Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Partizionamento del grafo dataflow prodotto dal compilatore Chiara per la risoluzione di un sistema di equazioni lineari con il metodo di Jacobi

Giovanni Gallo, Lorenzo Verdoscia

RT-ICAR-NA-2010-01 **Settembre 2010**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Napoli, Via P. Castellino 111, I-80131 Napoli, Tel: +39-0816139508, Fax: +39-0816139531, e-mail: napoli@icar.cnr.it, URL: *www.na.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Partizionamento del grafo dataflow prodotto dal compilatore Chiara per la risoluzione di un sistema di equazioni lineari con il metodo di Jacobi

Giovanni Gallo, Lorenzo Verdoscia[1]

Rapporto Tecnico N.:
RT-ICAR-NA-2010-01

Data:
Settembre 2010

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Napoli, Via P. Castellino 111, 80131 Napoli

Indice

Elenco delle figure

1 Introduzione

Il lavoro riportato in questa memoria rientra nel tema di ricerca riguardante le architetture parallele con particolare riferimento alla programmazione di processi many-core basati sul paradigma di programmazione/esecuzione demand-data-driven [3]. Caratteristica peculiare di questi processi è che essi sono processi dataflow costituiti da migliaia di unità funzionali identiche [6], i core del processore, in grado di eseguire direttamente in hardware qualunque operazione appartenente al set di operatori elementari del linguaggio di programmazione del processore che è il linguaggio funzionale Chiara [5, 4] sviluppato presso l'ICAR.
In particolare, in questo report è presentato lo studio del partizionamento dei grafi dataflow di programmi prodotti dal compilatore Chiara perchè le partizioni così ottenute possano poi essere mappate ed eseguite dai processi dataflow.
L'ambito di studio in cui si è mossi è quello dei problemi di partizionamento o clustering di insiemi in cui possano essere presenti relazioni più o meno forti e vincolanti tra gli elementi che si intende raggruppare in componenti, con l'obiettivo di ottimizzare una determinata funzione della partizione stessa.
È noto che il problema SET-PARTITION, in cui, dato un insieme S di naturali, si richiede di costruire una bipartizione $\{S_1, S_2\} = S$, $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$ e $\sum_{s \in S_1} s = \sum_{s \in S_2} s = k$, o la sua generalizzazione da 2 a p componenti sono problemi NP-completi.
Dunque, di per sè, il problema di suddividere in sottoinsiemi una determinata famiglia di oggetti, rispettando una certa regola, costituisce un problema interessante dal punto di vista dello studio di algoritmi esatti o approssimanti di bassa complessità.
Il problema diventa ancora più articolato nel caso in cui vengono introdotte delle relazioni tra gli elementi della famiglia da partizionare; tali relazioni, che possono essere omogenee o caratterizzate da un diverso livello di legame tra gli oggetti della famiglia, ci suggeriscono di rappresentare il modello attraverso un grafo. In questo modo si possono formulare altri problemi in cui si richiede di tenere conto dei vincoli (gli spigoli del grafo) esistenti tra gli elementi dell'insieme (i vertici del grafo) da partizionare.
In generale si potrà richiedere di produrre delle partizioni in componenti connesse, tali da massimizzare o minimizzare il valore di una specifica funzione obiettivo. Ai vertici e agli spigoli del grafo potranno essere associati dei pesi che possono essere coinvolti nel calcolo della funzione obiettivo, contribuendo così a definire quei vincoli che devono essere rispettati nella costruzione delle componenti della partizione.
Le applicazioni sono numerose e vanno dallo studio dei sistemi elettorali, al-

l'analisi dei gruppi. Sono numerosi i problemi in ambito ingegneristico che possono essere ricondotti ad un problema di partizionamento vincolato: si va dalla progettazione di circuiti elettronici VLSI, alla definizione ottimale di una rete di telecomunicazione o alla ottimizzazione del contrasto in una immagine grafica digitalizzata.
In questo lavoro analizzeremo il metodo iterativo di Jacobi, la sua implementazione nel linguaggio funzionale Chiara e il partizionamento del grafo risultante dalla fase di compilazione mediante il tool di partizionamento Chaco[1].

2 I metodi iterativi stazionari

La soluzione numerica della maggior parte dei problemi di interesse nella fisica e nell'ingegneria in particolare, anche molto complessi, si riduce alla soluzione di un sistema di equazioni algebriche lineari. Ad esempio, i sistemi a parametri concentrati lineari in condizioni stazionarie sono governati da equazioni algebriche lineari. Inoltre, se si risolve un sistema di equazioni algebriche non lineari con il metodo di **Newton-Raphson** ad ogni passo bisogna risolvere un sistema di equazioni algebriche lineari mediante l'impiego di metodi iterativi, facilmente utilizzabili dai sistemi di calcolo automatico.
Il termine 'metodo iterativo' si riferisce ad un ampio range di tecniche che usano approssimazioni successive, ad ogni step, per ottenere soluzioni molto accurate nella soluzione di sistemi lineari. I metodi iterativi si dividono in stazionari e non stazionari. I metodi stazionari sono antichi, semplici da capire ed implementare, ma solitamente poco efficaci. I metodi non stazionari sono di recente sviluppo; la loro analisi è solitamente difficile da capire, ma possono essere altamente efficaci.
La velocità con la quale un metodo iterativo converge verso la soluzione del sistema dipende molto dallo spettro della matrice dei coefficienti. Per cui, i metodi iterativi solitamente richiedono una seconda matrice che trasforma la matrice dei coefficienti in un'altra dotata di uno spettro favorevole. Tale trasformazione di matrici è comunemente chiamata *preconditioner*. Un buon *preconditioner* migliora sufficientemente la convergenza di un metodo iterativo, in modo da superare il costo extra dovuto alla costruzione e applicazione del *preconditioner*. Infatti, senza il *preconditioner* un metodo iterativo può perfino non convergere. I metodi iterativi che sono espressi nella seguente forma:

$$x^{(k+1)} = Bx^{(k)} + c \tag{1}$$

(dove B e c non dipendono dal passo di iterazione k) sono chiamati metodi iterativi stazionari.
In questo lavoro presenteremo un metodo iterativo stazionario: il metodo di Jacobi.
Il metodo di Jacobi è basato sulla risoluzione locale di ogni variabile con rispetto per le altre; una iterazione del metodo corrisponde alla risoluzione simultanea di tutte la variabili. Questo metodo è facile da capire e implementare, ma converge lentamente verso la soluzione del sistema.

3 Metodo di Jacobi

Il metodo di Jacobi è facilmente ricavabile se si esaminano singolarmente ognuna delle n equazioni del sistema lineare $Ax = b$. Se si risolve la i-esima equazione

$$\sum_{j=1}^{n} a_{i,j}x_j = b_i, \tag{2}$$

rispetto ad x_i e si assume che gli altri valori di x sono fissi, si ottiene

$$x_i = \left(b_i - \sum_{j \neq i} a_{i,j}x_j \right) / a_{i,i}. \tag{3}$$

Questo ci suggerisce la definizione del seguente metodo iterativo

$$x_i^{(k+1)} = \left(b_i - \sum_{j \neq i} a_{i,j}x_j^{(k)} \right) / a_{i,i}, \tag{4}$$

che è appunto il metodo di Jacobi. E' da notare che l'ordine di valutazione delle equazioni è del tutto irrilevante, poichè il metodo di Jacobi le tratta in modo indipendente. Per questa ragione, il metodo di Jacobi è anche conosciuto come *method of simultaneous displacements*, poichè la modifica in linea di principio può essere fatta simultaneamente.
In termini matriciali, la definizione del metodo di jacobi di (4) può essere espressa come

$$x^{(k+1)} = D^{-1}\left(L + U\right)x^k + D^{-1}b, \tag{5}$$

dove, rispettivamente, D è la matrice diagonale, -L è la matrice triangolare inferiore e -U è la matrice triangolare superiore di A.
Il metodo converge per tutti i sistemi lineari con matrice a diagonale dominante. Inoltre, la condizione d'arresto è $\mid x^{(k+1)} - x^{(k)} \mid < \epsilon$ (tolleranza richiesta).

