Calcolo distribuito dell'algoritmo di Jacobi

Christian Barbato `771459` `crbarbat@dsi.unive.it`

8 ottobre 2003

Sommario

Creazione di un programma parallelo per il calcolo distribuito di una applicazione dell'algoritmo iterativo di Jacobi. In particolare si utilizza Jacobi per modellare l'equazione differenziale di Laplace per il calcolo ai limiti della temperatura di una piastra metallica.

1 Introduzione

Molte applicazione scentifiche si riconducono allo schema detto five point stencil, ossia il calcolo di un valore è dato dalla somma dei 4 valori adiacenti ad esso (a nord, sud, est e ovest).

Tra le varie applicazioni possiamo trovare l'esecuzione dell'algoritmo di Jacobi per risolvere un sistema di n equazioni lineari in n incognite. Nell'applicazione che andrò a dimostrare lo si utilizzerà per risolvere l'equazione di Laplace adatta per esempio a modellare il comportamento termico di una lastra di metallo sottoposta a delle temperature costanti ai suoi limiti.

Utilizzando Jacobi per modellare Laplace si arriva a vedere che la temperatura su di un punto della lastra è data dalla formula:

$$f(x,y) = \frac{f(x+\Delta,y) + f(x-\Delta,y) + f(x,y+\Delta) + (fx,y-\Delta)}{4}$$

che altro non è che uno stencil a quattro punti.

Discretizzando il tutto, rappresentiamo la lastra su di una matrice di reali su cui itereremo l'algoritmo ossia calcolando, fino a raggiungimento del tasso di errore desiderato, per ogni elemento la sua temperatura sommando e quindi dividendo per 4 gli elementi con lui confinanti.

2 L'algoritmo sequenziale

L'algoritmo sequenziale viene eseguito su di una singola macchina, non ha quindi problemi di sincronia con altri calcolatori. Segue perciò l'algoritmo di Jacobi senza effettuare alcun cambiamento.

Se consideriamo `it_att` e `it_new` le matrici su cui si vanno ad effettuare i calcoli si ha:

```
do {
  // Calcolo i nuovi valori per gli elementi della matrice utilizzando
  // lo schema di Jacobi.
  for (unsigned int i=2; i<righe; i++)
    for (unsigned int j=2 ; j<colonne; j++)
      it_new[i][j] = 0.25*(it_att[i-1][j] + it_att[i][j+1] +
        it_att[i+1][j] + it_att[i][j-1]);

  // Calcolo l'errore relativo raggiunto considerandone il valore massimo
  // nel contempo copio i nuovi valori della matrice sulla matrice vecchia.
  errore_relativo = 0.0;
  for (unsigned int i=1; i<=righe; i++)
  for (unsigned int j=1 ; j<=colonne; j++) {
    errore_relativo = max(errore_relativo, fabs(it_new[i][j] - it_att[i][j]));
    it_att[i][j] = it_new[i][j];
  }
} while(errore_relativo > convergenza)
```

3 L'algoritmo parallelo

Per sua definizione un algoritmo parallelo rappresenta una procedura adatta ad essere eseguita contemporaneamente su più macchine così da distribuire il carico di lavoro dovuto alla sua esecuzione nella speranza di ottenerne i risultati in un tempo più breve rispetto a quello che si avrebbe utilzzando un algortimo di tipo sequenziale.

Diventa quindi ovvio che si presentano una serie di problemi nel concepire e nello sviluppare un algoritmo parallelo.

In questo caso la prima decisione da prendere riguardava il tipo di suddivisione della matrice di calcolo tra i vari processi, ossia se utilizzare una divisione a blocchi o a strisce della stessa, da assegnare poi ai processi. Del tutto arbitrariamente si è scelto di suddividere la matrice in strisce $(x \times N)$ con x variabile tra gli n processi tale che $\sum_{i=1}^{n} x_i = N$ e N fisso e uguale al lato della matrice stessa. In realtà questa non è una scelta priva di fondamento in quanto

permette di semplificare molto la parte dell'algoritmo che si occupa del trasferimento dei dati da processo a processo come si può capire dalle prossime sezioni.

Decisa la suddivisione della matrice resta il problema di sviluppare dei protocolli di comunicazione tra i processi in modo da permettere:

- il riconoscimento da parte dei vari processi di chi possiede i pezzi di matrice di cui neccessitano per finire una iterazione dell'algoritmo. Sostanzialmente se ad esempio la matrice ha lato N pari a 10, il numero di processi è uguale a 3 così che il processo i possiede le strisce $(3-6)$ della matrice allora dovrà conoscere quali processi possiedono le strisce 2 e 7 così da farne richiesta altrimenti non potrebbe calcolare i nuovi valori dei confini della propria matrice.
- lo scambio dei dati (strisce di dati) tra i processi per svolgere un'iterazione e lo scambio dei vari errori relativi così da decidere se continuare o meno con le iterazioni.
- la decisione di far migrare le strisce tra i processi in modo da alleviare il carico di lavoro ai processi lenti permettendone una distribuzione uniforme.
- il recupero dei dati forniti dalle computazioni dei vari processi per ricreare la matrice originale e presentare i risultati.

