &n);
        tag=10; //arbitrary value
        MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
    }
    else{
        tag=10;
        MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD);
    }

    MPI_Get_count(&info, MPI_INT_&num);
    MPI_Finalize();
    return 0;
}
```

`tag` ci serve per individuare univocamente il messaggio nel contesto;
`num` e `info` ci servono solo per eventuale debugging, con `MPI_Get_Count()`.

Innanzitutto inizializzo l'ambiente e ogni processore (attraverso rank) memorizza in menum il proprio id.
In questo caso non ci serve nproc, in quanto già sappiamo che i processori sono 2.

Es. broadcast su più processori (cod3_broadcastNProc.c)

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

int main(int argc, char *argv[]){
    int menum, nproc;
    int n, tag, num;

    MPI_Init(&argc, &argv);
    MPI_COMM_rank(MPI_COMM_WORLD, &menum);

    if(menum==0){
        scanf("%d", &n);
    }

    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

Il processore P_0 legge un intero e lo memorizza in `n`. Poi dovrà occuparsi di spedire il valore di *n* a tutti gli altri processori. Per questo usiamo MPI_Bcast().

In particolare, il processore P_0 spedice `n`, che è di tipo MPI_INT e di dimensione 1, a tutti i processori dell'ambiente MPI_COMM_WORLD.
Notare che non c'è bisogno di un `if` per individuare il master, in quanto già nella funzione stessa indichiamo chi sia il `root/master` (0, cioè P_0).

Somma di N numeri – parte di Lettura e Distribuzione dei dati

Notare che potremmo fare:

1. la **spedizione da P_0 agli altri processori uno per volta**, attraverso un ciclo for, dove il processore master invia al processore i-simo il valore di *n* attraverso una *send*. Gli altri processori invece aspettano di ricevere il valore con una *recv*.
È importante tener conto del valore del tag.
Il master potrebbe calcolarli come `tag=10+i`, dove `i` rappresenta l'id del processore;
Gli altri potrebbero calcolarli come `tag=10+menum`, dove `menum` è l'id del p. corrente.

2. la **spedizione da P_0 agli altri processori contemporaneamente**, attraverso il broadcast.

Ogni processore terrà conto di quanti numeri del vettore x dovrà sommare attraverso una variabile `nloc` = `n/nproc`.
C'è da considerare che non sempre il numero di elementi da sommare sia interamente divisibile per il numero di processori (es. 4p, ma 10 numeri), dunque dovrò anche calcolarmi un resto `rest` = `n%nproc`.
I numeri rimanenti saranno assegnati ad altri processori, ma ciò significa che dobbiamo modificare la `nloc` di questi processori. Un'idea è che per ogni processore il quale `menum<rest`, incrementiamo `nloc` di 1, ovvero hanno un elemento in più.

Dobbiamo occuparci poi dell'allocazione di `xloc`, cioè il vettore locale di ogni processore.
Nel caso del processore master, `xloc=x`, in quanto parte dall'indirizzo base di x.
Useremo una variabile `tmp` per tener traccia del *numero* di elementi su cui dovrà lavorare un dato processore, che ovviamente sarà inizializzato ad `nloc`.
Useremo una variabile `start` per tener traccia dell'indirizzo base degli altri processori.

Il ciclo for parte da 1 (dunque il secondo processore) a nproc.
L'indirizzo base per gli altri processori sarà dato da `start+temp`, dove `start` incrementerà sempre di più ad ogni iterazione.
Ci calcoliamo anche il tag per il processore i-simo.
L'if ci serve per "regolare" `tmp`: infatti ricordiamo che tutti i numeri il quale `i<rest` hanno un valore in più da calcolare, ma una volta che arriviamo a `i==rest` possiamo decrementare `tmp`. (tmp deve essere decrementato solo una volta!)

Es: se ho 4p, 10n; allora rest=2; dunque p0=3, p1=3, p2=2, p3=2.
Alla prima iterazione i<rest (1<2), dunque tmp rimane 3; già dalla seconda i == rest (infatti i=2) e quindi tmp diventa 2 e rimane 2 fino alla fine del programma.

- **Versione 1, uno alla volta.** (sommaN_LettDist_v1.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]){
    int menum, nproc;
    int n, nloc, tag, i;
    int *x, *xloc;

    MPI_Status status;  //solo per questo file

    MPI_Init(&argv, &argc);
    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    if(menum==0){

        //P0 legge n e tutto il vettore x
        printf("Quanti numeri inserire? ");
        scanf("%d",&n);
        x = (int*) malloc(sizeof(int)*n);
        printf("\nInserire i numeri:\n");
        for(i=0; i<n; i++){
            scanf("%d",&x[i]);
        }

        //spedizione di n da P0 agli altri processori UNO PER VOLTA
        for(i=1; i<nproc; i++){
            tag=10+i;
            MPI_Send(&n,1,MPI_INT,i,tag,MPI_COMM_WORLD);
        }

    }
    else{
        tag = 10+menum;
        MPI_Recv(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD, &status);
    }

    //calcolo quanti n. dovrà calcolare ogni p.
    nloc = n/nproc;
    rest = n%nproc;
    if(menum<rest) nloc++;

    if(menum==0){
        xloc = x;
        int tmp = nloc;
        int start = 0;

        for(i=1; i<nproc; i++){
            start = start+tmp;
            tag = 50+i;
            if(i==rest) tmp=tmp-1; //solo 1 volta!
            MPI_Send(&x[start],tmp,MPI_INT,i,tag,MPI_COMM_WORLD);
        }
    }
    else{
        tag = 50+menum;
        MPI_Recv(xloc,nloc,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
    }

}

