UNIVERSITÀ
DEL SALENTO

Parallel numerical integration using Gaussian Quadrature

Andrea Vedruccio

Matricola 20051313

CDLM
Computer Engineering

Corso
Parallel Algorithms A.A. 2020/2021

Docente
Chiar.mo Prof. Massimo Cafaro

INDICE

1. INTRODUZIONE

Data ***f,*** una funzione reale definita su un intervallo [a; b], supponiamo di voler calcolare l'integrale:

$$I = \int_a^b f(x)dx$$

Nel caso in cui la ***f*** sia una funzione continua, il teorema fondamentale del calcolo integrale assicura l'esistenza su [a; b] di una funzione F, primitiva della f, tale che

$$I = F(b) - F(a)$$

La funzione F non risulta essere sempre esprimibile in termini di funzioni elementari, e quando questo è possibile, il calcolo di *F(b)* e *F(a)* può essere difficoltoso. Per questo è importante disporre di tecniche numeriche per il calcolo approssimato dell'integrale.

Una formula esplicita che permetta di approssimare *I* viene detta *formula di quadratura* o formula d'integrazione numerica.

Le formule di quadratura che consideriamo sono le cosiddette formule *interpolatorie* che si ottengono sostituendo alla *f* un polinomio *p*: l'integrale di *p* fornisce l'approssimazione di *I.*

Queste formule hanno la forma

$$I_{n+1} = \sum_{i=0}^{n} (w_i \mathrm{f}(x_i))$$

dove gli $n + 1$ punti $\boldsymbol{x_i}$ ϵ [a; b] per i=0,…,n, sono i **nodi** della formula e i numeri $\boldsymbol{w_i}$ sono i **pesi** (o pesi) della formula.

1.1. Integrazione di gauss

Le formule di Gauss sono formule di quadratura interpolatorie dove i nodi si scelgono in modo di rendere massimo il grado di precisione.

L'integrale esatto può essere approssimato con la formula di quadratura di gauss:

$$I \cong \left(\frac{b-a}{2}\right) \sum_{i=1}^{n} (w_i \mathrm{f}(x_i))$$

I **nodi**, reali e distinti, appartenenti all'intervallo (-1,1), della formula di quadratura di Gauss sono le **radici** dei **polinomi di Legendre** definiti come

$$\int_{-1}^{1} L_i(x)L_m(x) = 0 \; se \; i \neq m$$

Ricorsivamente $L_0(x) = 1, \quad i = 0$

$L_1(x) = \mathrm{x}, \quad i = 1$

.

.

$$L_{i+1}(x) = \frac{2i+1}{i+1} - \frac{i}{i+1} L_{i+1}(x) \quad i = 1,2,\ldots$$