Di seguito verranno illustrati i protocolli ideati per risolvere i problemi appena elencati.

3.1 Protocollo di riconoscimento dei confini

E' un protocollo centralizzato in cui è il processo 0 (root) a prendere le decisioni e a informare gli altri processi (modello tipo client-server).

1. Per prima cosa i processi spediscono a root l'indice dei confini da loro posseduti. Se ad esempio l'*i*esimo processo possiede le strisce da 3 a 6 allora spedirà a root i valori 3 e 6.
2. Successivamente i vari processi spediranno a root l'indice dei confini di cui hanno bisogno (-1 altrimenti). Riprendendo l'esempio del punto precedente l'*i*esimo processo spedirà a root i valori 2 e 7, qualora N sia 6 spedirà -1 al posto di 7.
3. A questo punto i processi aspettano da root i *rank* dei processi che possiedono i confini da loro richiesti.
4. Per finire i processi ricevono da root un numero intero rappresentate il numero di processi clienti che dovranno servire durante l'esecuzione dell'algortimo di Jacobi. In pratica vengono a conoscenza di quanti dei loro confini sono richiesti dagli altri processi.

3.2 Protocollo per l'esecuzione di una iterazione dell'algoritmo

Questo protocollo non discerne il *rank* dei processi che eseguono tutti lo stesso codice:

1. Il processo inizia con una richiesta ed una attesa non bloccante dei confini di cui ha bisogno ai processi che ne sono in possesso (determinati utilizzando il protocollo di riconoscimento dei confini trattato precedentemente).
2. Procede con l'iniziare una attesa asincrona sulle richieste dei processi stranieri.
3. A questo punto esegue la parte di calcolo che non richiede l'utilizzo dei confini stranieri.
4. Quindi serve le richieste degli altri processi spedendo loro i confini richiesti.
5. Attende i confini altrui precedentemente richiesti per poter effettuare gli ultimi calcoli necessari alla fine dell'iterazione.
6. Finisce l'iterazione determinando l'errore relativo locale per poi confrontarlo con quello degli altri processi determinando e conservando il valore più alto.

3.3 Protocollo per lo scambio dinamico delle strisce

Anche questo è un protocollo centralizzato in cui è root a decidere e a comunicare il da farsi agli altri processi:

1. Ogni processo svolge la propria iterazione dell'algortimo di Jacobi tenendo traccia del tempo impiegato all'esecuzione per poterlo spedire a root.
2. *Punto valido solo per root.* Root calcola la media dei tempi di esecuzione. Nel caso in cui il tempo di esecuzione del processo più lento sia maggiore della media tenendo conto di un certo scarto (settabile) spedisce a tale processo il comando di diminuirsi il carico di lavoro di una striscia a favore del suo vicino (che verrà notificato anch'esso). A tali processi e a tutti gli altri sarà quindi richiesto di rieseguire il protocollo di riconoscimento dei confini.
3. Si attende una risposta da root agendo di conseguenza:

   -1: diminuire il carico di lavoro di una striscia e rieseguire il protocollo di riconoscimento dei confini.

   1: aumentare il carico di lavoro di una striscia e rieseguire il protocollo di riconoscimento dei confini.

   10: eseguire il protocollo di riconoscimento dei confini.

   0: continuare senza alcuna modifica.

3.4 Protocollo per il recupero dei dati

Il recupero dei dati ha senso se viene effettuato da un solo processo (sarebbe inutile replicare per n volte la matrice totale, rendendo quindi vano il tentativo di velocizzare l'esecuzione dell'algortimo). Viene da sè che anche questo protocollo è centralizzato:

1. Ogni processo manda a root l'indice dei propri confini. Quindi, ad esempio, il processo $i$esimo che possiede le strisce 3, 4 e 5 spedirà a root i valori 3 e 5.
2. Ogni processo crea un vettore, che spedirà poi a root, in cui inserisce i risultati delle proprie computazioni.
3. *Punto valido solo per root.* Raccolte le informazioni dagli altri processi, crea la matrice totale e ne inserisce i valori ricevuti, ne stampa poi i valori.