4 Implementazione dell'algoritmo nel linguaggio Chiara

Per ottenenere il grafo dataflow per la risoluzione di un sistema lineare mediante il metodo di Jacobi, è stato scritto il programma 'jacobi.chr' nel linguaggio funzionale Chiara che implementa l'algoritmo per un sistema di quattordici equazioni in quattordici incognite in modo da ottenere un grafo significativo per essere partizionato ed eseguito su processori dataflow costituiti da 128 Unità Funzionali identiche.
Al fine di fornire una corretta spiegazione del programma, il file 'jacobi.chr' è stato suddiviso in tre parti. Nella prima parte (Figura 1), sono presenti le costanti che definiscono la matrice dei coefficienti, nella seconda parte (Figura 2), invece, sono presenti le costanti che definiscono il vettore dei termini noti e il vettore diagonale principale, mentre nella terza parte sono definite le funzioni che implementano l'algoritmo di Jacobi.

Il significato delle funzioni di Figura 3 è il seguente:

- **IP** calcola il prodotto interno;
- **MV** calcola il prodotto matrice-vettore;
- **ABS** fornisce il valore assoluto di un numero;
- **SOTTRAZIONE** calcola la differenza tra il vettore dei termini noti e il risultato di **MV**;
- **DIVDIAGONALE** calcola la divisione tra il risultato di **SOTTRAZIONE** e gli elementi della diagonale principale;
- **XUP** calcola le x_i per $i = 1, \ldots, 14$ al passo (k+1)-esimo;
- **XMENOXUP** calcola $x_i^{(k+1)} - x_i^{(k)}$ per $i = 1, \ldots, 14$;
- **VALOREASSOLUTOVETTORE** fornisce il valore assoluto del vettore calcolato da **XMENOXUP**;
- **CONTROLLO** fornisce in uscita il valore 'token valido' se la condizione $\mid x_i^{(k+1)} - x_i^{(k)} \mid$<eps per $i = 1, \ldots, 14$, il valore bottom '$\perp$' altrimenti;
- **ITER** è il blocco della iterazione;
- **TEST** valuta la condizione di uscita dal ciclo **repeat**.

```
def a1x1y = %0
def a1x2y = %3
def a1x3y = %3
def a1x4y = %1
def a1x5y = %1
def a1x6y = %2
def a1x7y = %2
def a1x8y = %2
def a1x9y = %2
def a1x10y = %1
def a1x11y = %3
def a1x12y = %2
def a1x13y = %2
def a1x14y = %2
def a2x1y = %2
def a2x2y = %0
def a2x3y = %3
def a2x4y = %1
def a2x5y = %2
def a2x6y = %2
def a2x7y = %1
def a2x8y = %1
def a2x9y = %3
def a2x10y = %3
def a2x11y = %3
def a2x12y = %2
def a2x13y = %2
def a2x14y = %2
def a3x1y = %3
def a3x2y = %3
def a3x3y = %0
def a3x4y = %3
def a3x5y = %2
def a3x6y = %1
def a3x7y = %2
def a3x8y = %1
def a3x9y = %3
def a3x10y = %2
def a3x11y = %1
def a3x12y = %1
def a3x13y = %3
def a3x14y = %2

def a4x1y = %2
def a4x2y = %3
def a4x3y = %3
def a4x4y = %0
def a4x5y = %2
def a4x6y = %2
def a4x7y = %3
def a4x8y = %3
def a4x9y = %2
def a4x10y = %3
def a4x11y = %2
def a4x12y = %3
def a4x13y = %1
def a4x14y = %3
def a5x1y = %2
def a5x2y = %1
def a5x3y = %3
def a5x4y = %2
def a5x5y = %0
def a5x6y = %1
def a5x7y = %2
def a5x8y = %2
def a5x9y = %1
def a5x10y = %3
def a5x11y = %3
def a5x12y = %2
def a5x13y = %3
def a5x14y = %3
def a6x1y = %2
def a6x2y = %1
def a6x3y = %2
def a6x4y = %3
def a6x5y = %2
def a6x6y = %0
def a6x7y = %1
def a6x8y = %1
def a6x9y = %3
def a6x10y = %1
def a6x11y = %1
def a6x12y = %1
def a6x13y = %3
def a6x14y = %2

def a7x1y = %1
def a7x2y = %1
def a7x3y = %1
def a7x4y = %3
def a7x5y = %2
def a7x6y = %1
def a7x7y = %0
def a7x8y = %2
def a7x9y = %3
def a7x10y = %2
def a7x11y = %2
def a7x12y = %3
def a7x13y = %1
def a7x14y = %1
def a8x1y = %1
def a8x2y = %3
def a8x3y = %2
def a8x4y = %3
def a8x5y = %2
def a8x6y = %1
def a8x7y = %3
def a8x8y = %0
def a8x9y = %2
def a8x10y = %2
def a8x11y = %1
def a8x12y = %3
def a8x13y = %3
def a8x14y = %1
def a9x1y = %2
def a9x2y = %1
def a9x3y = %1
def a9x4y = %3
def a9x5y = %2
def a9x6y = %3
def a9x7y = %2
def a9x8y = %3
def a9x9y = %0
def a9x10y = %2
def a9x11y = %1
def a9x12y = %2
def a9x13y = %1
def a9x14y = %2

def a10x1y = %3
def a10x2y = %1
def a10x3y = %3
def a10x4y = %2
def a10x5y = %3
def a10x6y = %2
def a10x7y = %3
def a10x8y = %1
def a10x9y = %2
def a10x10y = %0
def a10x11y = %2
def a10x12y = %2
def a10x13y = %3
def a10x14y = %3
def a11x1y = %1
def a11x2y = %2
def a11x3y = %1
def a11x4y = %2
def a11x5y = %1
def a11x6y = %3
def a11x7y = %3
def a11x8y = %2
def a11x9y = %2
def a11x10y = %3
def a11x11y = %0
def a11x12y = %1
def a11x13y = %3
def a11x14y = %1
def a12x1y = %2
def a12x2y = %3
def a12x3y = %2
def a12x4y = %2
def a12x5y = %2
def a12x6y = %3
def a12x7y = %1
def a12x8y = %2
def a12x9y = %3
def a12x10y = %2
def a12x11y = %3
def a12x12y = %0
def a12x13y = %1
def a12x14y = %3

def a13x1y = %2
def a13x2y = %2
def a13x3y = %3
def a13x4y = %3
def a13x5y = %1
def a13x6y = %1
def a13x7y = %2
def a13x8y = %3
def a13x9y = %2
def a13x10y = %1
def a13x11y = %3
def a13x12y = %3
def a13x13y = %0
def a13x14y = %2
def a14x1y = %3
def a14x2y = %1
def a14x3y = %3
def a14x4y = %3
def a14x5y = %3
def a14x6y = %2
def a14x7y = %2
def a14x8y = %3
def a14x9y = %1
def a14x10y = %1
def a14x11y = %1
def a14x12y = %3
def a14x13y = %2
def a14x14y = %0
```

Figura 1: Programma Jacobi.chr parte 1: costanti che definiscono la matrice dei coefficienti.