```

- **Versione 2, contemporaneamente.** (sommaN_LettDist_v2.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]){
    int menum, nproc;
    int n, nloc, tag, i;
    int *x, *xloc;

    MPI_Status status;  //solo per questo file

    MPI_Init(&argv, &argc);
    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    if(menum==0){
        //P0 legge n e tutto il vettore x
        printf("Quanti numeri inserire? ");
        scanf("%d",&n);
        x = (int*) malloc(sizeof(int)*n);
        printf("\nInserire i numeri:\n");
        for(i=0; i<n; i++){
            scanf("%d",&x[i]);
        }
    }

    //spedizione di n da P0 agli altri processori CONTEMP.
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

    //calcolo quanti n. dovrà calcolare ogni p.
    nloc = n/nproc;
    rest = n%nproc;
    if(menum<rest) nloc++;

    if(menum==0){
        xloc = x;
        int tmp = nloc;
        int start = 0;

        for(i=1; i<nproc; i++){
            start = start+tmp;
            tag = 50+i;
            if(i==rest) tmp=tmp-1; //solo 1 volta!
            MPI_Send(&x[start],tmp,MPI_INT,i,tag,MPI_COMM_WORLD);
        }
    }
    else{
        tag = 50+menum;
        MPI_Recv(xloc,nloc,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
    }

}
```

LEZ 16:19

OpenMP viene implementato da molti compilatori, tra questi il gcc.
Per compilare, basta aggiungere al comando di compilazione l'opzione -fopenmp.

gcc -fopenmp -o nomeExe nomeSorg.c

Per conoscere le caratteristiche della nostra CPU possiamo fare:

cat /proc/cpuinfo (o anche `lscpu`).

Si possono modificare la variabili d'ambiente proprie dello standard OpenMP, ad esempio per fare delle prove:

export OMP_NUM_THREADS=2.

nb: una direttiva deve essere sempre seguita da una newline. Per questo motivo, quando scriviamo "#pragma omp parallel", il '{' deve andare a capo (altrimenti ci viene dato un errore).

Se una direttiva ha una sola istruzione, le parentesi graffe non sono necessarie.

Es. Hello World (ompHelloWorld.c)

Abbiamo una variabile privata (idt = id del thread) e una shared (numt = numero di thread). Possiamo ottenere queste due informazioni attraverso quelle due direttive.