4 Lo sviluppo

L'applicazione è stata sviluppata utilizzando C++ come linguaggio di programmazione ed MPICH come libreria per il calcolo distribuito. Il C++ è stato scelto in primis perchè grazie al meccanismo dell'ereditarietà permette una suddivisione migliore delle procedure rendendo più efficiente l'isolamento degli errori. In secondo luogo utilizzando il polimorfismo è più semplice scrivere programmi che si adattano alla scelta di utilizzare l'algortimo statico piuttosto che quello dinamico (ossia se eseguire o no il protocollo per lo scambio dinamico delle strisce).

4.1 L'oggetto per l'algoritmo sequenziale

Si tratta della classe `BloccoJacobi` ed è alla base anche delle classi per il calcolo parallelo.

Include al suo interno la matrice su cui si svilupperà il calcolo (una matrice di vector STL) e implementa, tra gli altri i metodi `calcola_jacobi` (per l'esecuzione dell'algoritmo vero e proprio) e `ai_limiti` (per definire i limiti ai contorni della matrice).

Il tutto è contenuto nel file `blocco_jacobi.h`.

Classe `BloccoJacobi`		
Metodo	**Visibilità**	**Descrizione**
`BloccoJacobi(int ux, int uy, int dx, int dy)`	pubblica	Costruttore. Crea una matrice con dimensioni tali che la coordinata superiore sinistra sia (ux, uy) mentre quella inferiore destra sia (dx, dy).
`BloccoJacobi(unsigned int righe, unsigned int colonne)`	pubblica	Costruttore. Crea una matirce righe × colonne.
`void set_convergenza(float c)`	pubblica	Setta la convergenza, ossia l'errore tollerato dall'algoritmo, a c
`void set(int riga, int colonna, float v)`	pubblica	Setta l'elemento che si trova su `riga, colonna` a v.
`void set_all(float v)`	pubblica	Setta tutti gli elementi della matrice a v.
`void ai_limiti(float nord, float sud, float est, float ovest)`	pubblica	Setta i confini nord, sud, est e ovest della matrice ai rispettivi valori, che verranno mantenuti nel corso della computazione.
`unsigned int calcola_jacobi()`	pubblica	Esegua la computazione dell'algoritmo di Jacobi sulla matrice. Ritorna il numero di iterazioni effettuate.
`float get_tempo_calcolo()`	pubblica	Restituisce il tempo impiegato nel calcolo dell'algoritmo espresso in secondi.
`void mostra_risultato()`	pubblica	Stampa la matrice sullo standard output.
`void print_info()`	pubblica	Oltre alla matrice stampa sullo standard output anche delle informazioni riguardanti l'oggetto.

4.2 Gli oggetti per l'algoritmo parallelo

Le classi sviluppate sono tre ognuna alla base dell'altra:

- `PBloccoJacobi` è astratta, deriva da `BloccoJacobi` e sostanzialmente incapsula alcune variabili necessarie alla raprresentazione del singolo processo, dichiara l'interfaccia per le classi specializzate e definisce i metodi per la gestione del tempo (per avere informazioni riguardo al tempo di esecuzione). Si trova nel file `blocco_jacobi_par.h`.

Classe `PBloccoJacobi` derivata da `BloccoJacobi`		
Metodo	**Visibilità**	**Descrizione**
`PBloccoJacobi(int ux, int uy, int dx, int dy)`	pubblica	Costruttore. Crea una matrice con angolo superiore pari a (ux, uy) e inferiore pari a (dx, dy).
`virtual void ai_limiti(float nord, float sud, float est, float ovest)`	pubblica	Ridefinizione di `ai_limiti` della classe base. Viene reso astratto perché dipenderà dalla diversa suddivisione della matrice (a strisce o a blocchi).
`void set_passo_sincronia(unsigned int n)`	pubblica	Setta il passo per la sincronia, ossia il valore che indica ogni quante iterazioni i processi aggiornano i dati dei bordi stranieri.
`unsigned int get_num_sincronie()`	pubblica	Ritorna il numero di sincronie che sono avvenute tra i processi.
`float get_tempo_calcolo()`	pubblica	Ritorna il tempo impiegato per il calcolo dell'algoritmo espresso in secondi.
`float get_tempo_setup()`	pubblica	Ritorna il tempo impiegato per l'esecuzione del protocollo di riconoscimento dei confini espresso in secondi.
`float get_tempo_totale()`	pubblica	Ritorna il tempo impiegato totale. La somma dei due precedenti.
`float get_errore()`	pubblica	Ritorna l'errore relativo raggiunto dalla computazione.
`virtual void mostra_risultato()`	pubblica	Metodo virtuale. Servirà per implementare il protocollo di recupero dei dati.
`virtual unsigned int calcola_jacobi()`	pubblica	Metodo virtuale. Servirà per il calcolo vero e proprio dell'algoritmo.
`void print_info()`	pubblica	Ridefinizione di `print_info()` della classe base.

