Progetto di

Sistemi Concorrenti

Prof. Umberto Villano

Candidato:
Luciano De Lucia - 399000046

Indice

Introduzione

Lo scopo del progetto è quello di implementare l'algoritmo iterativo di Jacobi mediante l'impiego di pattern di programmazione per il calcolo parallelo.
Dato un algoritmo sequenziale che implementa il metodo di Jacobi, se ne realizzerà una versione parallela, ossia una nuova implementazione che ha lo scopo di distribuire la computazione su più processori in modo tale da aumentare la velocità di esecuzione rispetto ad un'implementazione sequenziale che invece sfrutta un singolo processore.
La soluzione cercata non ha il solo obiettivo della maggiore velocità, si cerca anche una soluzione che possa scalare, ossia aumentare le prestazioni con l'aggiunta di risorse computazionali. Pertanto la soluzione che andiamo a comporre dovrà permettere l'assegnazione di un determinato numero di processori su cui verrà dispiegato il calcolo parallelo. In fine si cerca anche una soluzione efficiente, cioè una soluzione che sfrutti al massimo le risorse computazionali utilizzate.
Per valutare l'algoritmo sotto i vari punti di vista sopracitati verranno misurati i tempi di esecuzione e ricavati i valori di *speedup* ed *efficienza* al variare delle risorse computazionali assegnate.

1 Jacobi Seriale

1.1 Metodi iterativi per la risoluzione di sistemi lineari

Un sistema di n equazioni in n incognite si può rappresentare come

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases} \quad (1.1)$$

Oppure in forma matriciale più compatta

$$Ax = b \quad (1.2)$$

Dove A è la matrice dei coefficienti, b è il vettore dei termini noti e x è il vettore soluzione. Per risolvere un sistema lineare così definito si possono usare sia metodi diretti che metodi iterativi. L'algoritmo di Jacobi, oggetto della trattazione, è un metodo iterativo.
Un metodo iterativo è caratterizzato dalla risoluzione del sistema in un numero indefinito di passi, a differenza dei metodi diretti. Un metodo di questo tipo parte da un vettore iniziale $x^{(0)}$ e genera una successione di vettori $\{ x^{(k)} \}$ con $k = 0,1,2,\dots$ che al limite deve poter fornire un'approssimazione adeguata della soluzione x. Si dice che un metodo iterativo è globalmente convergente se per ogni vettore iniziale $x^{(0)} \in \mathbb{R}^n$ si ha che

$$\lim_{k\to\infty} \|x^k - x\| = 0 \quad (1.3)$$

Un metodo si dice consistente se

$$x^{(k)} = x \Rightarrow x^{(k+1)} = x \quad (1.4)$$

Il metodo di Jacobi (metodo iterativo oggetto della trattazione) fa parte dei metodi lineari stazionari del primo ordine. Tali metodi possono essere scritti nella forma

$$x^{(k+1)} = Bx^{(k)} + f \quad (1.5)$$

Si dice lineare perché la legge che lo esprime è di tipo lineare, stazionario perche B, la matrice di iterazione, e il vettore f non dipendono dall'indice di

iterazione k. Infine è del primo ordine perché il vettore soluzione al passo $k+1$ dipende esclusivamente dal vettore soluzione al passo k.
Nell'implementazione di un metodo iterativo occorre tener conto di opportuni criteri di arresto che permettano di interrompere il calcolo. Un primo criterio di arresto riguarda lo scarto tra due soluzioni successive e, data tolleranza $\tau > 0$, può essere rappresentato come

$$\|x^k - x^{k-1}\| \leq \tau \|x^k\| \qquad (1.6)$$

Un secondo criterio riguarda il numero massimo consentito di iterazioni esprimibile nella forma

$$k > N_{max} \qquad (1.7)$$

1.2 Metodo di Jacobi

Tramite la tecnica dello splitting additivo si può esprimere la matrice A con la relazione

$$A = P - N \qquad (1.8)$$

dove la matrice di precondizionamento P è non singolare. Sostituendo questa relazione nell'equazione (1.5), e applicando la definizione di consistenza (1.4), si ricava

$$x^{(k+1)} = P^{-1}Nx^{(k)} + P^{-1}b \qquad (1.9)$$

Si consideri ora lo splitting additivo

$$A = D - E - F \qquad (1.10)$$

dove

$$D_{ij} = \begin{cases} a_{ii} & i = j \\ 0 & i \neq j \end{cases}, \; E_{ij} = \begin{cases} a_{ij} & i > j \\ 0 & i \leq j \end{cases}, \; F_{ij} \begin{cases} a_{ij} & i < j \\ 0 & i \geq j \end{cases} \qquad (1.11)$$

Si parla di metodo di Jacobi quando $P = D$ e $N = E + F$. Di conseguenza il metodo di Jacobi porta alla risoluzione, alla k-esima iterazione, del seguente sistema derivato dal (1.9)

$$(D - E)x^{(k+1)} = Fx^{(k)} + b \qquad (1.12)$$

In alternativa si può esprimere il metodo di Jacobi in forma di coordinate vettoriali, ovvero

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{i+1}^{n} a_{ij}x_j^{(k)}\right],\ i = 1,\dots,n \qquad (1.13)$$

Dove n rappresenta il numero di righe (e di colonne) della matrice A.
Questa formula consente di calcolare le componenti di $x^{(k+1)}$ in qualsiasi ordine e indipendentemente l'una dall'altra. Pertanto il metodo di Jacobi è parallelizzabile. Per questa ragione il metodo di Jacobi è anche detto ″delle sostituzioni simultanee″.