```c
#include <omp.h>
#include <stdio.h>

int main(){
    int idt, numt;
    #pragma omp parallel private(idt), shared(numt)
    {
        idt = omp_get_thread_num();
        numt = omp_get_num_threads();
        printf("Hello from thread %d, of nthreads %d\n", idt, numt);
    }

    return 0;
}
```

Es. Somma 1 Strategia (n divisibile per numt) (somma1str_v1.c)

Vogliamo effettuare la somma di `n` numeri.
Ogni thread deve conoscere quanti numeri deve sommare, quindi ogni thread avrà un proprio `nloc`, che sarebbe `n/numt` (questa versione considera `n` divisibile per numt).

Effettuiamo somme di numeri float. `sumtot` conterrà la somma finale, `sum` conterrà la somma privata di ogni thread, e `*a` conterrà gli *n* elementi (la dimensione verrà allocata dinamicamente).

`sum` e `nloc` saranno private, mentre `sumtot` sarà shared.
nb: nloc potrebbe anche non essere dichiarato privato se ci fosse esatta divisibilità, ma per rimanere in linea con la v2 lo dichiariamo privato.

In questo esempio, ogni thread stampa anche il suo id, il numero di elementi che somma, e la sua somma parziale.

Le somme parziale sono tutte effettuate sullo stesso array `a`. I thread operano in zone diverse in base al loro identificativo che usano come *offset* (calcolato come "`nloc*idt`").
Infine, ogni thread aggiunge la propria somma parziale alla somma totale `sumtot`, che sarà acceduta in mutua esclusione (perché shared).

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(){
    int n, nloc, idt, numt;
    float sumtot, sum, *a;
    sumtot=0;

    printf("Inserire n: ");
    scanf("%d", &n);
    a=(float*)calloc(n,sizeof(float));
    printf("Inserire i numeri da sommare:\n");
    for(int i=0; i<n; i++) scanf("%f", &a[i]);

    #pragma omp parallel private(sum,nloc,idt), shared(sumtot, numt)
    {
        idt = omp_get_thread_num();
        numt = omp_get_num_threads();
        nloc = n/numt;

        sum=0;
        for(int i=0; i<nloc; i++) //somma parziale
            sum += a[i+nloc*idt];

        sumtot+=sum;    //aggiunta somma parziale a somma finale
    }

    printf("somma totale: %f\n", sumtot);
    return 0;
}
```

Es. Somma 1 Strategia (n non divisibile per numt) (somma1str_v2.c)

In questo caso, si devono prendere degli accorgimenti:

Nella direttiva, subito dopo il calcolo di `n/numt`, è necessario verificare anche il resto della divisione `r = n%numt`.
Se il resto non è 0, tutti i processori con id strettamente minore del resto devono occuparsi di sommare un elemento in più (idt < r), dunque il loro nloc=nloc+1;

Inoltre, faremo uso della variabile **`step`** per far sapere ad ogni core di quali elementi si deve occupare. In particolare, possiamo considerare *step* come un offset per tutti quei thread che NON DEVONO occuparsi del numero in più, e quindi "iniziano di qualche *passo* più in avanti".

- se idt < r, allora step = 0;
- altrimenti, step = r; (l'offset che devono applicare gli altri thread).

Ad es. se ho numt=4 e n=10,
i thread 0 e 1 avranno step=0 e nloc=3, mentre
i thread 1 e 2 avranno step = 2 e nloc=2.

