Dipartimento di Matematica e Informatica

Laurea Magistrale in Informatica



Corso di

Programmazione Concorrente e Parallela\ Metodi teorici e computazionali per le scienze molecolari

Progetto:

***“Implementazione Parallela del metodo di jacobi per il calcolo di autovalori e autovettori”***

Studenti

Luca Pagliochini

Matteo Riganelli

Prof.

Leonardo Pacifici

Antonio Laganà

Sergio Tasso

a.a. 2015/2016



1 Presentazione del Problema

Il metodo di Jacobi consiste in una sequenza di trasformazioni su una matrice di reali simmetrica. Le operazioni che vengono effettuate sulla matrice sono denominate “Jacobi rotation”. Ogni rotazione è destinata ad “annichilire” uno degli elementi della matrice fuori diagonale. Una sequenza di queste rotazioni tende a far diventare gli elementi fuori la diagonale sempre più piccoli, fino a quando la matrice data, non diventa effettivamente una matrice triangolaresuperiore simmetrica.

Accumulare il prodotto delle trasformazioni restituirà la matrice degli autovettori, mentre gli elementi della matrice diagonale sono gli autovalori. Il metodo di Jacobi è assolutamente infallibile per tutte le matrici simmetriche reali.

Più precisamente, la matrice data in input viene denominata con a[1...n][1...n]. In output, la superdiagonale (l’insieme degli elementi appena sopra gli elementi della diagonale) sono distrutti, ma la diagonale e la subdiagonale (insieme degli elementi al di sotto della diagonale ) sono invariati e restituiscono tutta l’informazione relativa alla matrice originale simmetrica ‘a’.

Il vettore d[1...n] restituisce gli autovalori di a. Durante la computazione, ‘d’ contiene la diagonale corrente di ‘a’. La matrice v[1...n][1...n] restituisce gli autovettori normalizzati corrispondenti al k-esimo elemento della matrice ‘v’.

Il parametro nrot rappresenta il numero di rotazioni di Jacobi che sono necessarie per ottenere la convergenza.

2 Algoritmo Seriale

L’algoritmo sequenziale nella parte iniziale istanzia alcune variabili necessarie al funzionamento dell’algoritmo per poi inizializzare la matrice v[1...n][1...n] come matrice identità. Inizializza il vettore ‘b’ e ‘d’ uguale alla diagonale di ‘a’, inizializzando anche il vettore z[1...n] a 0, il quale accumulerà i valori calcolati secondo una specifica equazione.

Dopodiché il metodo di jacobi viene effettivamente racchiuso all’interno di un ciclo for che lo esegue al massimo un numero prestabilito di volte, in questo caso il metodo viene eseguito al massimo per 50 iterazioni, migliorando di passo in passo l’approssimazione dei risultati.

Il cuore dell’algoritmo consiste nell’effettuare delle operazioni di “rotazione”, racchiuse in quattro cicli for distinti, che vanno a modificare i primi tre la matrice ‘a’ (per il calcolo degli autovalori) e l’ultimo la matrice ‘v’ (per il calcolo degli autovettori)

|  |  |
| --- | --- |
| Matrice | Tempo Esecuzione (s) |
| 500 | 14,203533 |
| 700 | 58,5994 |
| 1000 | 1233,367891 |
| 1200 | 2116,1571 |
| 1700 | 5264,408303 |

3 Algoritmo Parallelo

Abbiamo deciso di analizzare l’algoritmo per verificare dove fosse possibile effettuare la parallelizzazione. La prima parte è solamente una fase di inizializzazione dell’algoritmo e quindi è stata tralasciata. La parte più consistente è quella relativa ai due cicli for che scorrono gli indici ip e iq. Tali cicli però effettuano operazioni su stessi elementi in successive iterazioni, perciò non sono parallelizzabili, come è possibile vedere nel seguente esempio:

Esempio:

|  |
| --- |
| int n=6;  float a = matrix(1,n, 1,n);  float v = matrix(1,n, 1,n);  int ip, iq;  (1°iterazione)  ip=1; ip<=n-1; ip++  iq=ip+1; iq<=n; iq++ **ip=1; iq=2**  for j=1 a ip-1 ROTATE(a,j,ip,j,iq)  for j=ip+1 a iq-1 ROTATE(a,ip,j,j,iq)  for j = iq+1 a n ROTATE(a,ip,j,iq,j)  a[1][3], a[2][3],  a[1][4], a[2][4],  a[1][5], a[2][5],  a[1][6], a[2][6]  for j =1 a n ROTATE(v,j,ip,j,iq)  v[1][1], v[1][2],  v[2][1], v[2][2],  v[3][1], v[3][2],  v[4][1], v[4][2],  v[5][1], v[5][2],  v[6][1], v[6][2]  (2°iterazione)  ip=1; ip<=n-1; ip++  iq=ip+1; iq<=n; iq++ **ip=1; iq=3**  for j=1 a ip-1 ROTATE(a,j,ip,j,iq)  for j=ip+1 a iq-1 ROTATE(a,ip,j,j,iq)  a[1][2], a[2][3]  ... |