1.3 Algoritmo per il Metodo di Jacobi

l'algoritmo associato al metodo di Jacobi prende in input la matrice A, il vettore dei termini noti b e due parametri τ ed N_{max} che rappresentano rispettivamente la tolleranza e il numero massimo di iterazioni consentite.
C'è da considerare che l'algoritmo di Jacobi converge sicuramente quando la matrice A è diagonale dominante, ossia quando $|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|$.
$NMAX$ è un parametro definito sulla base delle dimensioni del problema e grande quanto basta per poter essere ragionevolmente sicuri che l'algoritmo non converga qualora si verifichi la disuguaglianza $k > NMAX$ (dove k è l'iterazione corrente).
Un algoritmo in pseudocodice per il metodo di Jacobi è il seguente:

input (a_{ij}), (b_i), τ, N_{max}
$k = 0$

for $i = 1$ to n do

    $x_i^{(0)} \leftarrow 0$
end do

do
    for $i = 1$ to n do

        $x_i^{(k+1)} \leftarrow \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{i+1}^{n} a_{ij}x_j^{(k)}\right) / a_{ii}$
    end do

    for $i = 1$ to n do

        $x_i^{(k)} \leftarrow x_i^{(k+1)}$
    end do

    $k \leftarrow k + 1$

while $(k \leq N_{max}) \wedge (\|x^{(k+1)} - x^{(k)}\| > \tau \|x^{(k+1)}\|)$

L'algoritmo itera fintanto che non è verificata la diseguaglianza (1.6) e non è verificata la disuguaglianza (1.7), cioè non si è superato il numero massimo di iterazioni consentito e, nel contempo, la condizione per la convergenza non è soddisfatta.
All'interno di ciascuna iterazione viene calcolata la soluzione successiva $x^{(k+1)}$ sulla base della soluzione corrente $x^{(k)}$ (l'algoritmo parte con una condizione iniziale $x^{(0)} = \varnothing$), dopodiché viene memorizzata la soluzione $x^{(k+1)}$ in $x^{(k)}$ per procedere con la prossima iterazione.
N_{max} e τ devono essere scelti in modo tale che, data la matrice A diagonale dominante, la condizione d'uscita

$$(k > N_{max}) \vee (\|x^{(k+1)} - x^{(k)}\| \leq \tau \|x^{(k+1)}\|)$$

deve risultare vera con $k \leq N_{max}$ verificata. Vale a dire che l'algoritmo deve terminare per via del soddisfacimento della condizione di convergenza e non per l'aver superato il numero massimo di iterazioni (che rappresenta la condizione di non convergenza). In altri termini non deve mai verificarsi che l'algoritmo segnali la non convergenza quando, invece, la matrice A è diagonale dominante (e pertanto l'algoritmo sicuramente convergere).

1.4 Memorizzazione di una matrice in forma compatta

Problemi ingegneristici che fanno uso di metodi iterativi, hanno spesso a che fare con matrici ad alti gradi di sparsità. Risulta pertanto molto più conveniente dal punto di vista computazionale memorizzare la matrice A in forma compatta, escludendo, cioè, tutti i coefficienti nulli. Oltre ad un ragguardevole risparmio di memoria, questa tecnica di memorizzazione consente anche di trascurare tutti i prodotti in cui uno dei fattori è a priori nullo ed il cui risultato è ovviamente già noto.
Il modo in cui viene memorizzata la matrice A del sistema lineare è detto *Compressed Row Storage* (CSR).
Da una generica matrice A quadrata di ordine n, la tecnica di memorizzazione CRS prevede la generazione di 3 vettori:

1. **SYSMAT**: vettore di numeri reali contenente gli nt coefficienti non nulli della matrice A memorizzati in successione per righe.

2. **JA**: vettore di numeri interi contenente gli nt indici di colonna dei corrispondenti elementi memorizzati in SYSMAT.

3. **IA**: vettore di numeri interi con $n+1$ componenti, contenente le posizioni in cui si trova in SYSMAT il primo elemento non nullo di ciascuna riga di A.

L'uso di IA e JA consente di individuare qualsiasi elemento non nullo a_{ij} memorizzato in SYSMAT. Infatti, l'elemento a_{ij} si troverà in una posizione k del vettore SYSMAT compresa nell'intervallo $IA_i \leq k \leq IA_{i+1}$ e tale per cui $JA_k = j$. Queste due condizioni permettono di individuare univocamente k per cui $SYSMAT_k = a_{ij}$. Si noti che l'occupazione di memoria si riduce da n^2 numeri reali (generalmente in doppia precisione) a nt numeri reali e $nt + n + 1$ numeri interi.
Ciò significa che maggiore è il grado di sparsità della matrice e maggiore è il beneficio ottenuto in termini di occupazione di memoria.

1.4.1 prodotto matrice-vettore

Occupiamoci ora di come eseguire il prodotto matrice-vettore quando la matrice è memorizzata in formato CSR. Il risultato ottenuto sarà utile per implementare un algoritmo di Jacobi che operi su di una matrice in forma compatta.
Si vuole calcolare il prodotto matrice-vettore

$$Av = w \qquad (1.13)$$

Con A matrice quadrata di ordine n, v e w vettori in $\mathbb{R}^n$. La componente i-esima di w è pari alla somma dei prodotti degli elementi della riga i di A per i corrispondenti componenti di v:

$$w_i = \sum_{j=1}^{n} a_{ij} v_j \qquad (1.14)$$

Se la matrice è memorizzata in modo compatto, gli elementi a_{ij} vanno opportunamente ricercati in SYSMAT mediante l'uso di IA, mentre gli indici j relativi alle colonne si trovano nel vettore intero JA. In particolare, gli elementi di A appartenenti alla riga i sono memorizzati in corrispondenza agli indici k del vettore SYSMAT compresi, per definizione di IA, nell'intervallo $IA_i \leq k \leq IA_{i+1} - 1$. Gli indici colonna j, di conseguenza, sono memorizzati in JA_k. il prodotto matrice-vettore, con A memorizzata in forma compatta, può quindi essere calcolato implementando il seguente algoritmo:

```
for i = 1 to n do
    w_i ← 0
    for k = IA_i to IA_{i+1} − 1 do
        j ← JA_k
        w_i ← w_i + (v_j · SYSMAT_z)
    end do
end do
```