- `PBloccoJacobiStriscia` deriva da PBloccoJacobi e implementa l'algoritmo di Jacobi statico e distribuito, ossia il calcolo avviene in parallelo ma non c'è migrazione delle strisce tra i processi per variare dinamicamente il carico di lavoro. Implementa il protocollo di riconoscimento dei confini con la procedura `setup_confini`, quello per il calcolo

di una iterazione dell'algoritmo con la procedura `step_jacobi` e quello per il recupero dei dati con `mostra_risultato` .Si trova nel file `blocco_jacobi_par_striscia.h`.

Classe `PBloccoJacobiStriscia` derivata da `PBloccoJacobi`		
Metodo	**Visibilità**	**Descrizione**
`PBloccoJacobiStriscia(int ux, int uy, int dx, int dy)`	pubblica	Costruttore. Crea una matrice con angolo superiore pari a (ux, uy) e inferiore pari a (dx, dy). Crea inoltre un file di trace dal nome `trace`*id*`.log`.
`void ai_limiti(float nord, float sud, float est, float ovest)`	pubblica	Ridefinizione rispetto alla classe base. Specializzato per la topografia a strisce.
`void setup_confini()`	protetta	Implementa ed esegue il protocollo per il riconoscimento dei confini.
`void step_jacobi()`	pubblica	Esegue una iterazione dell'algoritmo di Jacobi implementando il relativo protocollo.
`unsigned int calcola_jacobi()`	pubblica	Definisce il metodo astratto della classe base. Non fa altro che servirsi dei due metodi precedenti per eseguire l'algoritmo. Restituisce il numero di iterazioni svolte.
`void mostra_risultato()`	pubblica	Implementa ed esegue il protocollo per la raccolta dei dati. Stampa la matrice sullo standard output.

- `PDBloccoJacobiStriscia` deriva da PBloccoJacobiStriscia e implementa il protocollo per lo scambio dinamico delle strisce nella procedura `calcola_jacobi`. Si trova nel file `blocco_jacobi_par_din.h`.

Classe `PDBloccoJacobiStriscia` derivata da `PBloccoJacobiStriscia`		
Metodo	**Visibilità**	**Descrizione**
`PDBloccoJacobiStriscia(int ux, int uy, int dx, int dy)`	pubblica	Costruttore. Uguale a quello della classe base, tranne che per la definizione di una varianza del 15% (vedi prossimo metodo).
`void set_varianza(unsigned int var)`	pubblica	Permette di settare la varianza accettata tra i tempi di esecuzione dei vari processi. Qualora un processo la superasse ridurrà il proprio carico di lavoro cedendone parte ad un vicino di matrice. Il valore *var* va espresso in percentuale.
`void cambia_matrice(int ux, int uy, int dx, int dy)`	privata	Cambia la matrice secondo le nuove coordinate.
`unsigned int calcola_jacobi()`	pubblica	Ridefinisce il metodo della classe base implementando il protocollo di scambio dinamico delle strisce. Ritorna il numero di iterazioni effettuate.
`void print_matirce()`	privata	Stampa sul file di trace (vedi classe base) la matrice corrente.

Diventa quindi interessante notare come `PBloccoJacobi` funga da interfaccia per accedere alle funzioni degli algoritmi distribuiti permettendo di scegliere a run-time il tipo di oggetto da istanziare:

```
if (dinamico)
  b = new PDBloccoJacobiStriscia(1, N, dim_strisce+N%nProcesses,1);
else
  b = new PBloccoJacobiStriscia(1, N, dim_strisce+N%nProcesses,1);
```

5 Possibili miglioramenti

Sono varie le possibili migliorie da apportare al programma, sia per aumentare l'efficienza (velocità) dell'algoritmo sia per incrementare la chiarezza e la riutilizzabilità del codice:

- Creare dei comunicatori specifici. MPI permette infatti la creazione di gruppi di pro-

cessi che possono comunicare tra loro tramite appositi identificatori chiamati comunicatori. Non sono stati utilizzati in questa implementazione in quanto è stato ritenuto sufficiente l'utilizzo dei *tag* (un identificatore associato ad una singola comunicazione, una Send e una Receive) non dovendo sviluppare una libreria riutilizzabile ma solamente un programma stand-alone.

- Migliorare il funzionamento del protocollo di scambio dinamico delle strisce. Questa implementazione si limita a far migrare una singola striscia per iterazione tra due processi adiacenti.
- Realizzare, partendo dalla classe `PBloccoJacobi`, la classe `PBloccoJacobiBox` che contenga non più una striscia di matrice ma un blocco arbitrario della stessa. In pratica creare l'algoritmo parallelo per la matrice suddivisa in blocchi e non in strisce.
- Renderlo più robusto, nel senso di fault tolerant. In questa implementazione qualora un processo smettesse di funzionare bloccherebbe completamente tutti gli altri mandandoli in stato di deadlock. Questo è dovuto all'utlizzo in alcune parti del codice di Send e Receive bloccanti, e del non utilizzo di timeout quando vengono utilizzate le relative procedure non bloccanti.