**Figura 1**. Esempio iterazioni algoritmo

Dall’esempio sopra è possibile evidenziare come i due cicli for finiscano con il modificare stessi elementi in successive iterazione e ciò significa che se le facessimo in parallelo non otterremmo gli stessi risultati con le rotate.

Abbiamo dunque deciso di applicare la parallelizzazione a livello delle operazioni di rotate, in corrispondenza dei quattro cicli for, in quanto ad ogni operazione le rotate modificano elementi differenti e quindi risultano parallelizzabili.

Abbiamo deciso di studiare vari approcci alla parallelizzazione che andiamo ora a presentare con relativi risultati.

Per questi algoritmi e per i calcoli dello speed up, sono state eseguite diverse serie di prove, variando rispettivamente la dimensione della matrice e il numero di processori utilizzati per i calcoli.

3.1 versione 1 (file jacobi1.c)

Come primo tentativo abbiamo deciso di parallelizzare tutti e quattro i cicli for che effettuano le operazioni di rotate, facendo anche uso dell’allocazione dinamica del carico di lavoro, poiché il valore del range dei cicli for era variabile per ogni iterazione dei cicli for superiore (ip, iq).

Tale scelta non si è rivelata efficiente a causa dell’elevato numero di operazioni di Send e Receive.

Le quattro rotate si comportano tutte allo stesso modo, andando ad utilizzare i processori in maniera dinamica nel caso in cui il numero di iterazioni sia superiore al numero di processori disponibili.

Vengono quindi inviati agli slave i dati necessari per il calcolo delle rotate, racchiusi all’interno di un vettore ‘dati[10]’, il quale poi verrà rimandato a sua volta dagli slave con i valori calcolati che verranno cosi aggiornati dal master.

La difficoltà nell’approccio sta nel gestire in maniera corretta gli indici in modo da allocare dinamicamente il lavoro e gestire correttamente i calcoli.

|  |
| --- |
| 1. int indice=1; 2. if( ((iq-1)-(ip+1) +1) > (size-1) ) { //j=ip+1;j<=iq-1;j++; NUM\_ITERAZIONI 3. indice=ip+1; 4. for (j=1;j<size;j++) { 5. dati[0] = indice; 6. indice++; 7. dati[8] = a[ip][(int)dati[0]]; 8. dati[9] = a[(int)dati[0]][iq]; 9. MPI\_Isend(&dati,10,MPI\_FLOAT,j,1, MPI\_COMM\_WORLD, &request4); 10. } 11. int contatore=indice; 12. while (contatore<=iq-1) { 13. MPI\_Recv(&dati,10, MPI\_FLOAT,MPI\_ANY\_SOURCE,2,MPI\_COMM\_WORLD,&status); 14. int invia = status.MPI\_SOURCE; 15. a[(int)dati[1]][(int)dati[0]]=dati[6]; 16. a[(int)dati[0]][(int)dati[2]]=dati[7]; 17. dati[0] = contatore; 18. dati[8] = a[ip][(int)dati[0]]; 19. dati[9] = a[(int)dati[0]][iq]; 20. MPI\_Send(&dati,10,MPI\_FLOAT,invia,1, MPI\_COMM\_WORLD); 21. contatore++; 22. } 23. contatore=1; 24. while(contatore<size){ 25. MPI\_Recv(&dati,10, MPI\_FLOAT,MPI\_ANY\_SOURCE,2,MPI\_COMM\_WORLD,&status); 26. a[(int)dati[1]][(int)dati[0]]=dati[6]; 27. a[(int)dati[0]][(int)dati[2]]=dati[7]; 28. contatore++; 29. } 30. } 31. else{ 32. int contatore=1; 33. for (j=ip+1;j<=iq-1;j++) { 34. dati[0] =j; 35. dati[8] = a[ip][(int)dati[0]]; 36. dati[9] = a[(int)dati[0]][iq]; 37. MPI\_Isend(&dati,10,MPI\_FLOAT,contatore,1, MPI\_COMM\_WORLD, &request4); 38. MPI\_Recv(&dati,10, MPI\_FLOAT,contatore,2,MPI\_COMM\_WORLD,&status); 39. a[(int)dati[1]][(int)dati[0]]=dati[6]; 40. a[(int)dati[0]][(int)dati[2]]=dati[7]; 41. contatore++; 42. //ROTATE(a,ip,j,j,iq) 43. } 44. } |