1.4.2 Jacobi CSR

Il metodo di Jacobi esegue ad ogni iterazione una operazione simile al prodotto matrice-vettore A x $x^{(k)}$ (esclude da ogni prodotto scalare gli elementi sulla diagonale di A che andranno poi a dividere ciascun prodotto).
Pertanto è utile capire, una volta memorizzata A, in formato CSR, come eseguire tali prodotti accedendo ai vettori SYSMAT, JA e IA.
Utilizzando il risultato ottenuto in precedenza (prodotto matrice-vettore con matrice in formato CSR), opportunamente modificato per adattarlo all'algoritmo di Jacobi, otteniamo un nuovo algoritmo che implementa il metodo di Jacobi con la matrice A memorizzata in forma compatta:

input SYSMAT, IA, JA, b, τ, N_{max}

$k = 0$

for $i = 1$ to n do

    $x_i^{(0)} \leftarrow 0$

end do

do

    for $i = 1$ to n do

        $x_i^{(k+1)} \leftarrow b_i$

        for $z = IA_i$ to $IA_{i+1} - 1$ do

            $j \leftarrow JA_z$

            if $z \neq j$ then

                $x_i^{(k+1)} \leftarrow x_i^{(k+1)} - (x_j^{(k)} \cdot \mathrm{SYSMAT}_z)$

            end if

            $x_i^{(k+1)} \leftarrow x_i^{(k+1)} / \mathrm{SYSMAT}_{z \mid z=j}$

        end do

    end do

    for $i = 1$ to n do

        $x_i^{(k)} \leftarrow x_i^{(k+1)}$

    end do

    $k \leftarrow k + 1$

while $(k \leq N_{max}) \wedge (\|x^{(k+1)} - x^{(k)}\| > \tau \|x^{(k+1)}\|)$

2 Jacobi Parallelo

2.1 Caratteristiche Hardware: PoweRcost

Nell'implementare un programma parallelo c'è da tener conto delle caratteristiche Hardware del sistema a disposizione. Bisogna sapere, innanzitutto, se il sistema su cui operiamo è un sistema a memoria condivisa, o se è un sistema distribuito su più macchine che comunicano attraverso una rete di interconnessione. Questo ci indirizza nella scelta di opportuni pattern di programmazione concorrente (e di specifici strumenti) da utilizzare. Di seguito, quindi, una descrizione del sistema su cui il programma parallelo dovrà essere eseguito:

"PoweRcost è un cluster IBM che utilizza come sistema operativo la distribuzione Rocks 5.1. Si tratta di una distribuzione Linux basata su CentOS, pensata per la gestione dei cluster.
Il PoweRcost è formato da 2 rack: il rack di destra contiene 40 nodi computazionali, mentre il rack di sinistra ospita 14 nodi computazionale ed il frontend che controlla tutti i nodi. I nodi sono collegati da una rete Fast Ethernet (100 Mbps) e da una rete Myrinet ad alte prestazioni. I nodi del rack di destra sono degli IBM eServer x335, ognuno dotato di due processori Xeon da 2.8 GHz, 2 Gigabyte di memoria RAM e 2 dischi SCSI. Il frontend è un IBM eServer x345 Pentium 4 a 3GHz, un Gigabyte di RAM e un disco SATA. Sia il frontend che i nodi hanno l'hyperthreading abilitato."

Da tale descrizione possiamo osservare di aver a disposizione un sistema distribuito. Il programma parallelo da implementare dovrà quindi eseguire sui vari nodi del cluster un insieme di processi che si sincronizzeranno attraverso lo scambio di messaggi.
In aggiunta notiamo che ogni nodo ha a disposizione due processori. Ciò significa che ogni nodo è possibilmente un sistema a memoria condivisa su cui è possibile eseguire due processi (o quattro per sfruttare l'hyperthreading dei processori) per sfruttare un secondo livello di parallelismo.

2.2 MPI

Il Message Passing Interface (MPI) è una specifica che raggruppa un insieme di funzioni per il modello a scambio di messaggi su di un elaboratore parallelo, permettendo a processi residenti su processori diversi di scambiare dati e di sincronizzarsi pur non avendo uno spazio di indirizzamento comune della memoria (generalmente distribuita).

Il programma parallelo utilizzerà le funzioni della libreria MPI e verrà trasformato, dal middleware, in più processi in esecuzione sui diversi processori disponibili.
L'obiettivo è quello di far eseguire, ad ogni processo, lo stesso programma su dati differenti (Same Program Multiple Data - SPMD). E' possibile dichiarare, all'interno di un programma, una sezione parallela delimitata delle primitive MPI_Init() e MPI_Finalize(), ad indicare che quella porzione di programma verrà eseguita in concorrenza da più processi su più processori. Tali processi comunicheranno tra di loro attraverso opportune primitive di scambio messaggi, tra cui, ad esempio, MPI_Send() per l'invio e MPI_Receive() per la ricezione.
Oltre alle primitive per la comunicazione "punto punto" (MPI_Send e MPI_Receive), vengono messe a disposizione anche primitive per la comunicazione "collettiva", ossia tra un processo e molti processi (es. MPI_Scatter() o MPI_Broadcast()), tra molti processi ed uno (es. MPI_Gather()), e tra "molti" e "molti" (es. MPI_Allgather(), MPI_Allreduce()).
La distribuzione dei processi sui processori è trasparente al programmatore, che dovrà unicamente fornire al middleware le informazioni circa la posizione delle macchine su cui eseguire il programma parallelo e il numero di processi che dovranno eseguirlo. Di seguito la sintassi per la compilazione e l'esecuzione di un programma scritto utilizzando MPI:

mpicc -o [Exe Path] [Source Path]

mpirun -machinefile [Machinefile Path] -np [Number of Processes] [Exe Path]

E' possibile indicare, nel file *Machinefile*, gli *hostname* delle macchine su cui eseguire i vari processi, ed il numero di processi da creare (attraverso il flag *-np*). Ogni processo MPI viene caratterizzato da un identificativo (*rank)*. Sulla base di tale identificativo ciascun processo seguirà flussi di esecuzione possibilmente differenti, su dati differenti.