6 Risultati

Testando il programma con varie dimensioni del dataset si può notare come il suo andamento non sia da disprezzare. In realtà è il problema che si adatta bene ad essere parallelizzato in quanto non c'è sovraccarico dovuto ad un setup inziale (tutti i processi sanno come formare la matrice iniziale senza doverla ricevere da un processo esterno) ed i dati scambiati durante la fase di calcolo sono relativamente pochi.

Comunque si è notato un netto miglioramento dell'algoritmo all'aumentare della dimensione del problema, proprio come ci si può attendere da un algoritmo nato sequenziale e portato poi in parallelo.

Di seguito vengono riportati alcuni grafici che riassumono l'andamento delle prove effettuate.

Da notare che le prove hanno coinvolto un pool di macchine tutte uguali, ossia dei PC basati su Intel Celeron a 600 Mhz con 190 Mbyte di RAM collegati attraverso una subnet Ethernet a 100 Mbit al secondo con sistema operativo Linux Debian 3.0.

6.1 Algoritmo sequenziale

E' stato sviluppato per fornire il metro di misura per l'implementazione parallela. C'è poco da dire se non illustrare i risultai ottenuti visibili nella tabella 1.

Dimensione lato matrice	Tolleranza	Tempo (secondi)
100	0.1	2.5
200	0.1	11.03
500	0.1	65.32
800	0.1	165.19
1000	0.1	257.15
100	0.01	18.08
200	0.01	102
500	0.01	744.76
800	0.01	1793.45
1000	0.01	2585.30

Tabella 1: Tempi di calcolo dell'algoritmo sequenziale

6.2 Algoritmo parallelo statico

Con algoritmo parallelo statico si intende l'algoritmo parallelo che non implementa il protocollo di distribuzione dinamica delle strisce, si potrebbe quindi incappare in qualche processo lento che causa rallentamenti a tutti gli altri.

Sono state effettuate delle prove facendo variare il numero di processi da 2 a 10 (2, 5, 8, 10), il passo di sincronia da 1 a 10 (1, 5, 10) e con tolleranza pari a 0.1 e 0.01.

Nel grafico 1 viene illustrata una rappresentazione di massima dei tempi, nel numero 3 viene invece visualizzato lo speed-up raggiunto al variare del numero dei processi e della dimensione della matrice.

Si nota come per matrici inferiori a 500×500 non si raggiunge alcun beneficio dall'algoritmo che diventa invece molto buono all'aumentare delle dimensioni del problema.

Aumentando il passo di sincronia, ovvero indicando ogni quante iterazioni aggiornare i bordi dei processi, le cose non migliorano, anzi peggiorano quasi subito, come si evince dal grafico 4 in cui il passo di sincronia è stato portato a 5. Le performance migliorano con il passo a 10 ma solamente nel caso di grandi dimensioni della matrice, come si vede nel grafico 5. La cosa si può, parzialmente, spiegare consultando i dati delle tabella 2 e 3 che riassumono il numero di iterazioni e sincronie necessarie al raggiungimento del risultato variando il passo di sincronia sapendo che con passo 1 il numero di iterazioni (e quindi anche di sincronie) nel caso con tolleranza pari a 0.1 è sempre stato di 243. Il numero di iterazioni in più, necessarie per sopperire alle minorie sincronie, sono molto poche (mai più di 10) eppure lo speedup raggiunto con passo di sincronia 5 è molto basso. Non posso spiegare il fatto se non con la possibilità che qualche processo si sia trovato casualmente in una macchina già molto carica sacrificando tutte le prestazioni.