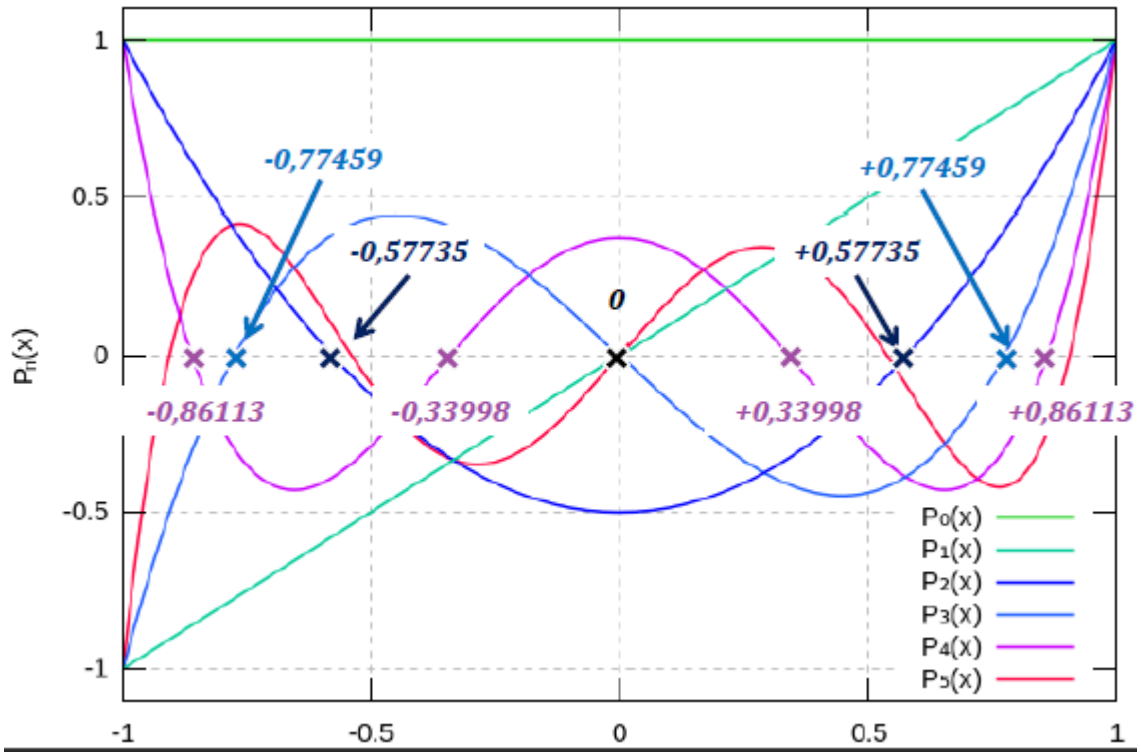

Figura 1-Grafico dei polinomi di Legendre per n ≤ 5

Dove:

$$P_0(x) = 1;$$
$$P_1(x) = x;$$
$$P_2(x) = \frac{1}{2}(3x^2 - 1);$$
$$P_3(x) = \frac{1}{2}(5x^3 - 3x);$$
$$P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3);$$
$$P_5(x) = \frac{1}{8}(63x^5 - 70x^3 + 15x);$$

I **pesi** vengono calcolati integrando i termini di interpolazione di **Lagrange** da -1 a 1

$$w_i = \int_{-1}^{1} L_i(x)dx$$

dove $L_i(x)$ può anche essere espresso come

$$L_i(x) = \prod_{j=0;j\neq i}^{n} \left(\frac{x - x_j}{x_i - x_j}\right)$$

Dove x_i è la radice i-esima del polinomio di Legendre di grado n

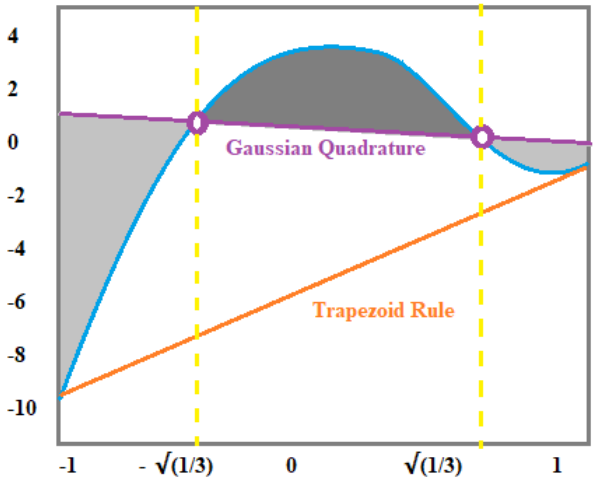

Figura 2-Esempio per n=2 in confronto con il metodo del trapezio

2. IMPLEMENTAZIONE

E’ possibile presentare l’implementazione dell’algoritmo suddividendolo in 2 step.