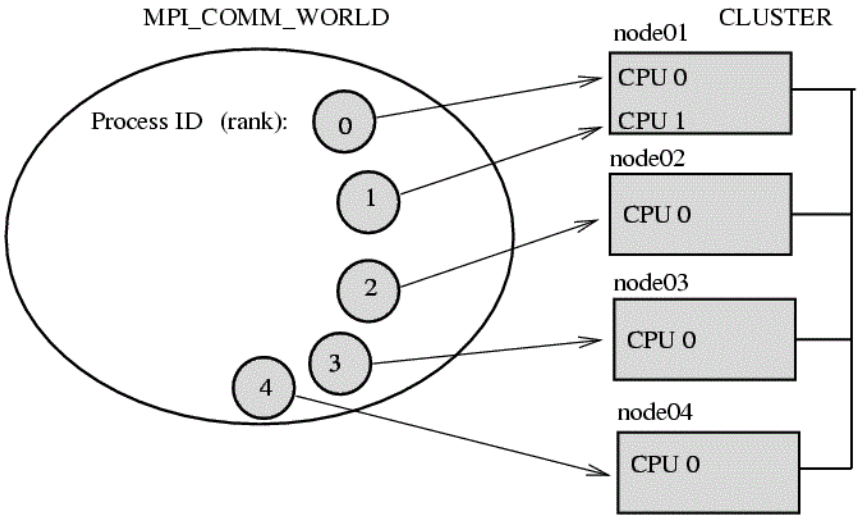


Figura 1 - dispiegamento di processi MPI sui nodi di un cluster

MPI definisce il concetto di "Comunicatore" come un insieme di processi.
Gli identificatori dei processi hanno senso all'interno di un comunicatore sicché anche le primitive di scambio messaggi valgono all'interno di un comunicatore. Il comunicatore di default è definito come "MPI_COMM_WORLD" e sta ad indicare l'insieme tutti i processi.
L'implementazione dell'algoritmo di Jacobi, parallelo, utilizzerà il middelware OpenMPI (implementazione dello standard MPI).

2.3 Jacobi Parallelo

Abbiamo osservato, nel paragrafo 1.2, che il metodo di Jacobi, ad ogni iterazione, permette di calcolare le componenti della soluzione al passo successivo ($x_i^{(k+1)}$), in qualsiasi ordine e indipendentemente l'una dall'altra.
Pertanto La porzione di codice parallelizzabile, è il ciclo **for** che calcola la soluzione $x_i^{(k+1)}$. Partiamo dall'algoritmo sequenziale per il metodo di Jacobi ed individuiamo quale è la porzione di algoritmo che è possibile parallelizzare.

input SYSMAT, IA, JA, b, τ, N_{max}
$k = 0$

for $i = 1$ to n do
    $x_i^{(k)} \leftarrow 0$
end do

do
    for $i = 1$ to n do ----------
        $x_i^{(k+1)} \leftarrow b_i$
        for $z = IA_i$ to $IA_{i+1} - 1$ do
            $j \leftarrow JA_z$
            if $z \neq j$ then
                $x_i^{(k+1)} \leftarrow x_i^{(k+1)} - (x_j^{(k)} \cdot \text{SYSMAT}_z)$
            end if
            $x_i^{(k+1)} \leftarrow x_i^{(k+1)} / \text{SYSMAT}_{z \mid z=j}$
        end do
    end do ---------- calcolo da parallelizzare

    for $i = 1$ to n do
        $x_i^{(k)} \leftarrow x_i^{(k+1)}$
    end do
    $k \leftarrow k + 1$

while $(k \leq N_{max}) \wedge (\|x^{(k+1)} - x^{(k)}\| > \tau \|x^{(k+1)}\|)$

Il fatto che ogni componente della soluzione, ad una determinata iterazione, è calcolabile in maniera indipendente dalle altre, permette di distribuire il calcolo su più processi.
Ogni processo potrà calcolare una porzione della soluzione, cioè eseguire un sottoinsieme di iterazioni del ciclo **for** individuato. E' possibile, quindi, utilizzare un particolare pattern di programmazione concorrente detto "parallelismo iterativo". Implementare il parallelismo iterativo significa sostanzialmente far si che tutti i processi eseguano un sottoinsieme delle iterazioni, su porzioni di dati differenti (perché sarà l'indice dell'iterazione ad indicare dove comincia la porzione di dati assegnata ad un processo).
Il parallelismo iterativo è un pattern di programmazione molto più vicino ad una soluzione basata su di un sistema a memoria condivisa in quanto le varie iterazioni avvengono su dati possibilmente *shared* (condivisi). In questo caso c'è solo da occuparsi di come distribuire le iterazioni tra i processi.
Quando ci troviamo su di un sistema distribuito, invece, oltre che di occuparsi della distribuzione delle iterazioni, bisogna trovare una soluzione affinché i vari processi, in esecuzione su processori che non condividono memoria, possano accedere a tali dati. Bisognerà, dunque, distribuire i dati prima di avviare il calcolo. In secondo luogo possiamo osservare come, una volta distribuiti i dati su cui ogni processo dovrà lavorare, essendo le iterazioni associate alle porzioni di dati su cui i vari processi eseguiranno, non c'è più necessità di conservare l'informazione sull'iterazione assegnata ad un processo, perché la porzione di dati ad esso assegnati è già stata distribuita. Molto più vicino ad una soluzione basata su di un sistema distribuito è, invece, il pattern "interacting peers" (con coordinatore). Un processo denominato *coordinator* si occuperà di distribuire le porzioni di dati, dette *job*, tra i vari processi, denominati in questo ambito *workers*.