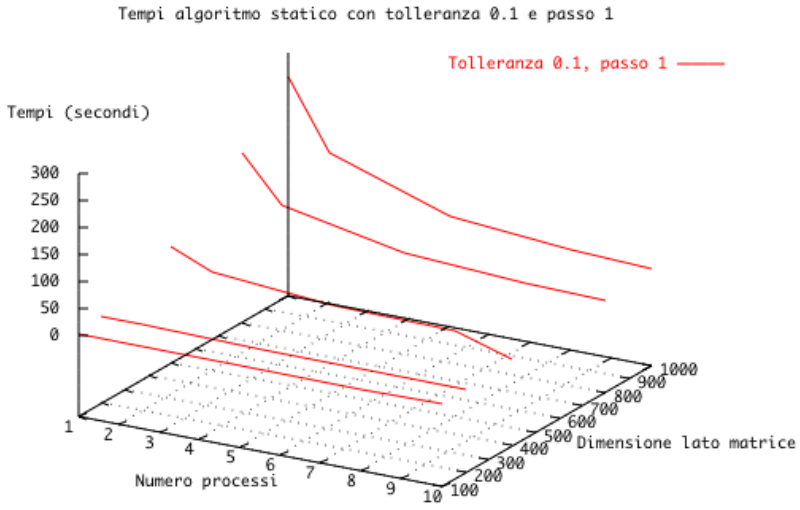

Figura 1: Tempi algoritmo parallelo statico

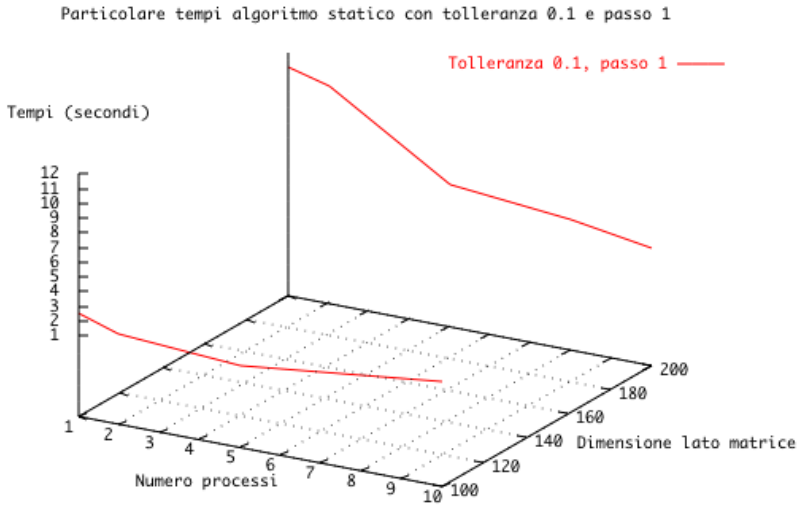

Figura 2: Particolare dei tempi dell'algoritmo parallelo statico

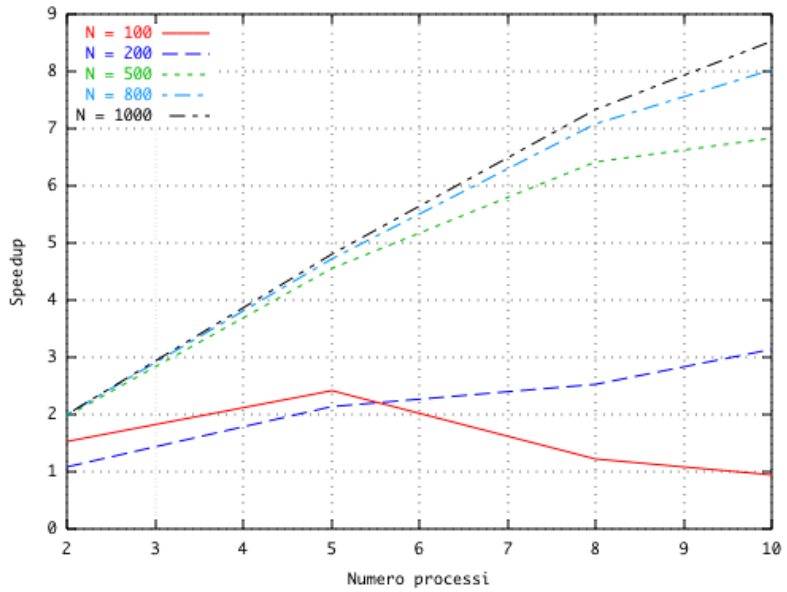

Figura 3: Speedup dell'algoritmo parallelo statico con tolleranza 0.1 e passo di sincronia 1