Tenendo conto che: `sum += a[i+nloc*idt+step];`
I thread si occuperanno delle somme di questi numeri:

- t_0:
  sum += a[0+0+0]; // a[0]
  sum += a[1+0+0]; // a[1]
  sum += a[2+0+0]; // a[2]
- t_1:
  sum += a[0+3+0]; // a[3]
  sum += a[1+3+0]; // a[4]
  sum += a[2+3+0]; // a[5]
- t_2:
  sum += a[0+4+2]; // a[6]
  sum += a[1+4+2]; // a[7]
- t_3:
  sum += a[0+6+2]; // a[8]
  sum += a[1+6+2]; // a[9]

Infatti (somma primo 10 numeri):

```
sp del thread 2 di 4 thread: 15.00,      nloc: 2, step: 2
sp del thread 0 di 4 thread: 6.00,       nloc: 3, step: 0
sp del thread 1 di 4 thread: 15.00,      nloc: 3, step: 0
sp del thread 3 di 4 thread: 19.00,      nloc: 2, step: 2
somma totale: 55.000000
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(){
    int n, nloc, idt, numt, r, step;
    float sumtot, sum, *a;
    sumtot=0;

    printf("Inserire n: ");
    scanf("%d", &n);
    a=(float*)calloc(n,sizeof(float));
    printf("Inserire i numeri da sommare:\n");
    for(int i=0; i<n; i++) scanf("%f", &a[i]);

    #pragma omp parallel private(sum,nloc,idt,step), shared(sumtot, numt, r)
    {
        idt = omp_get_thread_num();
        numt = omp_get_num_threads();
        nloc = n/numt;
        r = n%numt;

        if(idt < r){
            nloc++;
            step = 0;
        } else {
            step = r;
        }

        sum=0;
        for(int i=0; i<nloc; i++) //somma parziale
            sum += a[i+nloc*idt+step];

        sumtot+=sum;    //aggiunta somma parziale a somma finale
    }

    printf("somma totale: %f\n", sumtot);
    return 0;
}
```

Es. Calcolo PI (con for, parallel e critical)

Ricordiamo la formual per calcolare PI:

$$\int_0^1 \frac{4}{1+x^2}dx = \pi \quad \Rightarrow \quad \sum_{i=1}^{N} \frac{4h}{1+\left[\left(i-\frac{1}{2}\right)h\right]^2} = \pi$$

Il **codice seriale** è: (pi.c)

```c
#include <stdio.h>
#define N 10000000

int main(int argc, char **argv)
{
    long int i, n = N;
    double x, dx, f, sum, pi;
    printf("numero di intervalli: %ld\n", n);
    sum = 0.0;

    dx = 1.0/(double)n;
    for (i = 1; i<=n; i++){
        x = dx*((double)(i-0.5));
        f = 4.0/(1.0+x*x);
        sum+=f;
    }

    pi = dx*sum;

    printf("PI %.24f\n", pi);

    return 0;
}
```

nb: dx sarebbe lo “step”.
Ricordiamo che sarebbe una somma di aree di rettangoli.

Il codice `pi_par.c` è una primo approccio di parallelizzazione, tuttavia ha un problema: non c’è sincronizzazione per la variabile *sum*. Per questo dobbiamo usare il costrutto critical.

dx e *n* sono solo in lettura, dunque possono anche essere shared.

Il **codice parallelo con costrutto critical** è: (pi_par_v2.c)

```c
#include <stdio.h>
#define N 100000000

int main(int argc, char **argv)
{
    long int i, n = N;
    double x, dx, f, sum, pi;
    printf("numero di intervalli: %ld\n", n);
    sum = 0.0;

    dx = 1.0/(double)n;
    #pragma omp parallel for private(x,f,i) shared(dx, sum,n)
    for (i = 1; i<=n; i++){
        x = dx*((double)(i-0.5));
        f = 4.0/(1.0+x*x);
        #pragma omp critical
        sum+=f;
    }

    pi = dx*sum;

    printf("PI %.24f\n", pi);

    return 0;
}
```

nb: L'aritmetica floating point non è associativa, cioè (a+b)+c NON è generalmente lo stesso di a+(b+c). Le differenze sono minime (entrambi i risultati andranno bene per la maggior parte delle applicazioni), ma saranno visibili.