**Figura 2**.Esempio di parallelizzazione secondo ciclo for dell’algoritmo

Gli altri cicli for effettuano in maniera simile le stesse operazioni, con alcune differenze negli indici utilizzati per effettuare le operazioni e le comunicazioni con gli slave.

3.1.1 Valutazione e Misurazione delle Prestazione

La tabella seguente riporta i dati raccolti circa il tempo di esecuzione dell’algoritmo.

|  |  |  |
| --- | --- | --- |
| Matrice | Num.Processori | Tempo Esecuzione (s) |
| 20 x 20 | 5 | 1,624494 |
| 20 x 20 | 6 | 1,5007405 |
| 20 x 20 | 7 | 1,498454 |
| 20 x 20 | 8 | 1,446221 |
| 50 x 50 | 5 | 25,563046 |
| 50 x 50 | 6 | 21,783693 |
| 50 x 50 | 7 | 19,531824 |
| 50 x 50 | 8 | 17,830466 |
| 100 x 100 | 8 | 69,655001 |

I tempi ovviamente risultano essere molto maggiori rispetto alla versione seriale poiché l’algoritmo fa un elevato uso di Send e receive per ogni rotate e quindi il prezzo delle comunicazioni è molto maggiore del guadagno che si ottiene a distribuire le rotate. Inoltre la rotate che viene eseguita il maggior numero di volte e in tutte le iterazioni è quella relativa al calcolo degli autovettori, perciò abbiamo deciso di lasciare seriale il calcolo degli autovalori e parallelizzare solo il calcolo degli autovettori.

3.2 versione 2 (file jacobi2.c)

Come secondo tentativo abbiamo deciso di parallelizzare solo l’ultimo ciclo for, quello che modifica la matrice ‘v’, in quanto tale ciclo, ad ogni iterazione del indice ‘iq’ viene ripetuto da 1 ad n, mentre gli altri cicli fanno un numero molto minore iterazioni e in alcuni casi non vengono neppure eseguite.

|  |
| --- |
| 1. if(n > size-1){ 2. for (j=1;j<size;j++) { 3. dati[0] =j; 4. dati[8] = v[(int)dati[0]][ip]; 5. dati[9] = v[(int)dati[0]][iq]; 6. MPI\_Send(&dati,10,MPI\_FLOAT,j,1, MPI\_COMM\_WORLD); 7. } 8. int contatore=(size); 9. while (contatore<=n) { 10. MPI\_Recv(&dati,10, MPI\_FLOAT,MPI\_ANY\_SOURCE,2,MPI\_COMM\_WORLD,&status); 11. int invia = status.MPI\_SOURCE; 12. v[(int)dati[0]][(int)dati[1]]=dati[6]; 13. v[(int)dati[0]][(int)dati[2]]=dati[7]; 14. dati[0] = contatore; 15. dati[8] = v[(int)dati[0]][ip]; 16. dati[9] = v[(int)dati[0]][iq]; 17. MPI\_Send(&dati,10,MPI\_FLOAT,invia,1, MPI\_COMM\_WORLD); 18. contatore++; 19. } 20. contatore=1; 21. while(contatore<size){ 22. MPI\_Recv(&dati,10, MPI\_FLOAT,MPI\_ANY\_SOURCE,2,MPI\_COMM\_WORLD,&status); 23. v[(int)dati[0]][(int)dati[1]]=dati[6]; 24. v[(int)dati[0]][(int)dati[2]]=dati[7]; 25. contatore++; 26. } 27. } 28. else{ 29. for (j=1;j<=n;j++) { 30. dati[0] = j; 31. dati[8] = v[(int)dati[0]][ip]; 32. dati[9] = v[(int)dati[0]][iq]; 33. MPI\_Send(&dati,10,MPI\_FLOAT,j,1, MPI\_COMM\_WORLD); 34. MPI\_Recv(&dati,10, MPI\_FLOAT,j,2,MPI\_COMM\_WORLD,&status); 35. v[(int)dati[0]][(int)dati[1]]=dati[6]; 36. v[(int)dati[0]][(int)dati[2]]=dati[7]; 37. } 38. } |