Inoltre, come valori di input del vettore soluzione iniziale $x^{(0)}$ è stato assunto il vettore dei termini noti.
Analizzando le tre parti del programma jacobi.chr, è possibile osservare che le funzioni così definite, non dipendono dalla dimensione del sistema e di conseguenza possono essere utilizzate per l'implementazione in Chiara del metodo di Jacobi per sistemi di qualsiasi dimensione. Al fine di sfruttare la generalità delle funzioni di Figura 3, si è realizzato un programma 'genera_sorgente.c', in linguaggio C sviluppato in ambiente Linux-Ubuntu, che genera automaticamente il sorgente jacobi.chr per sistemi di qualsiasi dimensione. Dopo la compilazione del file 'genera_sorgente.c' (comando *gcc -o genera_sorgente genera_sorgente.c*), la sua esecuzione (comando *./gen-*

```
def riga1 = [a1x1y,a1x2y,a1x3y,a1x4y,a1x5y,a1x6y,a1x7y,a1x8y,a1x9y,a1x10y,a1x11y,a1x12y,a1x13y,a1x14y]
 def riga2 = [a2x1y,a2x2y,a2x3y,a2x4y,a2x5y,a2x6y,a2x7y,a2x8y,a2x9y,a2x10y,a2x11y,a2x12y,a2x13y,a2x14y]
 def riga3 = [a3x1y,a3x2y,a3x3y,a3x4y,a3x5y,a3x6y,a3x7y,a3x8y,a3x9y,a3x10y,a3x11y,a3x12y,a3x13y,a3x14y]
 def riga4 = [a4x1y,a4x2y,a4x3y,a4x4y,a4x5y,a4x6y,a4x7y,a4x8y,a4x9y,a4x10y,a4x11y,a4x12y,a4x13y,a4x14y]
 def riga5 = [a5x1y,a5x2y,a5x3y,a5x4y,a5x5y,a5x6y,a5x7y,a5x8y,a5x9y,a5x10y,a5x11y,a5x12y,a5x13y,a5x14y]
 def riga6 = [a6x1y,a6x2y,a6x3y,a6x4y,a6x5y,a6x6y,a6x7y,a6x8y,a6x9y,a6x10y,a6x11y,a6x12y,a6x13y,a6x14y]
 def riga7 = [a7x1y,a7x2y,a7x3y,a7x4y,a7x5y,a7x6y,a7x7y,a7x8y,a7x9y,a7x10y,a7x11y,a7x12y,a7x13y,a7x14y]
 def riga8 = [a8x1y,a8x2y,a8x3y,a8x4y,a8x5y,a8x6y,a8x7y,a8x8y,a8x9y,a8x10y,a8x11y,a8x12y,a8x13y,a8x14y]
 def riga9 = [a9x1y,a9x2y,a9x3y,a9x4y,a9x5y,a9x6y,a9x7y,a9x8y,a9x9y,a9x10y,a9x11y,a9x12y,a9x13y,a9x14y]
 def riga10 = [a10x1y,a10x2y,a10x3y,a10x4y,a10x5y,a10x6y,a10x7y,a10x8y,a10x9y,a10x10y,a10x11y,a10x12y,a10x13y,a10x14y]
 def riga11 = [a11x1y,a11x2y,a11x3y,a11x4y,a11x5y,a11x6y,a11x7y,a11x8y,a11x9y,a11x10y,a11x11y,a11x12y,a11x13y,a11x14y]
 def riga12 = [a12x1y,a12x2y,a12x3y,a12x4y,a12x5y,a12x6y,a12x7y,a12x8y,a12x9y,a12x10y,a12x11y,a12x12y,a12x13y,a12x14y]
 def riga13 = [a13x1y,a13x2y,a13x3y,a13x4y,a13x5y,a13x6y,a13x7y,a13x8y,a13x9y,a13x10y,a13x11y,a13x12y,a13x13y,a13x14y]
 def riga14 = [a14x1y,a14x2y,a14x3y,a14x4y,a14x5y,a14x6y,a14x7y,a14x8y,a14x9y,a14x10y,a14x11y,a14x12y,a14x13y,a14x14y]

 def matrice = [riga1,riga2,riga3,riga4,riga5,riga6,riga7,riga8,riga9,riga10,riga11,riga12,riga13,riga14]


 def d1 = %27
 def d2 = %28
 def d3 = %28
 def d4 = %33
 def d5 = %29
 def d6 = %24
 def d7 = %24
 def d8 = %28
 def d9 = %26
 def d10 = %31
 def d11 = %26
 def d12 = %30
 def d13 = %29
 def d14 = %29

 def diagonale = [d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14]

 def b1 = %2
 def b2 = %3
 def b3 = %2
 def b4 = %2
 def b5 = %1
 def b6 = %1
 def b7 = %2
 def b8 = %1
 def b9 = %3
 def b10 = %2
 def b11 = %2
 def b12 = %2
 def b13 = %3
 def b14 = %3

 def termininoti = [b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14]
```

Figura 2: Programma Jacobi.chr parte 2: costanti che definiscono il vettore dei termini noti e il vettore diagonale principale.

```
def eps = %0.000001
def IP = ! + @ & * @ trans
def ABS = (geq @ [id,%0] --> id; * @ [%-1,id])
def XMENOXUP = & - @ trans
def VALOREASSOLUTOVETTORE = & ABS
def CONTROLLO = ! + @ & (lt @ id-->%0; %1) @ distr
def MV = & IP @ distl
def SOTTRAZIONE = & - @ trans
def DIVDIAGONALE = & / @ trans
def CONTROLLO = ! + @ & (lt @ id-->%0; %1) @ distr
def XUP = DIVDIAGONALE @ [SOTTRAZIONE @ [termininoti, MV @ [1, matrice] ], diagonale]
def ITER = [2, CONTROLLO @ [VALOREASSOLUTOVETTORE @ XMENOXUP, eps]] @ [1, XUP]
def TEST = neq @ [2, %0]



(repeat ITER, TEST) :<<2,3,2,2,1,1,2,1,3,2,2,2,3,3>,0>



stop
```

Figura 3: Programma Jacobi.chr parte 3: funzioni utilizzate per implementare l'algoritmo di Jacobi.

era_sorgente) chiede all'utente di digitare la dimensione della matrice dei coefficienti (Figura 4).

Al termine dell'esecuzione del file 'genera_sorgente' sono creati due file: 'jacobi.chr' e 'matrice.txt'.
Il file 'jacobi.chr' (Figura 5) contiene l'implementazione dell'algoritmo di Jacobi per un sistema di dimensione precedentemente decisa dall'utente e può, quindi, essere sucessivamente compiltato dal compilatore Chiara.
Il file 'matrice.txt' (Figura 6) contiene le costanti definite in 'jacobi.chr', quali: la matrice dei coefficienti; il vettore diagonale principale; il vettore dei termini noti e il vettore soluzione iniziale.
Tutte le costanti in 'matrice.txt' sono generate dal file 'genera_sorgente.c' in modo casuale ed inoltre la matrice dei coefficienti è generata in modo