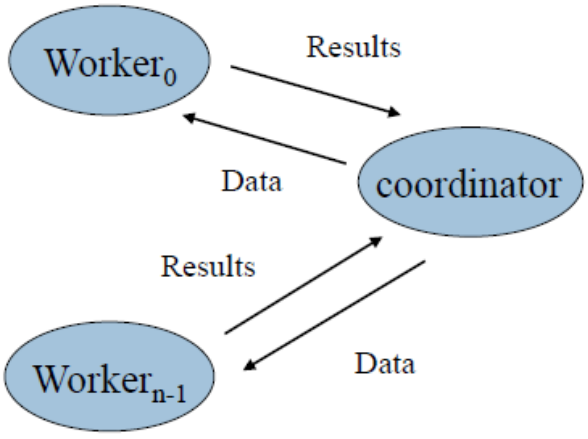


Figura 2 - interacting peers (con coordinatore)

Avvenuta la distribuzione ,ogni worker potrà, in parallelo agli altri, eseguire la propria computazione sul *job* ad esso assegnato.
Nell'implementazione dell'algoritmo di Jacobi, parallelo, ci si avvarrà, quindi, del pattern *Interacting Peers* (con coordinatore).
Un ulteriore problema da affrontare, ora, è quello della sincronizzazione. E' vero che i processi operano indipendentemente l'uno dall'altro per il calcolo della soluzione al passo k, ma la soluzione al passo successivo dipende dalla soluzione al passo precedente (essendo l'algoritmo iterativo). I *worker* quindi operano in parallelo quando si tratta di calcolare la soluzione in una determinata iterazione, successivamente però dovranno sincronizzarsi sulla soluzione totale di cui ogni processo ha calcolato una porzione. Questo perchè una porzione della soluzione, calcolata in una determinata iterazione dell'algoritmo di Jacobi, dipende dalla soluzione complessiva calcolata nell'iterazione precedente. Detto ciò, i *worker*, una volta calcolata una porzione di soluzione $x^{(k+1)}$ si scambieranno dei messaggi il cui contenuto sarà la propria porzione di soluzione calcolata, in modo tale che tutti possano essere aggiornati sulla soluzione complessiva $x^{(k+1)}$ e proseguire con il calcolo della propria porzione di soluzione all'iterazione successiva.
Per quanto riguarda la condizione d'uscita, ogni processo, in ogni iterazione, dispone sia di $x^{(k+1)}$ che di $x^{(k)}$ (data la sincronizzazione di cui si è parlato). Ciò fa si che ognuno possa valutare la condizione d'uscita dell'algoritmo senza bisogno di ulteriore sincronizzazione.
Di seguito una descrizione in pseudocodice dell'algoritmo di Jacobi, parallelo.

```
init() //Inizio sezione parallela
input τ, N_max
k = 0, coordinator = size − 1

//soluzione di partenza (x=0)
for i = 1 to n do
        x_i^(k) ← 0
end do

rank ← rank() //identificativo del processo
size ← size() //numero di processi

//COORDINATOR
if rank = size − 1 then
        //lettura matrice A (CRS) e vettore dei termini noti b
        input SYSMAT, IA, JA, b
```

```
//definizione della dimensione del job dei worker e del coordinatore. worker e coordinatore si
//dividono in parti uguali le righe della  matrice A (CRS). Il coordinatore lavora anche il resto.
JobW = ⌊length(IA)/size⌋
jobC = ⌊length(IA)/size⌋  + length(IA) mod size

//distribuzione job ai worker. ciclo su tutti i worker.
for worker = 0 to size − 2 do

        //creazione del job da distribuire al worker corrente.
        sysmatW ← sysmat_job(worker, jobW)
        iaW ← ia_job(worker, jobW)
        jaW ← ja_job(worker, jobW)
        bW ← job_b(worker, jobW)

        //Invio del job al worker
        send(sysmatW, worker)
        send(iaW, worker)
        send(jaW, worker)
        send(bW, worker)

end do

//Jacobi coordinatore. Il coordinatore esegue l'algoritmo sulla sua parte di job.
do
        for z = size ∗ jobW to n do

                x_i^(k+1) ← b_i
                for z = IA_i to IA_{i+1} − 1 do

                        j ← JA_z
                        if i ≠ j then

                                x_i^(k+1) ← x_i^(k+1) − (x_j^(k) · SYSMAT_z)
                        end if

                        x_i^(k+1) ← x_i^(k+1) / SYSMAT_{z | z=j}

                end do

        end do

        //il coordinatore riceve, da ogni i worker, una porzione di soluzione,
        //dopodiché costruisce la soluzione completa e la invia a tutti i worker.
        gather(x^(k+1), coordinator)
        broadcast(x^(k), coordinator)

        k = k + 1

while (k ≤ N_max) ∧ ( ‖x^(k+1) − x^(k)‖ > τ ‖x^(k+1)‖ )

//WORKER
else
        //ricezione del job.
        sysmatW ← receive(coordinator)
```

$iaW \leftarrow$ receive$(coordinator)$
$jaW \leftarrow$ receive$(coordinator)$
$bW \leftarrow$ receive$(coordinator)$

//Jacobi worker. Ogni worker lavora sulla parte di job ricevuta dal coordinatore.
do
for $i = 1$ to $length(iaW) - 1$ do