In un primo momento sono calcolati i nodi $\boldsymbol{x_i}$ e i pesi $\boldsymbol{w_i}$ nel secondo momento è calcolata la soluzione

$$I \cong \left(\frac{b-a}{2}\right)\sum_{i=1}^{n}(w_i \mathrm{f}(x_i))$$

ALGORITMO

```
**************INIZIO*************
j=0
my_a_n = inizio intervallo dove calcolare le radici
my_b_n = fine intervallo dove calcolare le radici
for (x = my_a_n to my_b_n) do
      //calcolo radici del polinomio di legendre con il metodo della bisezione
      root=bisection()
      //Invio root al Master process *
      Send(root)
      //Master process salva le radici ricevute in un array
      Recv (root)
      Xi[j]=root
      j++
endfor
if(Master Process)
//ordina gli elementi ricevuti
Qsort(X[N])
//Invia l’array delle radici a tutti i processi
Broadcast(X[N])
Endif
//Calcolo i pesi con le radici trovate
i=0
my_a = inizio intervallo dove calcolare i pesi
my_b = fine intervallo dove calcolare i pesi
for (x = my_a to my_b) do
```

```
      peso=CalcolaPeso()
      risultato_parziale =peso*f(xi)
endfor
I = ((b-a)/2) *risultato_parziale
return I;
************FINE************
```

Le radici del polinomio di Legendre sono calcolate con il *bisection method*, ovvero, data l'equazione f(x)=0 definita e continua in un intervallo [a,b] tale che $f(a) \cdot f(b) < 0$, è allora possibile calcolarne un'approssimazione in [a,b] . Si procede dividendo l'intervallo in due parti uguali e calcolando il valore della funzione nel punto medio di ascissa $\left(\frac{a+b}{2}\right)$.

Se risulta $f\left(\frac{a+b}{2}\right) = 0$ allora $\left(\frac{a+b}{2}\right)$ è la radice cercata; altrimenti tra i due intervalli $[a, \left(\frac{a+b}{2}\right)]$ e $[\left(\frac{a+b}{2}\right), b]$ si sceglie quello ai cui estremi la funzione assume valori di segno opposto. Si ripete per questo intervallo il procedimento di dimezzamento.

Per il calcolo dei pesi è utilizzata la *Simpson's 1/3 Rule* , una tecnica numerica utile per trovare l'integrale definito di una funzione dato un intervallo. La funzione è divisa in tanti sotto intervalli e ogni intervallo è approssimato dalla funzione integranda mediante archi di parabola, cioè mediante polinomi quadratici.

$$\int_a^b f(x)dx \approx \frac{h}{3}(f(a)+4f(a+h)+2f(a+2h)+4f(a+3h)+2f(a+4h)+....+f(b))$$

$$\Longrightarrow \int_a^b f(x)dx \approx \frac{h}{3}(f(a)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\ldots+f(b))$$

Dove x_i = a+ih con i=1,2,....n-1 e h=(b-a)/n

3.ESEMPIO NUMERICO

Per comprendere meglio come lavora l'algoritmo, procediamo con un esempio. Supponiamo di voler calcolare il seguente integrale

$$I = \int_1^4 cos(x) - xe^x \; dx \;\; , a = 1 \; b = 4$$

Risolvendo l'integrale definito si ottiene

$$I = \; -3e^4 - \sin(1) + \sin(4) \approx \; -165.39$$

Supponiamo di avere a disposizione:

- *Processi* $p = 4$
- *Nodi* $n = 10$

L'algoritmo, presi i dati p,n,a,b in input, procede a suddividere in base al rank del processo l'intervallo [-1,1] *(questo intervallo è dato dagli estremi di integrazione dell'integrale utile per calcolare le radici del polinomio di Legendre* $\int_{-1}^{1} L_i(x)L_m(x) = 0 \; se \; i \neq m)$. In questo caso ad ogni processo sarà assegnato $1/4$ dell'intero intervallo, dove calcolare le radici con il metodo della bisezione.

P0 P1 P2 P3

-1	.....	-0.5	.....	0	.....	0,5	1

Ogni singolo processo, subito dopo aver trovato una radice, la invia al processo master P0 con una MPI_Send, questa verrà ricevuta dalla MPI_Recv chiamata precedentemente. Tutte le radici ricevute verranno salvate all'interno di un array xi[ ].