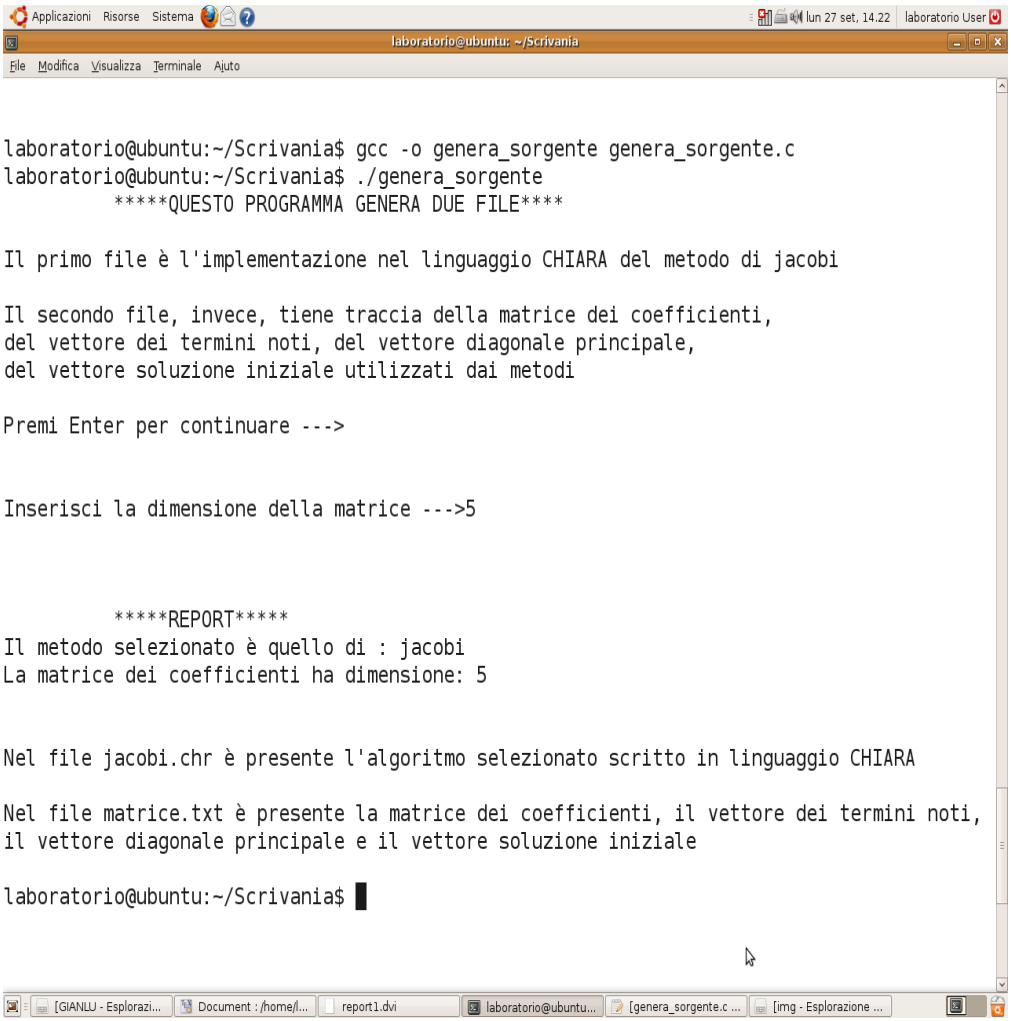

Figura 4: Compilazione ed esecuzione del file genera_sorgente.c .

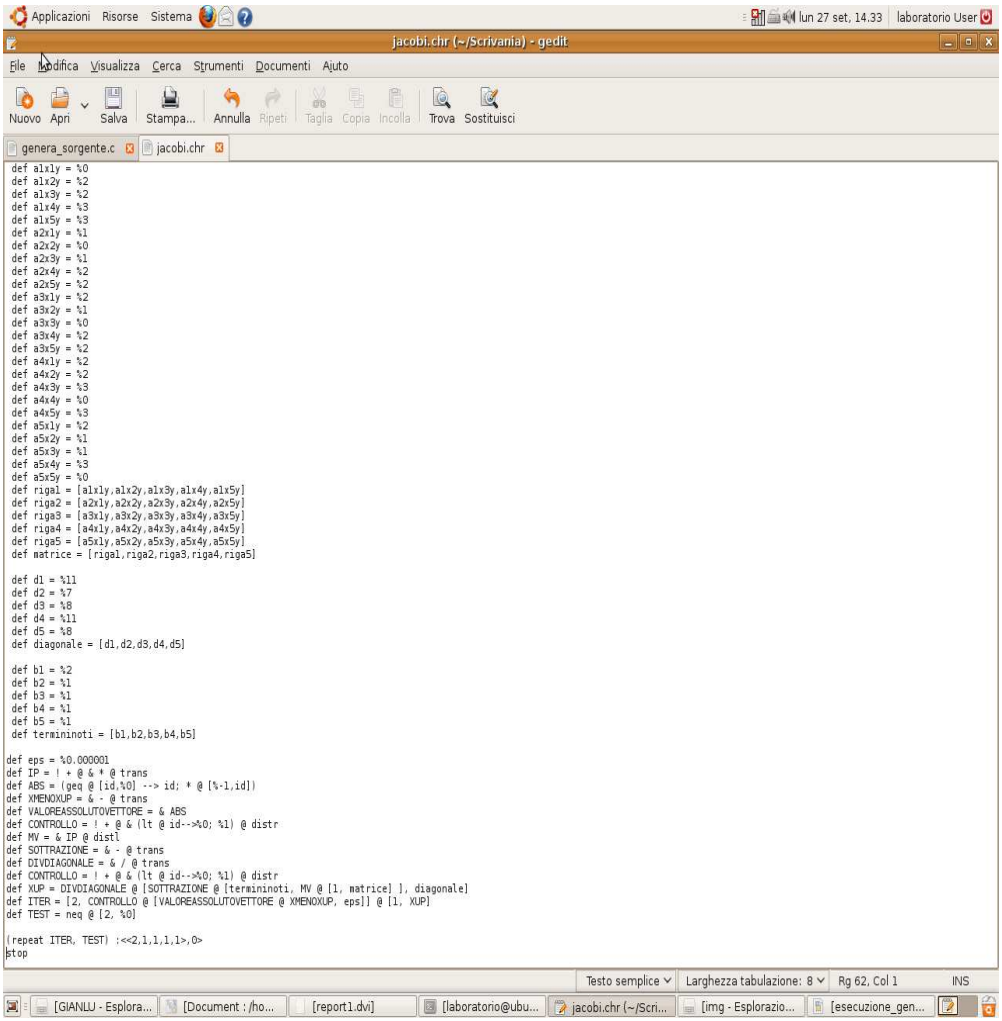

Figura 5: Sorgente jacobi.chr generato da genera_sorgente.c.