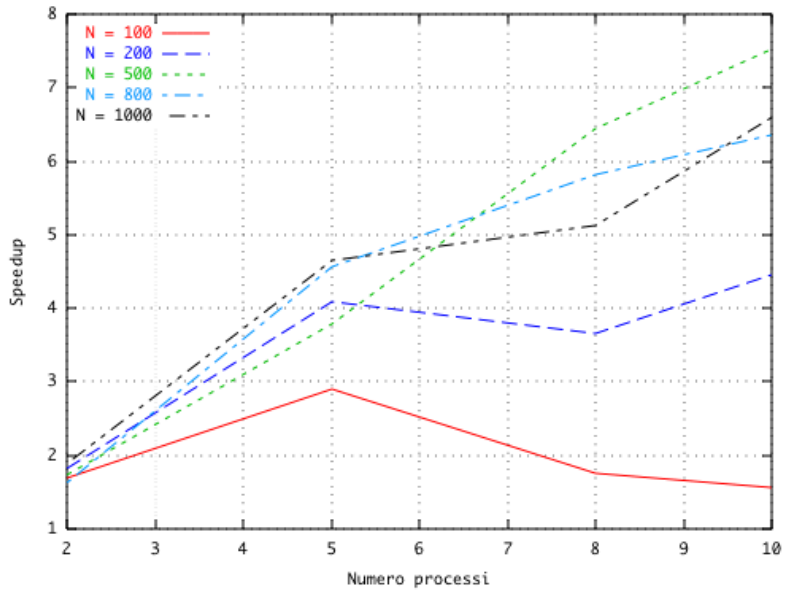

Figura 4: Speedup dell'algoritmo parallelo statico con tolleranza 0.1 e passo di sincronia 5

Tabella 2: Numero di iterazioni e sincronie con passo di sincronia 5 e tolleranza 0.1

Dimensione matrice	Numero processi	Numero iterazioni	Numero di sincronie
100	2	254	51
	5	250	50
	8	245	49
	10	245	49
200	2	254	51
	5	254	51
	8	250	50
	10	250	50
500	2	254	51
	5	254	51
	8	254	51
	10	254	51
800	2	254	51
	5	254	51
	8	254	51
	10	254	51
1000	2	254	51
	5	254	51
	8	254	51
	10	254	51

Tabella 3: Numero di iterazioni e sincronie con passo di sincronia 10 e tolleranza 0.1

Dimensione matrice	Numero processi	Numero iterazioni	Numero di sincronie
100	2	250	25
	5	249	25
	8	240	24
	10	240	24
200	2	250	25
	5	250	25
	8	250	25
	10	249	25
500	2	250	25
	5	250	25
	8	250	25
	10	250	25
800	2	250	25
	5	250	25
	8	250	25
	10	250	25
1000	2	250	25
	5	250	25
	8	250	25
	10	250	25

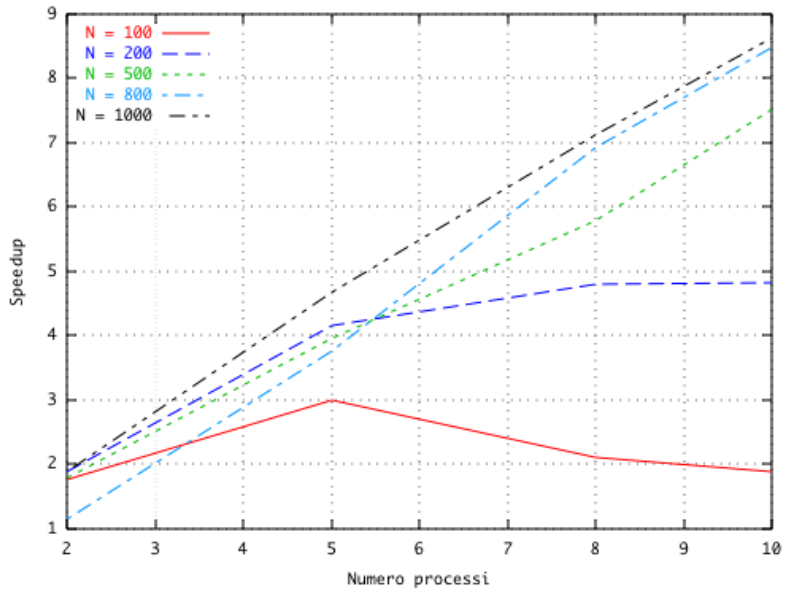

Figura 5: Speedup dell'algoritmo parallelo statico con tolleranza 0.1 e passo di sincronia 10

6.3 Algoritmo parallelo dinamico

Questo algoritmo implementa il protocollo di distribuzione dinamica delle strisce e, per lo meno in teoria, dovrebbe fornire delle prestazioni migliori.

Sono state effettuate delle prove con tolleranza a 0.1, passo di sincronia a 1 e scarto massimo tra i processi del 15%.

Come si evince dai grafici di speedup (figura 6 le velocità salgono, e non di poco, rendendolo molto peggiore alla versione statica. Probabilmente il degrado è dovuto alle prestazioni piuttosto simili dei computer che non rendono utile far migrare i dati in quanto non c'è vero sovraccarico di alcuni processi rispetto agli altri. Tra l'altro si dovrebbe anche vairare un po' lo scarto ammesso tra i vari tempi per cercare il valore migliore.