**Figura 3**.Esempio di parallelizzazione quarto ciclo for dell’algoritmo

3.2.1 Valutazione e Misurazione delle Prestazione

La tabella seguente riporta i dati raccolti circa il tempo di esecuzione.

|  |  |  |
| --- | --- | --- |
| Matrice | Num Processori | Tempo Esecuzione (s) |
| 20 x 20 | 5 | 0,730178 |
| 20 x 20 | 6 | 0,61245 |
| 20 x 20 | 7 | 0,567031 |
| 20 x 20 | 8 | 0,443899 |
| 50 x 50 | 5 | 12,431188 |
| 50 x 50 | 6 | 10,249582 |
| 50 x 50 | 7 | 8,746173 |
| 50 x 50 | 8 | 7,746388 |
| 100 x 100 | 4 | 3,140 min |

Questa versione mostra dei miglioramenti nei tempi di esecuzione rispetto alla versione 1, ma non ancora da potersi ritenere soddisfacente, a causa sempre dal tasso elevato di comunicazioni, dovuto all’uso di Send e Receive.

3.3 versione 3 (file jacobi3.c)

Dato il rallentamento dovuto all’elevato numero di operazioni di Send e Receive che venivano effettuate, abbiamo mantenuto l’idea di parallelizzare l’ultimo ciclo for essendo il più oneroso e utile per la parallelizzazione, ma abbiamo deciso di utilizzare un approccio diverso e dunque si è scelto di far calcolare agli slave non una singola iterazione, ma un range, calcolato sulla base del valore di n(dimensione della matrice) e della size(num processori). A seconda di questi due valori, inizialmente, viene calcolato un valore intero denominato ‘chunk’:

int chunk = n/(size-1)

Tale valore permette di stabilire il range che ogni processore dovrà calcolare. Inoltre abbiamo deciso che ogni slave manterrà una matrice v ed opererà su tale matrice evitando cosi di doverla rimandare al master risparmiando cosi sulle Send di ritorno.

La funzione che realizza il funzionamento di tale versione è la seguente

|  |
| --- |
| 1. int chunk = n/(size-1) 2. int dispari = n % (size-1); 3. double indiceIni = 1.0; 4. double indiceFin = chunk; 5. float indiceVec[(size-1)\*2]; 6. int indice = 0; 7. for(j=1; j<size; j++){ 8. indiceVec[indice] = indiceIni; 9. indiceVec[indice+1] = indiceFin; 10. indiceIni = indiceVec[indice+1] + 1; 11. if(dispari == 0) indiceFin = indiceVec[indice+1] + chunk; 12. else{ 13. if(j == size -1) indiceFin = n; 14. else indiceFin = indiceVec[indice+1] + chunk; 15. } 16. indice = indice +2 ; 17. } |

**Figura 4**.funzione ripartizione range indici

Viene quindi associato un indice iniziale e un indice finale da inviare ad ogni processore, cosi che in base al chunk ogni processore effettuerà un certo numero di rotate.

Viene inoltre fatto un controllo sul numero di processori che si hanno a disposizione:

int dispari = n % (size-1);

Nel caso un cui il valore della variabile dispari sia diverso da zero, all’ultimo processore dovrò assegnare un range diverso dagli altri processori, in quanto arriverà fino ad n come numero di iterazioni. Tale controllo permette di utilizzare il qualsiasi numero di processori e dividere in maniera coerente il valore di n per numero di processori.