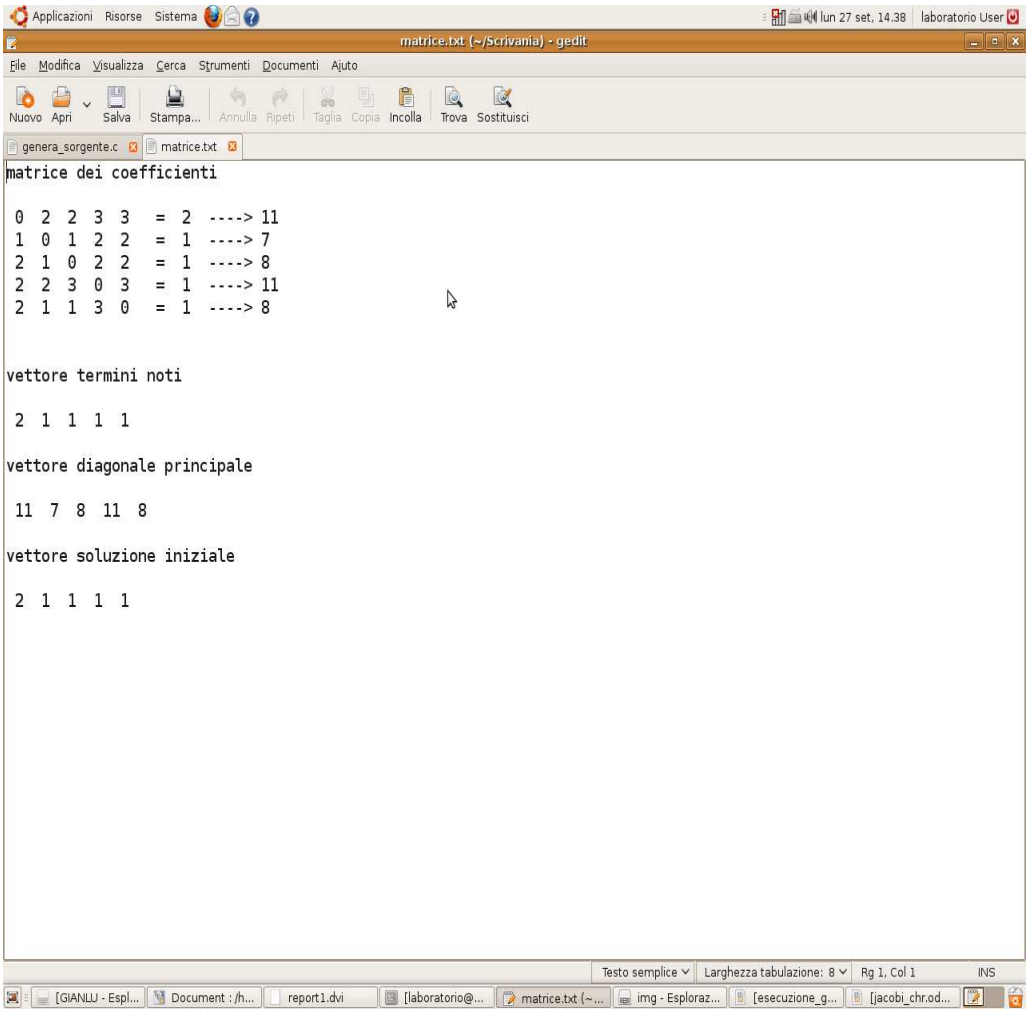

Figura 6: File matrice.txt generato da genera_sorgente.c.

da garantire la convergenza del metodo(matrice a diagonale dominante). A questo punto il file 'jacobi.chr', generato dal file 'genera_sorgente.c', è pronto per essere compilato dal compilatore del linguaggio Chiara.
Il compilatore, come output, genera la tabella di configurazione di un processore dataflow (Figura 7) ed in Figura 8 è mostrata una piccola rappresentazione di tale grafo.

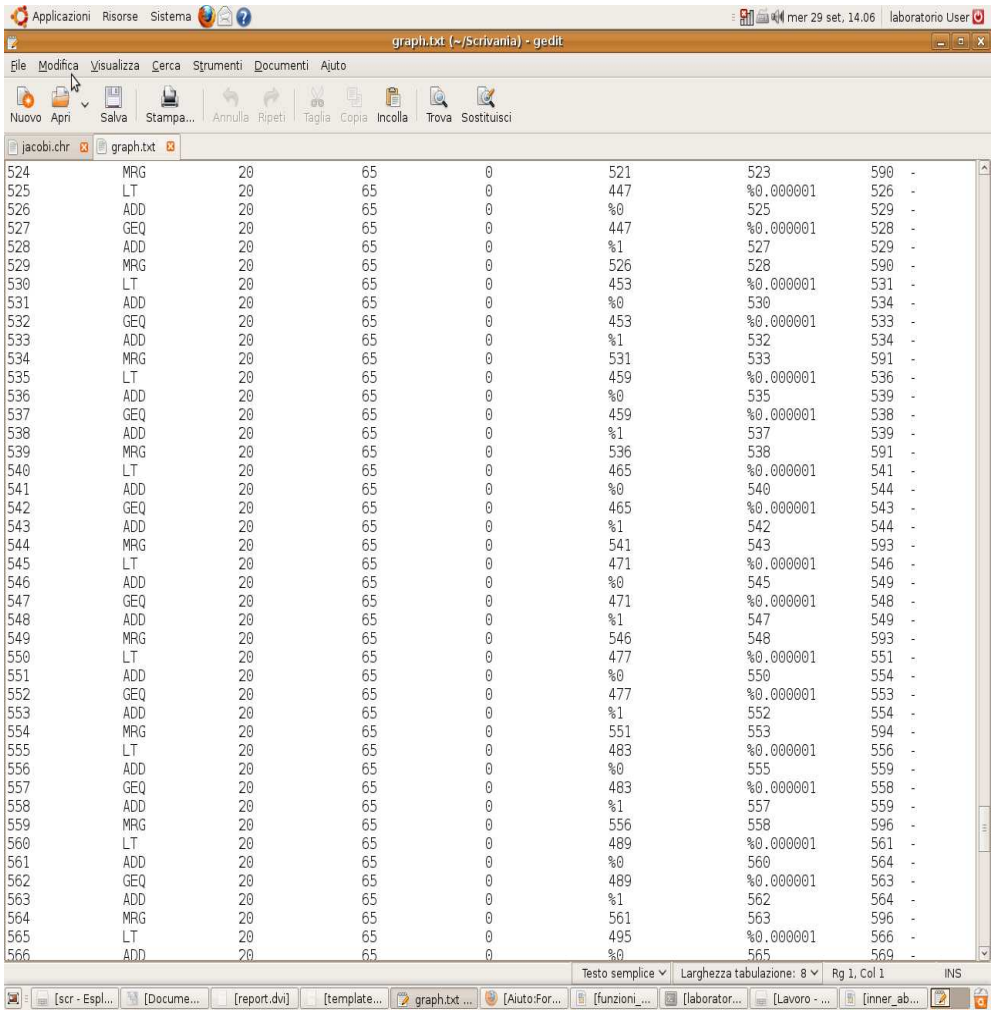

Figura 7: Parte della tabella di configurazione.

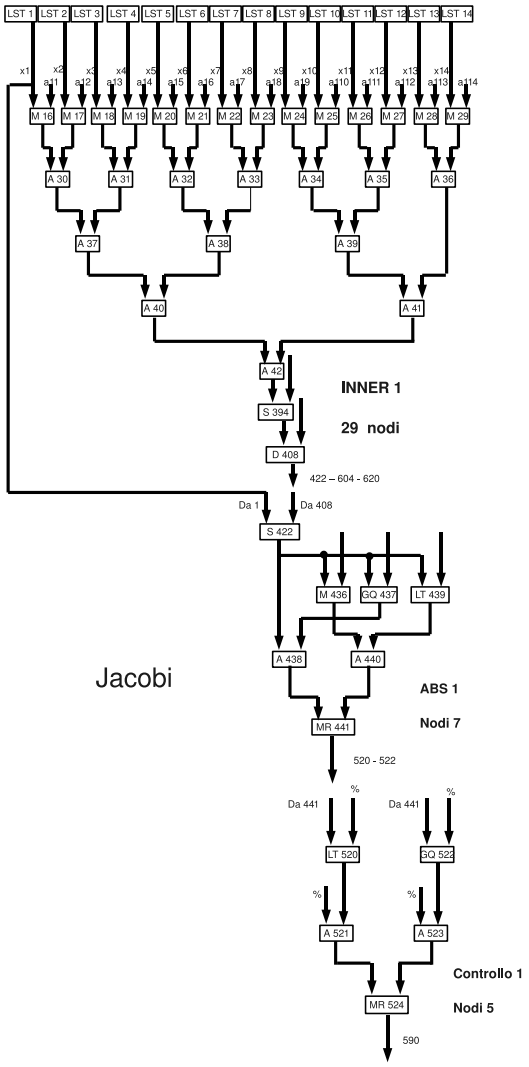

Figura 8: Rappresentazione del dataflow.