$xW_i^{(k+1)} \leftarrow b_i$
for $z = iaW_i$ to $iaW_{i+1} - 1$ do

$j \leftarrow jaW_z$
if $z \neq j$ then

$xW_i^{(k+1)} \leftarrow xW_i^{(k+1)} - (x_j^{(k)} \cdot sysmatW_z)$
end if

$xW_i^{(k+1)} \leftarrow xW_i^{(k+1)} /$ $\mathrm{SYSMAT}_{z \mid z=j}$
end do
end do

// aggiornamento soluzione al passo k. I worker inviano al coordinatore
//le porzioni di soluzione e ricevono dal coordinatore la soluzione complessiva.
gather$(xW^{(k+1)}, coordinator)$
broadcast$(x^{(k+1)}, coordinator)$

$k \leftarrow k + 1$

while $(k \leq N_{max}) \wedge (\|x^{(k+1)} - x^{(k)}\| > \tau \|x^{(k+1)}\|)$

end if

finalize() //Fine sezione parallela

Le funzioni evidenziate (in rosso) sono quelle che, nell'implementazione dell'algoritmo, saranno fornite da *OpenMPI*.

3 Analisi delle prestazioni

3.1 Tempo di esecuzione, Speedup ed Efficienza

Il tempo di esecuzione, per una determinata implementazione di un algoritmo, è un valore assoluto e sta ad indicarci quanto tempo impiega il processo ad elaborare i dati e produrre i risultati.
Lo speedup invece è una misura relativa. E' difatti il rapporto tra i tempi di esecuzione di due processi e ci dice di quante volte uno è più veloce dell'altro

$$speedup = \frac{T_s}{T_p} \qquad (3.1)$$

Nel nostro caso il numeratore sarà rappresentato dal tempo di esecuzione del programma seriale, mentre il denominatore da quello del programma parallelo.
Le misure effettuate hanno prodotto dati relativi ai tempi di esecuzione dei processi seriale e parallelo al variare sia della dimensione dei dati in input che del numero di processori utilizzati.
Oltre a sapere quanto sono veloci i processi, è utile avere l'indicazione circa quanto bene essi sfruttano le risorse assegnate. Pertanto entra in gioco la misura dell' efficienza vista come il rapporto tra lo speedup ed il numero di processori utilizzati. OpenMPI assegna a ciascun processore disponibile, un processo. Pertanto, a patto che il numero dei processi del programma parallelo sia minore o uguale del numero dei processori (ed effettivamente è ciò che accade nel nostro caso), verranno utilizzati tanti processori quanti sono i processi con cui vogliamo eseguire il nostro programma parallelo.

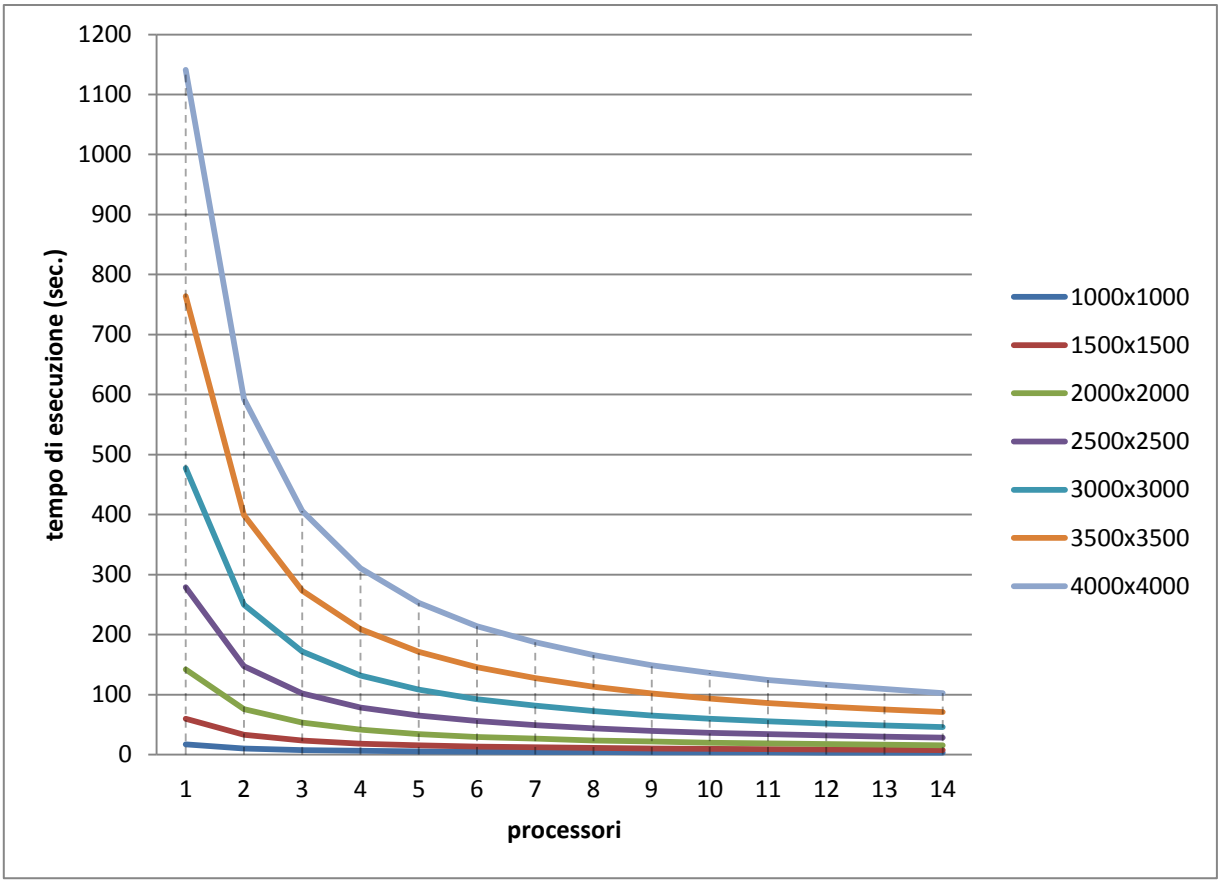
$$efficienza = \frac{speedup}{processori} \qquad (3.2)$$