3.1.1 Valutazione e Misurazione delle Prestazione

La tabella seguente riporta i dati raccolti circa il tempo di esecuzione, lo speedUp e l’efficienza.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Matrice | Processori | Tempo | SpeedUp | Efficienza |
| 500 x 500 | 3 | 32,17385325 | 0,4414619812 | 0,0008829239625 |
| 500 x 500 | 5 | 39,206288 | 0,3622769133 | 0,0007245538267 |
| 500 x 500 | 6 | 48,1519955 | 0,2949728844 | 0,0005899457687 |
| 500 x 500 | 8 | 62,494858 | 0,2272752264 | 0,0004545504528 |
| 700 x 700 | 3 | 113,0999075 | 0,3246103804 | 0,0004637291149 |
| 700 x 700 | 5 | 197,993783 | 0,1854270545 | 0,0002648957922 |
| 700 x 700 | 6 | 256,4060397 | 0,1431846303 | 0,0002045494719 |
| 700 x 700 | 8 | 137,678136 | 0,4256260413 | 0,0006080372019 |
| 1000 x 1000 | 3 | 539,624875 | 2,285602366 | 0,002285602366 |
| 1000 x 1000 | 5 | 239,8267653 | 5,14274497 | 0,00514274497 |
| 1000 x 1000 | 6 | 285,248927 | 4,323830081 | 0,004323830081 |
| 1000 x 1000 | 8 | 336,492222 | 3,665368203 | 0,003665368203 |
| 1200 x 1200 | 3 | 259,49045 | 3,421407697 | 0,002851173081 |
| 1200 x 1200 | 5 | 365,6194385 | 2,428269751 | 0,002023558126 |
| 1200 x 1200 | 6 | 404,0381863 | 2,197373053 | 0,001831144211 |
| 1200 x 1200 | 8 | 517,0959587 | 1,716939783 | 0,001430783152 |
| 1700 x 1700 | 3 | 1324,477304 | 3,974706314 | 0,002338062538 |
| 1700 x 1700 | 5 | 1762,797318 | 2,986394551 | 0,001756702677 |
| 1700 x 1700 | 6 | 1857,35959 | 2,834350619 | 0,00166726507 |
| 1700 x 1700 | 8 | 2254,571063 | 2,33499329 | 0,001373525464 |

Il tempo di esecuzione è notevolmente ridotto rispetto alle precedenti versione e l’algoritmo inoltre può operare su matrici molto più grandi però in alcuni casi risulta essere più lento rispetto alla versione seriale e un incremento del numero di processori provoca un calo rapido delle prestazione. Abbiamo dunque operato un ulteriore modifica relativa al modo in cui il master comunica con gli slave.

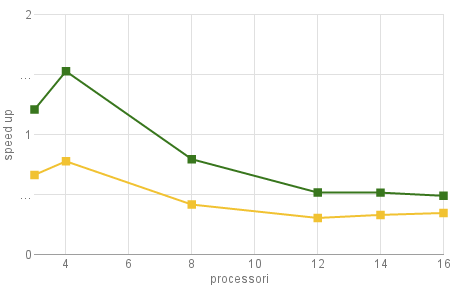
3.4 versione 3.1 (file jacobi4.c)

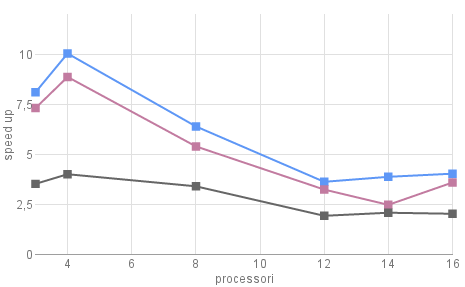
Rispetto alla versione 3 si è deciso di utilizzare per l’invio dei dati dal master agli slave la funzione MPI\_Bcast, poiché secondo quanto riportato dalla documentazione all’aumentare dei processi la funzione invia i dati a tutti in maniera più efficiente e veloce.