In quanto l'ordine delle somme in parallelo è casuale, ad ogni esecuzione si otterrà un risultato leggermente diverso.

Se invece usiamo la reduction, questo problema non ce l'abbiamo (vedi pagina dopo).

Invece del costrutto critical, possiamo usare la *reduction*.
La reduction anche evita il verificarsi di race condition (crea una variabile privata per ogni thread che poi combina nell'unica variabile shared).

Il **codice parallelo con clausola reduction** è: (pi_par_v3.c)

```c
#include <stdio.h>
#define N 100000000

int main(int argc, char **argv)
{
    long int i, n = N;
    double x, dx, f, sum, pi;
    printf("numero di intervalli: %ld\n", n);
    sum = 0.0;

    dx = 1.0/(double)n;
    #pragma omp parallel for private(x,f,i) shared(dx,n) reduction(+:sum)
    for (i = 1; i<=n; i++){
        x = dx*((double)(i-0.5));
        f = 4.0/(1.0+x*x);
        sum+=f;
    }

    pi = dx*sum;

    printf("PI %.24f\n", pi);

    return 0;
}
```

nb: in questo caso non abbiamo lo stesso problema della Critical in quanto abbiamo tutte variabili private *sum* che poi verranno sommate nell'unica variabile condivisa *sum* (ad ogni esecuzione sempre nello stesso ordine).

Inoltre, sarebbe meglio usare la reduction rispetto alla critical per ragioni di performance.

Es. Modalità del costrutto for

(staticSchedule_ex.c | dynamicSchedule_ex.c | dynamicScheduleChunk_ex.c | guidedSchedule_ex.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

#define N 1000
#define CHUNK 10

int main(){
    int i;

    //------------------------- STATIC

    #pragma omp parallel for private(i) schedule(static) num_threads(4)
        for(i=0; i<N; i++){
            //aspetta un numero "i" di secondi
            sleep(i);
            printf("Il thread %d ha completato l\'operazione %d.\n", omp_get_thread_num(),i);
        }
    printf("Tutti i thread hanno terminato!\n");

    //------------------------- DYANMIC

    #pragma omp parallel for private(i) schedule(dynamic) num_threads(4)
        for(i=0; i<N; i++){
            //aspetta un numero "i" di secondi
            sleep(i);
            printf("Il thread %d ha completato l\'operazione %d.\n", omp_get_thread_num(),i);
        }
    printf("Tutti i thread hanno terminato!\n");

    //------------------------- DYNAMIC CHUNK

    #pragma omp parallel for private(i) schedule(dynamic, CHUNK) num_threads(4)
    for(i=0; i<N; i++){
        //aspetta un numero "i" di secondi
        sleep(i);
        printf("Il thread %d ha completato l\'operazione %d.\n", omp_get_thread_num(),i);
    }
    printf("Tutti i thread hanno terminato!\n");

    //------------------------- GUIDED

        #pragma omp parallel for private(i) schedule(guided) num_threads(4)
        for(i=0; i<N; i++){
            //aspetta un numero "i" di secondi
            sleep(i);
            printf("Il thread %d ha completato l\'operazione %d.\n", omp_get_thread_num(),i);
        }

    printf("Tutti i thread hanno terminato!\n");


    return 0;
}
```

LEZ 25: Algoritmo per il prodotto Matrice X Vettore (OpenMP)

Algoritmo in sequenziale

L'i-simo elemento di `c` è il prodotto scalare della i-sima riga di `A` per il vettore `b`.