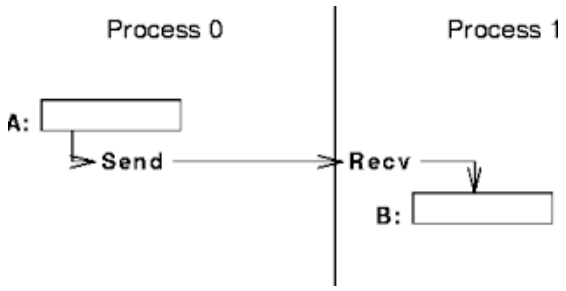

Figura 3 MPI_Send - MPI_Recv

Le **radici** del polinomio di Legendre per $n = 10$ sono

i	**root - x_i**
1	-0.148874
2	0.148874

i	root - x_i
3	-0.433395
4	0.433395
5	-0.679409
6	0.679409
7	-0.865063
8	0.865063
9	-0.973906
10	0.973906

Trovate le radici,P0 ordina con QUICKSORT i risultati in modo da poter abbinare correttamente ad ogni radice x_i il corrispettivo peso w_i

i	root - x_i
1	-0. 973906
2	-0. 865063
3	-0. 679409
4	-0. 433395
5	-0. 148874
6	0. 148874
7	0. 433395
8	0. 679409
9	0. 865063
10	0.973906

Array 1 - Radici ordinate da P0, ricevuto in Broadcast da P1,P2,P3

e l'invia a tutti i processi con una MPI_Broadcast.

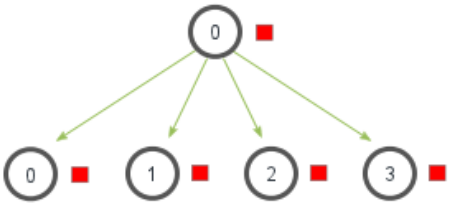

Figura 4- MPI_BROADCAST

Ora che tutti i processi conoscono l'array x_i è possibile calcolare i w_i. L'algoritmo, assegna nuovamente in base al rank dei processi una parte dell'array x_i

i	root - x_i	
1	-0. 973906	P0
2	-0. 865063	
3	-0. 679409	P1
4	-0. 433395	
5	-0. 148874	P2
6	0. 148874	

i	**root - x_i**	
7	0. 433395	P3
8	0. 679409	
9	0. 865063	
10	0.973906	
	RESIDUO	P3

Ogni singolo processo, subito dopo aver trovato il peso, calcola il contributo parziale dato da

$$(w_i) * f(\left(\frac{b-a}{2}\right) * x_i + \left(\frac{b+a}{2}\right)) \text{ con a=1 e b=4}$$

I **pesi** w_i del polinomio di Legendre per $n = 10$ sono

i	**pesi - w_i**	
1	0.066671	P0
2	0.149451	P0
3	0.219086	P1
4	0.269266	P1
5	0.295524	P2
6	0.295524	P2
7	0.269266	P3
8	0.219086	P3
9	0.149451	
10	0.066671	
	RESIDUO	P3

mentre **f**$(\left(\frac{b-a}{2}\right) * x_i + \left(\frac{b+a}{2}\right))$

con i=1 e $f(x) = cos(x) - xe^x$

⇨ f(1,5*[(-0. 973906)+2,5])= f(1.039141)= -2.430465

Per i=1,..10 avremo

i	f$(\left(\frac{b-a}{2}\right) * x_i + \left(\frac{b+a}{2}\right))$
1	-2.430465
2	-3.641623
3	-6.421424
4	-12.039480
5	-22.833584
6	-42.391659
7	- 74.517586
8	-119.715622
9	-170.141663
10	-208.637589

Calcoliamo il risultato:

$$I \cong \left(\frac{b-a}{2}\right) \sum_{i=1}^{n} (w_i f(x_i)) = 1.5[\ 0.066671(-2.430465) + 0.149451(-3.641623) +$$ — P0

$$0.219086(-6.421424) + 0.269266(-12.039480) +$$ — P1

$$0.295524(-22.833584) + 0.295524(-42.391659) +$$ — P2

$$0.269266(-74.517586) + 0.219086(-119.715622) +$$ — P3

$$0.149451(-170.141663) + 0.066671(-208.637589)] =$$ — RESIDUO P3

$$-165.39$$ — P0

Per sommare tutti i singoli contributi dei processi, viene eseguita una MPI_Reduce(MPI_SUM).