Vediamo i tempi di esecuzione

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Matrice | Processori | Tempo | SpeedUp | Efficienza |
| 500 x 500 | 3 | 21,41565 | 0,663231 | 0,001326462937 |
| 500 x 500 | 4 | 18,2885 | 0,77663 | 0,00155327479 |
| 500 x 500 | 8 | 34,13635 | 0,416082358 | 0,0008321647159 |
| 500 x 500 | 12 | 46,6388 | 0,3045432773 | 0,0006090865545 |
| 500 x 500 | 14 | 43,04365 | 0,3299797531 | 0,0006599595062 |
| 500 x 500 | 16 | 41,06755 | 0,3458578123 | 0,0006917156246 |
| 700 x 700 | 3 | 48,515 | 1,207861486 | 0,001725516409 |
| 700 x 700 | 4 | 38,37855 | 1,526878947 | 0,002181255638 |
| 700 x 700 | 8 | 73,81045 | 0,7939173925 | 0,001134167704 |
| 700 x 700 | 12 | 113,46565 | 0,5164505734 | 0,0007377865334 |
| 700 x 700 | 14 | 113,6502 | 0,5156119391 | 0,0007365884844 |
| 700 x 700 | 16 | 119,86995 | 0,4888581333 | 0,0006983687619 |
| 1000 x 1000 | 3 | 152,0599 | 8,11106604 | 0,00811106604 |
| 1000 x 1000 | 4 | 122,7225 | 10,05005513 | 0,01005005513 |
| 1000 x 1000 | 8 | 192,5750215 | 6,404609909 | 0,006404609909 |
| 1000 x 1000 | 12 | 339,365731 | 3,634332457 | 0,003634332457 |
| 1000 x 1000 | 14 | 317,5778395 | 3,883671143 | 0,003883671143 |
| 1000 x 1000 | 16 | 305,1927325 | 4,041275429 | 0,0006983687619 |
| 1200 x 1200 | 3 | 289,11335 | 7,319472103 | 0,006099560086 |
| 1200 x 1200 | 4 | 238,38185 | 8,877173745 | 0,007397644787 |
| 1200 x 1200 | 8 | 391,4627485 | 5,405768769 | 0,004504807307 |
| 1200 x 1200 | 12 | 651,7727395 | 3,246771415 | 0,002705642846 |
| 1200 x 1200 | 14 | 852,377418 | 2,482652702 | 0,002068877252 |
| 1200 x 1200 | 16 | 587,929291 | 3,599339465 | 0,002999449555 |
| 1700 x1700 | 3 | 1490,7501 | 3,531382157 | 0,002077283621 |
| 1700 x1700 | 4 | 1311,47025 | 4,014127124 | 0,002361251249 |
| 1700 x1700 | 8 | 1543,92465 | 3,409757272 | 0,002005739572 |
| 1700 x1700 | 12 | 2720,986786 | 1,934742326 | 0,001138083721 |
| 1700 x1700 | 14 | 2520,679935 | 2,088487408 | 0,001228522005 |
| 1700 x1700 | 16 | 2585,227256 | 2,036342566 | 0,001197848568 |

Dai grafici registrati in alcuni casi risulta un miglioramento dei tempi di esecuzione mentre in altri si è registrato un lieve decremento. Non è stato possibile stabilire se ciò sia o meno dovuto all’uso di MPI\_Bcast, poiché spesso i tempi registrati sono soggetti a fluttuazioni dovuti presumibilmente al carico di lavoro del cluster.





Legenda

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 500 x 500 | 700 x 700 | 1000 x 1000 | 1200 x 1200 | 1700 x 1700 |

3.2 Terminazione

La terminazione per tutte e tre le versioni viene effettuata dal master, una volta terminati tutti i cicli di iterazione, inviando un valore specifico (in questo caso -1) agli slave, alla ricezione di tale valore gli slave effettueranno break, terminando il loro lavoro e visualizzando l’output.

4 Conclusioni

In questo progetto sono state studiate varie tecniche per cercare di parallelizzare e migliorare i tempi di computazione dell’algoritmo di Jacobi. Sperimentalmente abbiamo constatato che l’algoritmo seriale risulta essere molto performante, consentendo tempi piuttosto soddisfacenti anche se,per matrici più grandi, si cominciano ad avere dei rallentamenti più consistenti.

Negli studi da noi effettuati, ci siamo concentrati a livello delle operazioni di rotate essendo questo l’unico punto per noi dove si potrebbero effettuare operazioni sensate di parallelizzazione. Per quanto riguarda la prima e la seconda versione di parallelizzazione il criterio adottato non è efficiente poiché vengono eseguite operazioni di Send e Receive in un numero proporzionale alla grandezza della matrice della quale si vogliono ottenere gli autovalori e autovettori ed essendo eseguite tali operazioni per tutte e quattro le rotate, il costo di comunicazione diventa molto alto.

La terza versione risulta invece essere migliore, ottenendo risultati più accettabili. Apportando la suddivisione del carico di lavoro per range dinamici, calcolati sulla base del valore del numero di processori, abbiamo ottenuto anche casi in cui si raggiunge la risoluzione del metodo con la metà del tempo che impiega l’algoritmo seriale (esempio caso 1000 x 1000).

Implementando la comunicazione con MPI\_Bcast abbiamo ottenuto un abbassamento del ritardo di comunicazione ottenendo tempi migliori.

In generale l’algoritmo parallelo non migliora in termini di ordine di grandezza il tempo di esecuzione dell’algoritmo seriale ma ciò è dovuto a vari fattori: l’algoritmo seriale è molto performante ed effettua operazioni comunque non costose e quindi parallelizzando solo alcuni punti come nel nostro caso è molto più probabile che si ottengano dei lievi peggioramenti piuttosto che miglioramenti ma bisogna tenere in considerazione anche del fatto che i due cicli for relativi all’iterazione degli indici ip q iq (che sono quelli più onerosi) non possono essere parallelizzati come abbiamo visto nella parte iniziale della relazione.