```
1  int main(int argc, char** argv){
2
3      if(argc<4){
4          printf("Not enough arguments. Insert: rows, cols, maxVal\n");
5          return -1;
6      }
7
8      //INITIALIZE
9
10     int **A;
11     int *b;
12     int n = atoi(argv[1]);
13     int m = atoi(argv[2]);
14     int maxVal = atoi(argv[3]);
15
16     srand(time(NULL));
17     createMatrix(&A,n,m,maxVal);
18     createVector(&b,m,maxVal);
19
20     printMatrix(A,n,m);
21     printVector(b,m);
22
23     //PRODUCT
24
25     int *c = calloc(n,sizeof(int));
26
27     int i,j;
28     for(i=0; i<n; i++)
29         for(j=0; j<m; j++)
30             c[i] = c[i]+A[i][j]*b[j];
31
32
33     printVector(c,n);
34     return 0;
35 }
```

Funzioni utili

```c
typedef int* Array;
typedef int** Matrix;

void createMatrix(Matrix *A, int n, int m, int maxVal){
    int i,j;
    *A = malloc(n*sizeof(int*));
    for(i=0; i<n; i++)
        (*A)[i] = malloc(m*sizeof(int));

    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            (*A)[i][j] = rand()%maxVal;
}

void createVector(Array *b, int m, int maxVal){
    *b = malloc(m*sizeof(int));
    for(int i=0; i<m; i++)
        (*b)[i] = rand()%maxVal;
}

void printMatrix(Matrix A, int n, int m){
    printf("A =\n");
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            printf("%d\t", A[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void printVector(Array b, int m){
    printf("b =\n");
    for(int i=0; i<m; i++)
        printf("%d\t", b[i]);
    printf("\n\n");
}
```

Nb: in questo caso abbiamo passato i vettori A e b per riferimento, tuttavia avremmo potuto anche ritornare direttamente la matrice/vettore allocato.
(in tal caso avremmo passato **A e *b invece di ***A e **b).

Algoritmo matriceXvettore in parallelo (OpenMP)

Ricordiamo:

La **decomposizione** del problema avviene direttamente sulla matrice A. Nessuno invia niente a nessuno perché tutto è già nella memoria condivisa.
Lo stesso vale per il vettore b.

Ognuno dei `p` core si occuperà di uno dei blocchi di righe della matrice A.
I `p` core si occuperanno di TUTTO il vettore b.

I **passi** da eseguire sono fondamentalmente questi:

Nella funzione **matxvet()**:

- SOLO il processore Master si occuperò dell'allocazione della memoria per il vettore risultato.
- Il doppio ciclo for che si occupa del prodotto matXvet verrà eseguito in parallelo. Vista la situazione possiamo usare insieme alla direttiva `parallel` anche la direttiva `for`: tutti fanno la stessa cosa MA distribuendosi le iterazioni.
  Se volessimo, si può scegliere la distribuizione usando la clausola *schedule*.
- In linea generale, vogliamo che sia tutto condiviso tranne gli indici (ogni core dovrà averne una sua copia).

Se ponessimo in private anche *m*, *n*, *A* e *x* questi verrebbero de-inizializzati.
Il vettore risultato `b` deve essere posto in shared, altrimenti non sarebbe accessibile al di fuori della regione parallela.

In questo caso RITORNIAMO il vettore (e non lo passimo per riferimento). Non dovrebbero esserci grosse differenze.

Nb: rispetto all'algoritmo sequenziale, qui matrice e vettori sono double.

```
1  double* matxvet(double **A, double *x, int n, int m){
2      int i,j;
3      double *b = calloc(n,sizeof(double)); //allocation
4
5      #pragma omp parallel for default(none) shared(A,x,b,n,m) private(i,j)
6      for(i=0; i<n; i++)
7          for(j=0; j<m; j++)
8              b[i] += A[i][j]*x[j];
9
10     return b; //returned from master
11 }
```

Per generare un numero double casuale:

```
((double)rand() * ( high - low ))/(double)RAND_MAX + low;
```

nel nostro caso:

```
((double)rand()*maxVal)/(double)RAND_MAX;
```