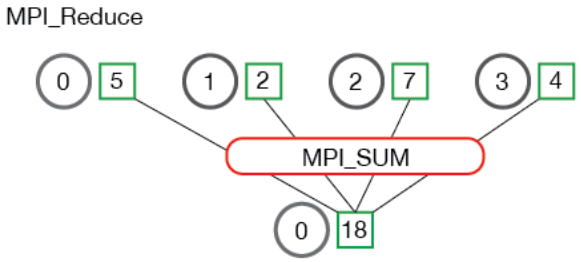

Figura 5-ESEMPIO MPI_REDUCE

Come possiamo notare il risultato dell'algoritmo è corretto.

4. CODICE

In questo capitolo sarà analizzato il codice C usato per implementare l'algoritmo.

Per il calcolo parallelo è utilizzata la libreria "mpich", un implementazione open source di MPI "Message Passing Interface"

Con mpich possiamo compilare programmi in C usando il comando mpicc:

```
mpicc -O3 –o myprog  myprog.c
```

ed eseguirli con mpirun

```
mpirun -n  p  ./ myprog
```

Per compilare il codice del progetto si utilizza

```
mpicc -O3 –o parallel_gaussian_quadrature parallel_gaussian_quadrature.c -lm
```

dove -lm fa riferimento alla liberia <math.h> inclusa nel codice

Per eseguirlo

```
mpirun -n p  ./ myprog n a b
```

Dove p è il numero di processi, n il numero di nodi, a e b gli estremi di integrazione della function(float x)

```
6   //funzione da integrare
7   float function(float x){
8       float y;
9       y = cos(x) - x * exp(x);
10      return y;
11  }
```

Inizia l'analisi da int main (int argc, char** argv)

DA RIGA 102 A 128 dichiarazione variabili

```
      //Variabile MPI necessaria per salvare lo stato della MPI_RECV

      MPI_Status status;

      //Variabile MPI necessaria  per salvare lo stato della MPI_IRECV e
controllo MPI_WAIT

      MPI_Request req;

      //utile per calcolare il tempo d'esecuzione
```

```
	double time;
	//varibili per MPI size e rank
	int size, rank;
	//buffer per send e recv
	double buff_send,buff_recv,buff_recv1;
	//estremi di integrazione per il polinomio di legendre
	int la = -1, lb = 1;
	//estremi di integrazione della function ricercata
	double a, b, temp;
	//variabili per salvare (b-a)/2 e (b+a)/2
	 double d, e;
	//Utili per differenziare in base al rank l'intervallo di lavoro per
processso
	double my_a, my_b;
	int i=0,n;
	//per salvare risultati parziali e finali dei singoli processi
	double my_res = 0,result;
	//variabile utile per cicli for
	int y = 0 ;
	//window(Step-size) per bisection method
	double h=0.00001;
	//variable utile per bisection method
	double x;
	//variabile dove viene ritornata la radice dopo il bisection loop
	double root;
```

Tutti i programmi MPI incominciano con la chiamata della funzione MPI_INIT()

```
130     MPI_Init(&argc, &argv);
131
132     MPI_Comm_size(MPI_COMM_WORLD, &size);
133     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

e terminano con la chiamata MPI_Finalize()

```
275     }
276     MPI_Finalize();
277 }
```

Communicator (COMM) è un gruppo di processi che può comunicare.

I comunicatori non sono partizioni dell'insieme dei processi (pertanto un processo può appartenere a più comunicatori), MPI_COMM_WORLD (=tutti i processi) è il comunicatore predefinito.

Il rank è un Identificatore numerico (da 0 a N) di un processo all'interno di un comunicatore, un processo può avere un diverso rank per ogni comunicatore cui appartiene MPI_COMM_RANK(comm, &rank)

Per conoscere il numero di processi all'interno di un comunicatore uso MPI_COMM_SIZE(comm, &size)

Il codice prosegue controllando se input da terminale è corretto e assegnare alla variabili dichiarate precedentemente i valori corrispondenti