5 Partizionamento del grafo dataflow con Chaco

Dopo aver scritto e compilato il programma 'jacobi.chr', il passo successivo è stato partizionare il grafo dataflow mediante il software Chaco.
Prima di presentare la fase del partizionamento, è utile fornire una breve spiegazione del grafo e la sua rappresentazione.
Esso è composto da 634 nodi e 929 archi come rappresentato nello schema a blocchi di Figura 9. Per una chiara comprensione, i blocchi più significativi appaiono colorati in modo da spiegarli singolarmente.

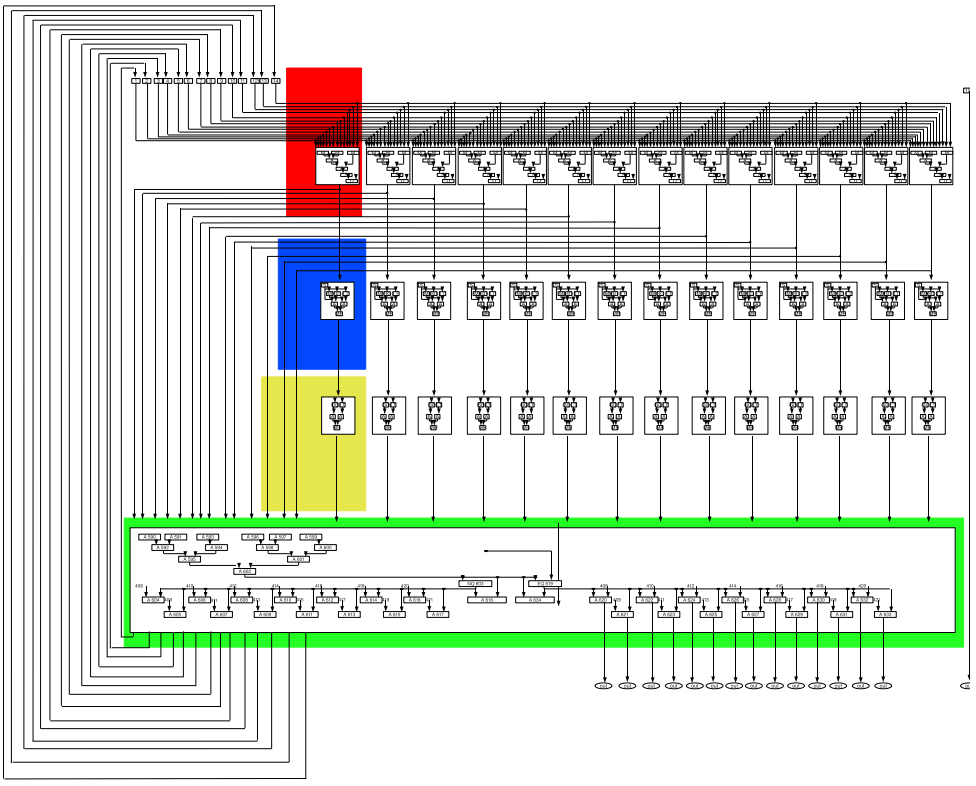

Figura 9: Rappresentaziona a blocchi del dataflow di jacobi.

Il blocco in rosso di Figura 9 è dettagliato in Figura 10 ed è formato da 29 nodi che calcolano il generico passo dell'algoritmo di Jacobi per l'incognita x_1: $x_1^{(k+1)} = -\frac{1}{a_{1,1}} \sum_{j=1}^{n} a_{1,j} x_j^{(k)} + \frac{b_1}{a_{1,1}}$. Questo schema è presente 14 volte nella rappresentazione a blocchi di Figura 9, per ogni x_i.

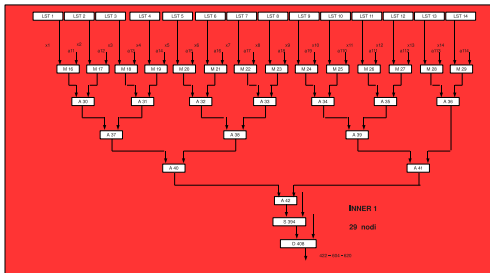

Figura 10: Blocco rosso di Figura 9 in dettaglio.

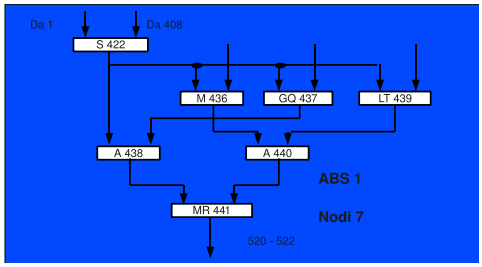

Figura 11: Blocco blu di Figura 9 in dettaglio.

Il blocco in blu è dettagliato in Figura 11 ed è composto da 7 nodi e calcolano il valore assoluto della differenza tra $x_1^{(k+1)}$ e $x_1^{(k)}$, cioè: $\mid x_1^{(k+1)} - x_1^{(k)} \mid$. Anche questo schema, come il precedente, è presente 14 volte.
Il blocco in giallo è dettagliato in Figura 12 ed è composto da 5 nodi che valutano la condizione di arresto: $\mid x_1^{(k+1)} - x_1^{(k)} \mid$<eps. Anche esso è presente 14 volte.

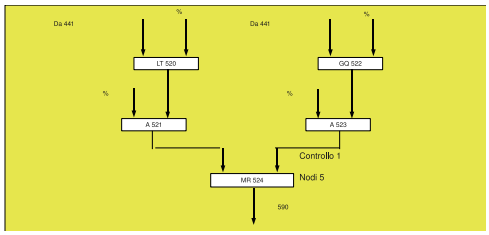

Figura 12: Blocco giallo di Figura 9 in dettaglio .

Il blocco in verde è dettagliato in Figura 13 ed è composto da 45 nodi che valutano la condizione di terminazione dell'algoritmo di Jacobi.

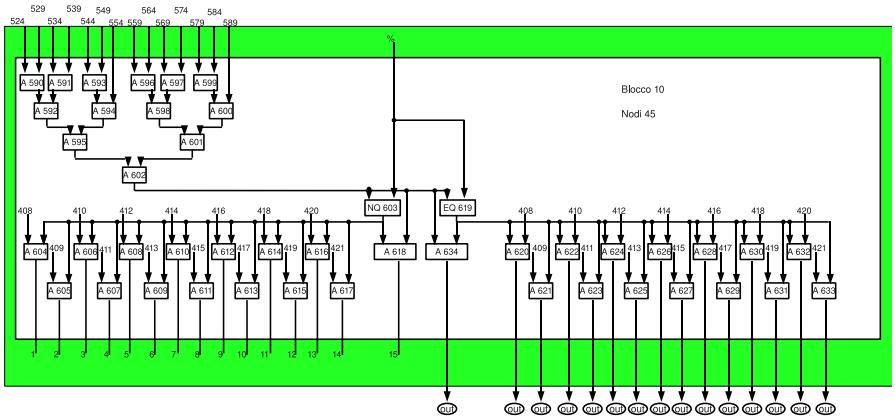

Figura 13: Blocco verde di Figura 9 in dettaglio.