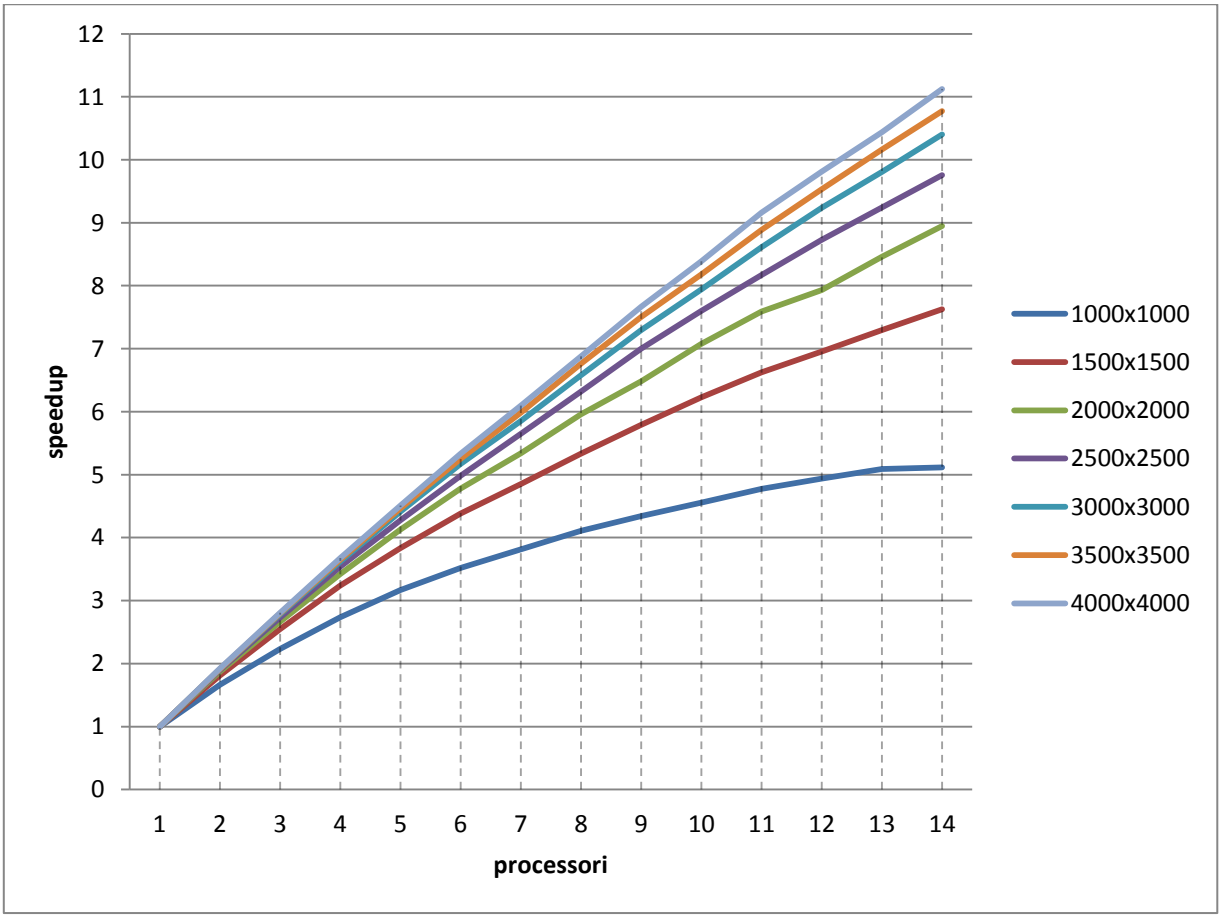
Speedup ed efficienza del programma seriale, per quanto detto, saranno pari ad 1. Questo perchè il tempo di esecuzione del programma seriale, confrontato con se stesso per ottenere lo speedup seriale, non può che dare un risultato unitario. Per quanto riguarda l'efficienza, essendo il programma seriale eseguito su di un solo processore, anche essa varrà 1. Per quanto riguarda il programma parallelo, invece, ci si aspetta che lo speedup, all'aumentare dei processi (e quindi dei processori utilizzati), aumenti per via del fatto di aver parallelizzato il calcolo. Per quanto riguarda l'efficienza, se lo speedup fosse sempre uguale al numero di processori utilizzati (caso ideale) allora avremmo sempre una efficienza del 100%. Ciò però non avviene, a causa del fatto che per