```
136     if (argc != 4 )
137     {
138     if (rank == 0)
139     {
140
141         printf("Specificare i data point <n> e gli estremi di integrazione <a> <b>\n");
142         return 1;
143     }
144     }
145
146
147     n = strtof(argv[1], NULL);
148     a = strtof(argv[2], NULL);
149     b = strtof(argv[3], NULL);
```

Per calcolare il tempo di esecuzione, invoco una

```
152     MPI_Barrier(MPI_COMM_WORLD);
153     time = -MPI_Wtime();
```

Assegno per comodità alle variabili "d", "e" il valore di (b-a)/2 e (b+a)/2, eseguendo prima un controllo per verificare b>a

```
155     //se il primo estremo di integrazione è minore del secondo, scambio le variabili
156     if (a>b)
157     {
158         temp = a;
159         a = b;
160         b = temp;
161     }
162
163     d=((b-a)/2);
164     e=((b+a)/2);
```

Ora che conosco n posso dichiarare l'array per salvare le radici del Legendre polynomials

```
167     //Array per salvare le radici del Legendre polynomials
168     double xi[n];
```

L'algoritmo, presi i dati p,n,a,b in input, procede a suddividere in base al rank del processo l'intervallo [-1,1] *(questo intervallo è dato dagli estremi di integrazione dell'integrale utile per calcolare le radici del polinomio di Legendre* $\int_{-1}^{1} L_i(x)L_m(x) = 0$ *se* $i \neq m$*).*

```
175     double w = (lb - la)/h;
176
177     my_a = (rank) * (w/(size));
178     my_b = my_a + (w/(size));
179
180     double my_a_n =my_a*h -1;
181     double my_b_n =my_b*h -1;
```

Inizia la ricerca delle radici, ogni processo cercherà nel proprio intervallo my_a_n, my_b_n, aumentando my_a_n ad ogni iterazioni di h= window(Step-size) per bisection method

```
183     for (x = my_a_n; x < my_b_n; x += h)
184     {
185         root=bisection(n,Pn,x,x+h,0.0000001,1000000);
186         //printf("Trovata\n");
187             if(root!=999)
188             {
189                 if (rank == 0)
190                  {
191                 MPI_Irecv(&buff_recv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0 , MPI_COMM_WORLD,&req);
192              }
193
194                     buff_send = root;
195                     MPI_Send(&buff_send, 1, MPI_DOUBLE, 0 ,0, MPI_COMM_WORLD);
196
197             if (rank == 0)
198                  {
199                 MPI_Wait(&req, MPI_STATUS_IGNORE);
200                     xi[i]=buff_recv;
201                  }
202
203                  i++;
204              }
205     }
```

Se il return della funzione bisection è diverso da 999, root è la nostra radice e sarà inviata con una MPI_Send. Il processo master P0, riceverà questa radice tramite una MPI_Irecv, chiamata non bloccante che ritorna "immediatamente". Il buffer del messaggio non è letto subito dopo il ritorno al processo chiamante, ma si controlla che la RECEIVE sia completata, tramite una MPI_Wait. Completata la RECEIVE, salviamo il risultato nell'array xi[i]

MPI_Send*(void *data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm)*

Input Parameters

data : indirizzo del buffuer di invio

count: numero di elementi nel buf di invio

datatype: tipo di dato di ogni elemento del buf di invio

destination: rank del destinatario del messaggio

tag: message tag

comm: comunicatore

MPI_Irecv*(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request)*

Input Parameters

buf : indirizzo di ricezione del buffuer

count: numero di elementi nel buf di ricezione

datatype: tipo di dato di ogni elemento del buf di ricezione

source: rank di chi ha inviato il messaggio

tag: message tag

comm: comunicatore

Output Parameters

request: communication request (handle)

MPI_Wait*(MPI_Request *request, MPI_Status *status)*

Input Parameters

Request: request

Output Parameters

Status: status object.