Poichè si è ipotizzato di avere 128 Unità Funzionali a disposizione e il grafo dataflow del programma è formato da 634 nodi, non è possibile creare una corrispondenza 1 ad 1 tra le Unità Funzionali del processore e i nodi del grafo. Di conseguenza, è necessario partizionare il grafo per la sua esecuzione.
Tra i partizionamenti possibili si è scelta la strategia mostrata in Figura 14 quale obiettivo da raggiungere. Tale obiettivo, come migliore soluzione, richiede la suddivisione del grafo del programma in 5 partizioni con un numero di nodi per partizione inferiore od uguale al numero di Unità Funzionali per processore.Inoltre, questa strategia di partizionamento, preserva la natura parallela del metodo, minimizza il numero di comunicazioni tra le partizioni e massimizza il carico computazionale delle partizioni.

Per raggiungere l'obiettivo prefissato e rendere così automatico il processo di partizionamento mediante Chaco, partendo dall'informazione del numero di Unità Funzionali per processore, è stato utilizzato, tra i suoi vari algoritmi implementati, quello basato sul metodo globale **KL-multilivello** [2].

Le Figure 15 e 16 mostrano le opzioni offerte da Chaco per selezionare le scelte che si desiderano applicare al partizionamento. In Figura 15 è possibile selezionare:

1. il nome del file di input che contiene le informazioni relative al grafo dataflow da partizionare. Esso è generato dal file della tabella di configurazione prodotta dal compilatore Chiara, mediante una sua trasformazione eseguita dal programma 'gdltograph.c'. Si presenta come file in formato testo con estensione '.graph' e presenta le seguenti caratteristiche (Figura 17):

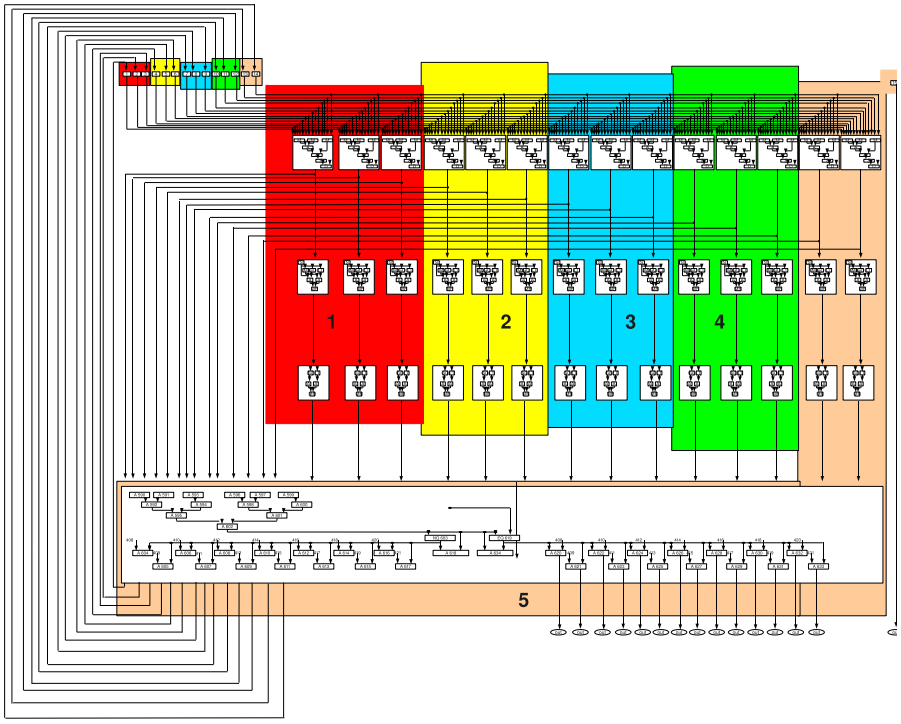

Figura 14: Obiettivo da raggiungere.

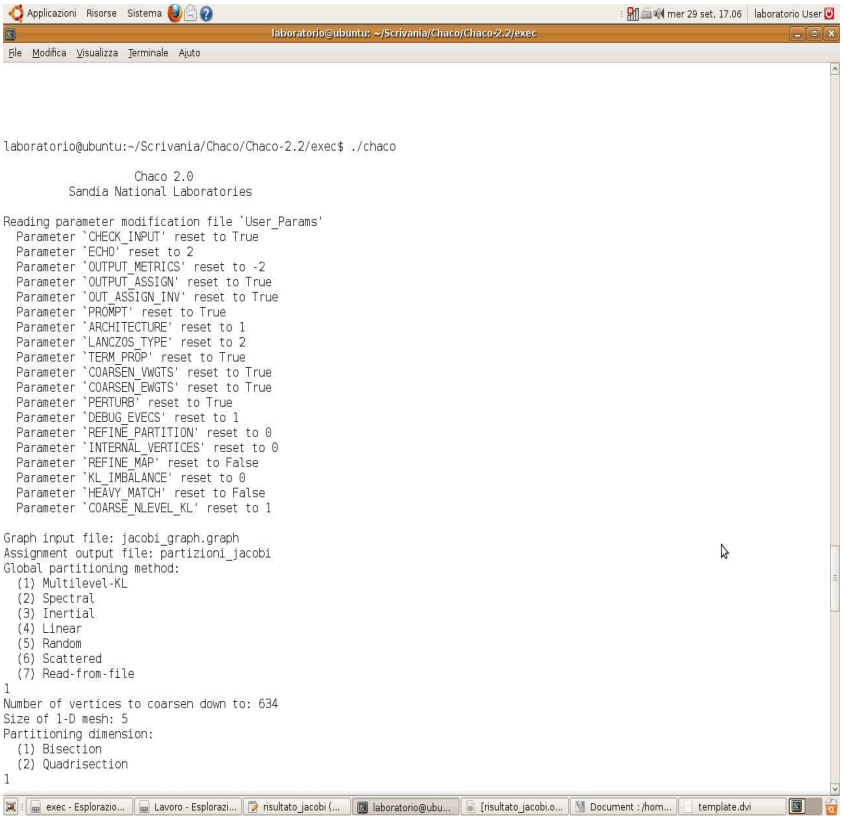

Figura 15: Esecuzione del file 'jacobi_graph.graph' con Chaco.

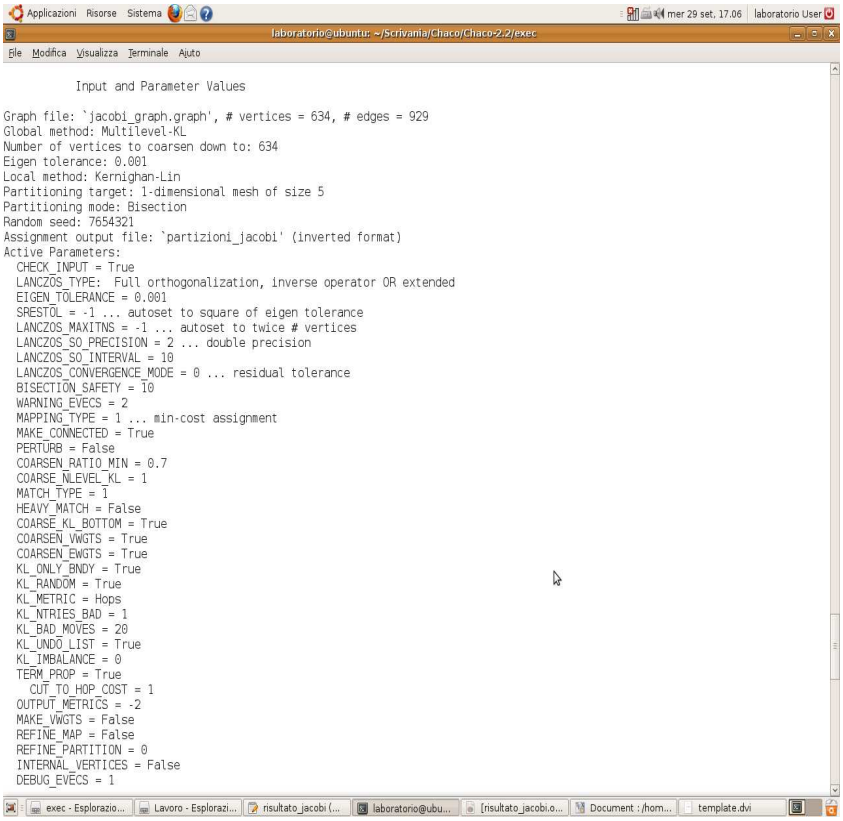

Figura 16: Esecuzione del file 'jacobi_graph.graph' con Chaco.