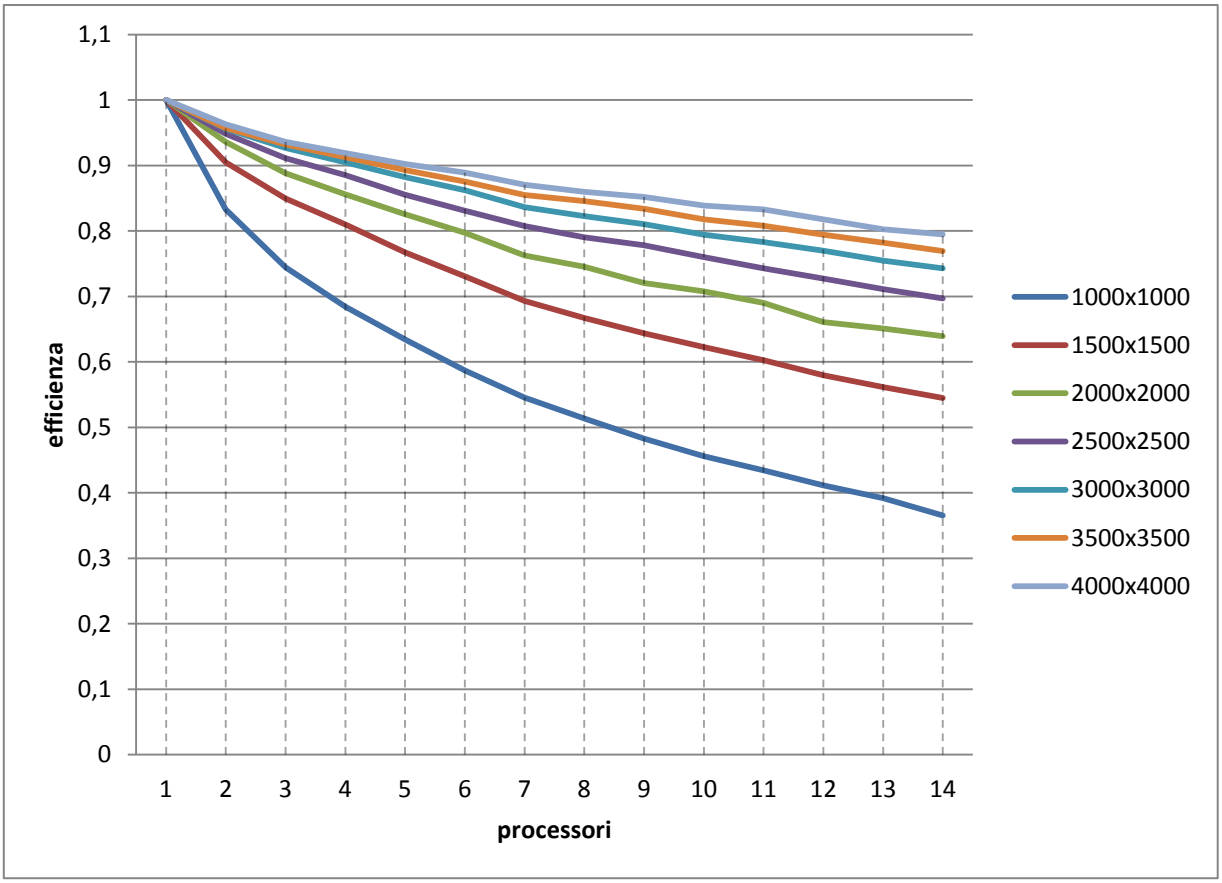
implementare il parallelismo si è aggiunto codice da eseguire che prima non appariva nel programma seriale, ed inoltre i processi non sono del tutto indipendenti gli uni dagli altri, perchè, come abbiamo visto, ad ogni iterazione essi devono sincronizzarsi sul risultato completo, ricostruendolo a partire dalle singole porzioni calcolate privatamente. Ciò fa si che si aggiungano ritardi di sincronizzazione e che, quindi, non si verifichi il cosiddetto "speedup lineare", fenomeno non impossibile ma dipendente anche dalle caratteristiche intrinseche di un algoritmo, oltre che dalla sua implementazione.

3.2 Misure e confronti

I dati sui tempi di esecuzione, dai quali si ricavano speedup ed efficienza, sono stati raccolti utilizzando il comando *time* di Unix e, in particolare, tra i tre valori che tale comando fornisce in output, è stato considerato il tempo *real*, ovvero il tempo trascorso dall'avvio al termine del programma.
Notiamo come, all'aumentare del numero di processori, il tempo di esecuzione diminuisca. Quando i dati sono relativamente pochi, la differenza tra il programma seriale e quello parallelo è minima, questo perchè l'overhead introdotto dalla sincronizzazione dei processi è di molto maggiore rispetto al calcolo. Ciò rende il programma parallelo inefficiente su problemi di piccole dimensioni. Man mano che le dimensioni del problema aumentano, lo speedup cresce, segno del fatto che il tempo di calcolo diventa preponderante sulla comunicazione. Con il crescere dei dati i processori vengono sfruttati sempre meglio. Ciò lo si evince dall'incremento dell' efficienza in corrispondenza di problemi più grandi. Fissata la dimensione del problema, aumentando i processori, lo speedup satura in accordo con la legge di Amdahl. Aggiungendo processori viene migliorata la parte dell'algoritmo che calcola in parallelo le varie componenti della soluzione. Risulta evidente che con il crescere dei processori la frazione migliorata diventa sempre meno determinante sul tempo totale che dipenderà maggiormente dall'overhead di sincronizzazione (c'è anche da tener conto che aumentando i processori aumentano i messaggi). In accordo con la legge di Gustafson, invece, aumentando le dimensioni del problema, aumentano anche le prestazioni perchè viene dato maggiore peso alla frazione del problema soggetta a miglioramento. Aumentando i dati aumenta il peso del calcolo rispetto a quello delle comunicazioni.
In fine osserviamo che, anche all'aumentare delle dimensioni del problema, lo speedup cresce sempre meno. Tale effetto è dovuto sia alla saturazione delle risorse che al contrasto delle comunicazioni che crescono al crescere delle dimensioni del problema. Maggiore è il numero dei dati e più iterazioni servono affinchè si raggiunga la soluzione. Più iterazioni abbiamo e più messaggi vengono scambiati dai processi.


tempo di esecuzione (sec.)
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14
processori
1000x1000
1500x1500
2000x2000
2500x2500
3000x3000
3500x3500
4000x4000


speedup
12
11
10
9
8
7
6
5
4
3
2
1
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14
processori
1000x1000
1500x1500
2000x2000
2500x2500
3000x3000
3500x3500
4000x4000


1,1
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
efficienza
1
2
3
4
5
6
7
8
9
10
11
12
13
14
processori
1000x1000
1500x1500
2000x2000
2500x2500
3000x3000
3500x3500
4000x4000