```
207     //ricevo le n-i restanti radici
208     if (rank == 0)
209     {
210         for(y = i ; y < n; y++ )
211         {
212             MPI_Recv(&buff_recv1, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0 , MPI_COMM_WORLD,&status);
213             xi[y]=buff_recv1;
214     }
```

Ricevo le restanti n-i radici con una MPI_RECV

MPI_Recv*(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)*

Input Parameters

count: numero di elementi nel buf di ricezione

datatype: tipo di dato di ogni elemento del buf di ricezione

source: rank di chi ha inviato il messaggio

tag: message tag

comm: comunicatore

Output Parameters

buf : indirizzo di ricezione del buffuer

status: status object (Status)

Ricevute da P0 tutte le radici,le ordina tramite qsort e stampa a video i risultati

```
215         //ordino array
216            qsort (xi, n, sizeof(double), cmpfunc);
217            printf("\nOrdino l'array\n\n");
218         for( int y = 0 ; y < n; y++ )
219            {
220                printf(" x[%d] = %f\n",y+1, xi[y]);
221         }
```

eseguendo una MPI_BROADCAST(ogni processo ha bisogno di tutte le radici per calcolare i relativi pesi)

```
226      //Invio a tutti i processi l'array ordinato
227         MPI_Bcast(xi, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

MPI_Bcast*(void *data, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

Input/Output Parameters

data : indirizzo del buffuer di invio

Input Parameters

count: numero di elementi nel buf di invio

datatype: tipo di dato di ogni elemento del buf di invio

root: rank del processo master che invia

comm: comunicatore

Ora che tutti i processi conoscono l'array xi[i] è possibile calcolare i C[i]. Il codice assegna nuovamente in base al rank dei processi una parte dell'array xi[i] .

```
229     my_a = (rank) * (n/(size));
230     my_b = my_a + (n/(size));
231
232     for(i=my_a;i<my_b;i++)
233     {
234         double my_root = xi[i];
235         double my_peso = Ci(i,n,xi,-1,1,100000);
236         printf("%d) Peso c[%d] = %7.6lf\n",rank,i+1,my_peso);
237         double t = my_peso*function((d*my_root) + e);
238         my_res += t;
239
240     }
```

Trovato il peso e conoscendo la rispettiva radice,il processo calcola il suo risultato parziale `my_res`.

Stessa procedura avviene per il residuo da my_b a n (sarà il processo(size-1) a effettuare questo calcolo)

```
241     //residuo
242     int rankn = size-1;
243
244     if(rank==rankn)
245     {
246         for(i=my_b;i<n;i++)
247         {
248             double my_root = xi[i];
249             double my_peso = Ci(i,n,xi,-1,1,100000);
250             printf("%d) Peso c[%d] = %7.6lf\n",rank,i+1,my_peso);
251             double t = my_peso*function((d*my_root) + e);
252
253             my_res += t;
254         }
255     }
```

Per sommare tutti i singoli contributi dei processi, utilizzo una MPI_Reduce(MPI_SUM)

```
256     //MPI_reduce(MPI_SUM) per sommare tutti i singoli contributi
257     MPI_Reduce(&my_res, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

MPI_Reduce*(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,MPI_Op op, int root, MPI_Comm comm)*

Input Parameters

sendbuf: indirizzo del buffuer di invio

count: numero di elementi nel buf di invio

datatype: tipo di dato di ogni elemento del buf di invio

op: operazione di reduce

root: rank del processo master

comm: comunicatore

Output Parameters

recvbuf : indirizzo del buffuer di ricezione

Ottenuto il risultato finale, P0 stamperà a video il risultato e il tempo di esecuzione

```
259     if (rank == 0){
260         result = d * result;
261     //  Arrotondo Risultato
262     //ceilif restituisce un valore a virgola mobile che rappresenta il numero
263     //intero più piccolo maggiore o uguale a x
264         float rounded_up = ceilf(result * 100) / 100;
265         printf("\n_____\n");
266         printf("\nRisultato Integrale: %.2f\n", rounded_up);
267         printf("_____\n\n");
268     }
269
270     //Tempo
271     MPI_Barrier(MPI_COMM_WORLD);
272     time += MPI_Wtime();
273     if (rank == 0)
274     {
275         printf("Wall clock time %f\n", time);
276     }
```

Per arrotondare il risultato utilizzo la funzione ceilf che restituisce un valore a virgola mobile che rappresenta il numero intero più piccolo maggiore o uguale a x.

4.1 Altre funzioni utilizzate

Funzione per definire l'integrazione con Simpson's 1/3rd Rule

$$C_i = \int_{-1}^{1} L_i(x)dx$$