6.4 Note sul testing

Il testing di questi algortimi non è potuto essere dei migliori per motivi indipendenti dalla mia volontà. Sarebbe stato infatti opportuno fare più prove sotto le medesime condizioni per poi mediare i risultati, ma oltre a mancare il tempo necessario era praticamente impossibile ricreare le medesime condizioni di partenza. C'è infatti da tener presente che il laboratorio su

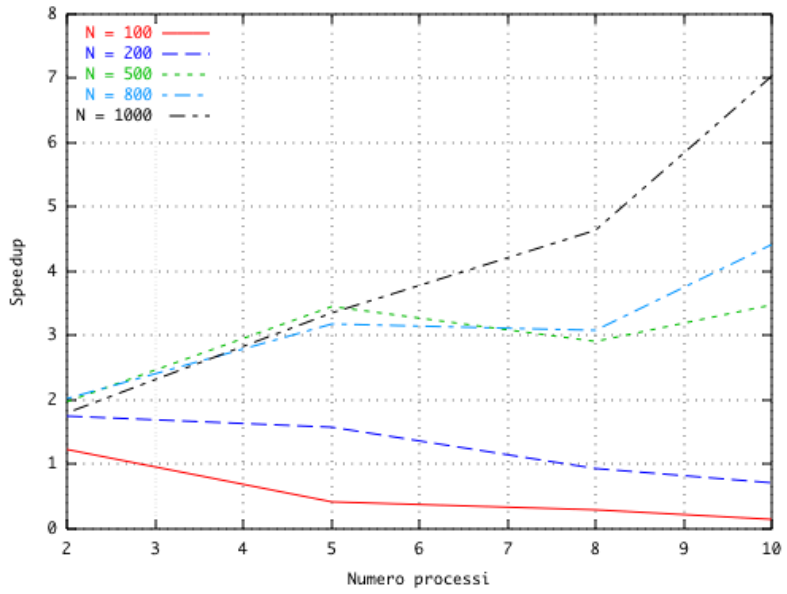

Figura 6: Speedup dell'algoritmo parallelo dinamico con tolleranza 0.1, passo di sincronia 1 e scarto massimo tra i processi del 15%

cui sono state effettuate le prove non è ad esclusivo utilizzo per il corso di Calcolo Parallelo, risulterebbe quindi ingiusto bloccare le macchine all'accesso degli altri studenti. Inoltre una fase di testing abbastanza accurata richiederebbe non meno di 5 giorni di prove a gruppo, per cui sapendo che i gruppo di lavoro sono 3 sarebbero state necessarie tre settimane di uso esclusivo del laboratorio da parte nostra, senza contare il tempo necessario allo sviluppo ed al debugging.

Tutto ciò solo per dire che i risultati vanno presi con le molle come può esemplificare il grafico di speedup dell'algoritmo statico con tolleranza a 0.01 e passo di sincronia a 1 (figura numero 7.

Si può infatti notare la quasi totale incongruenza dei risultati: con N pari a 1000 c'è un brusco blocco dalla prova con 8 processi a quella con 10 oppure, inspiegabilmente, il caso con matrice a 500 gira meglio di quello con matrice a 800 ma peggio di quello a 1000.

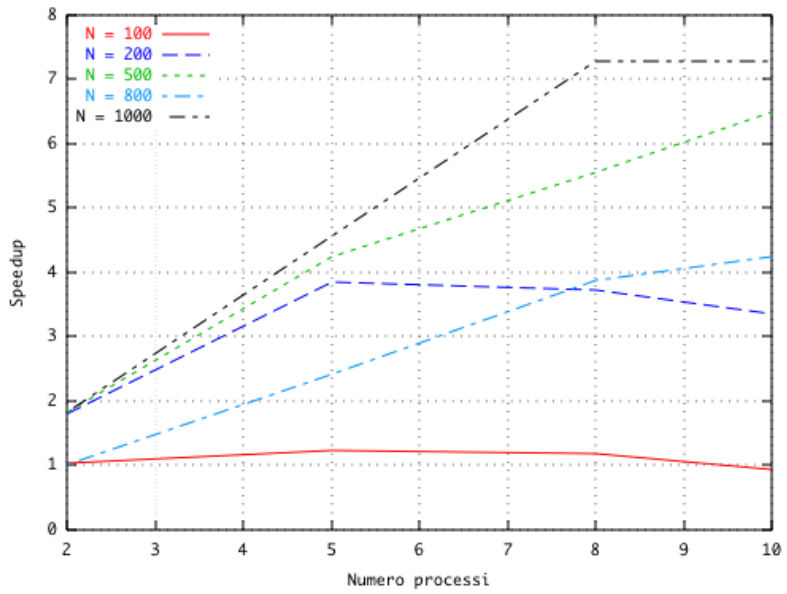

Figura 7: Speedup dell'algoritmo parallelo statico con tolleranza 0.01 e passo di sincronia 1.