```
53  /*Funzione per definire l'integrazione con Simpson's 1/3rd Rule */
54  long double Ci(int i, int n, double x[n], double a, double b, int N){
55    long double h,integral,X,sum=0;
56    int j,k;
57    h=(b-a)/N;
58    for(j=1;j<N;j++){
59      X=a+j*h;
60      if(j%2==0){
61        sum=sum+2*Li(n-1,x,i,X);
62      }
63      else{
64        sum=sum+4*Li(n-1,x,i,X);;
65      }
66    }
67      long double Fa=Li(n-1,x,i,a);;
68      long double Fb=Li(n-1,x,i,b);
69
70    integral=(h/3.0)*(Fa+Fb+sum);
71    return integral;
72  }
```

Funzione che calcola i termini di Lagrande di grado n

$$L_i(x) = \prod_{j=0; j\neq i}^{n} \left(\frac{x - x_j}{x_i - x_j} \right)$$

```c
/*Lagrange terms*/
long double Li(int n, double x[n+1], int i, double X){
    int j;
    long double prod=1;
    for(j=0;j<=n;j++){
        if (j!=i){
            prod=prod*(X-x[j])/(x[i]-x[j]);
        }
    }
    return prod;
}
```

Funzione metodo della bisezione[Return la radice quando la trova o 999 altrimenti]

```c
/*Funzione del metodo della bisezione[Return la radice quando la trova o 999 altrimenti]*/
double bisection(int n,double f(int n,double x),double a, double b, double eps, int maxSteps){
  double c;
  if(f(n,a)*f(n,b)<=0){
    int iter=1;

    do{
      c=(a+b)/2;
      if(f(n,a)*f(n,c)>0){
      a=c;
    }
    else if(f(n,a)*f(n,c)<0){
      b=c;
    }
    else if(f(n,c)==0){
       return c;
    }
      iter++;

    }while(fabs(a-b)>=eps&&iter<=maxSteps);
    return c;
  }
  else{
    return 999;
  }
}
```

Funzione che confronta i termini per ordinare array

```c
//funzione che confronta termini per ordinare array
int cmpfunc (const void * a, const void * b)
{
  if (*(double*)a > *(double*)b)
    return 1;
  else if (*(double*)a < *(double*)b)
    return -1;
  else
    return 0;
}
```

Funzione che calcola i polinomi di Legendre di grado n

$$(l+1)P_{l+1}(x) - (2l+1)xP_l(x) + lP_{l-1}(x) = 0$$

Caso base per i polinomi di grado 0 e 1.

$$P_0(x) = 1$$
$$P_1(x) = x$$

```
24  /*Nth Legendre Polynomial Pn(x)*/
25  double Pn( int n ,double x )
26  {
27      double r,s,t ;
28      int m;
29      r=0; s=1;
30
31      for (m=0; m<n; m++)
32      {
33          t=r;
34          r=s;
35          s=(2*m+1 )*x*r-m*t;
36          s/=(m+1);
37      }
38      return s;
39  }
```

5. SPEEDUP & EFFICIENZA

Il codice parallelo è stato testato su una macchina con 4 GB di RAM e Processore Intel® Core™ i3-6006U

Di seguito è riportata una tabella che mostra i risultati del codice parallelo.

TEMPO DI ESECUZIONE

Input size	1 process	4 process
10	0.68721 [s]	0.39830 [s]
100	9.513868 [s]	2.892502 [s]
200	40.112161 [s]	11.026709 [s]
500	258.462845 [s]	69.596909 [s]
1000	1045.246108 [s]	268.365330 [s]