- La prima riga contiene il numero di nodi, il numero di archi e la direttiva per Chaco sulla topologia del grafo. La direttiva è costituita da un codice binario a tre cifre con i seguenti significati:

  bit meno significativo: valore '0' senza peso associato agli archi; valore '1' peso associato agli archi.

  bit intermedio: valore '0' senza peso associato ai nodi; valore '1' peso associato ai nodi.

  bit più significativo:
  - valore '0': per ognuna delle righe successive Chaco assegna automaticamente in sequenza progressiva il valore dei nodi di partenza del grafo.
  - valore '1': per ognuna delle righe successive il valore dei nodi di partenza del grafo sono assegnati manualmente.

- Le righe successive rappresentano le liste di adiacenza associate ai nodi di partenza, costituite dalle coppie 'nodo' 'peso arco' se il bit meno significativo è '1'.

2. Nome del file in formato testo contenente le partizioni calcolate.
3. Metodo globale scelto.
4. Numero di nodi del grafo grossolano (Number of vertices to coarsen down).
5. Numero di partizioni (Size of 1-D mesh).
6. Tecniche di partizionamento ad ogni passo (Bisection).

Dopo l'inserimento dei parametri richiesti in Figura 15, Chaco genera una panoramica delle scelte effettuate corredendole con i valori dei parametri assegnati automaticamente e rappresentati in Figura 16.

Al termine dell'esecuzione, Chaco stampa a video una serie di informazioni relative al processo di partizionamento e alle partizioni prodotte dando risalto anche al tempo impiegato dal metodo **KL-multilivello** per raggiungere i risultati finali (Figura 18).

Le cinque partizioni prodotte da Chaco sono visualizzate in Figura 19. Il file è composto da cinque colonne, una per ogni partizione generata, ed in ognuna di queste è prima indicato il numero di nodi della partizione e poi gli

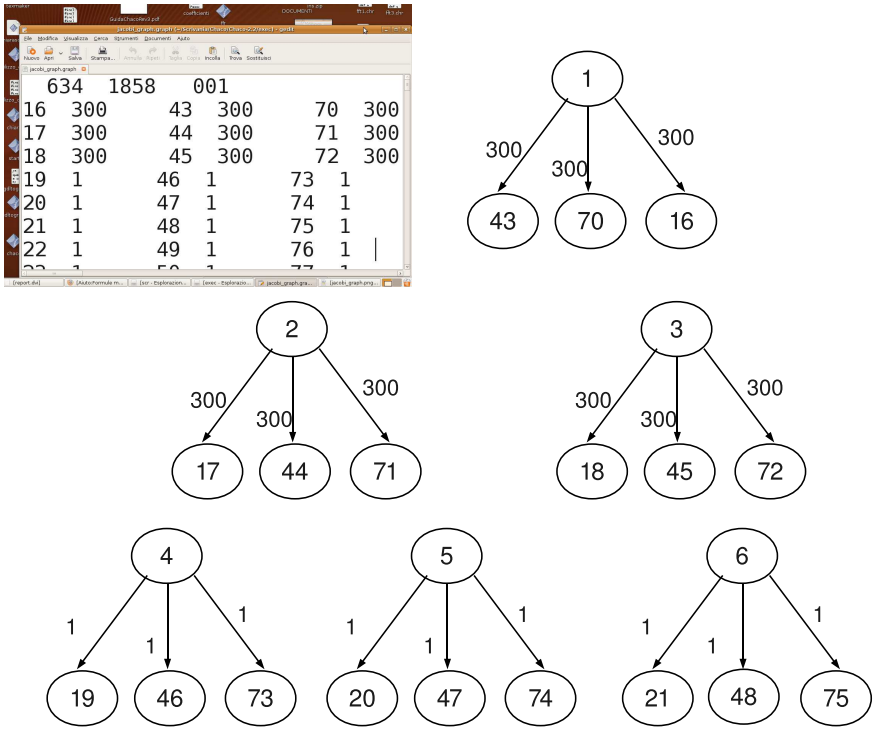

Figura 17: Parte di file 'jacobi_graph.graph' contenente il grafo da partizionare.

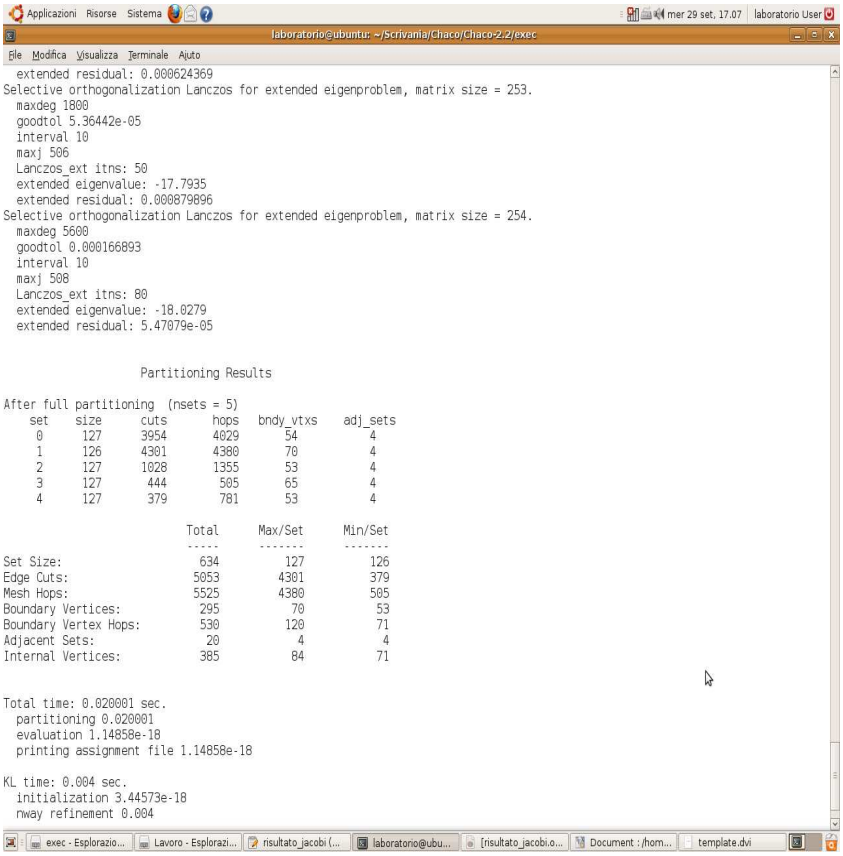

Figura 18: Esecuzione del file 'jacobi_graph.graph' con Chaco.

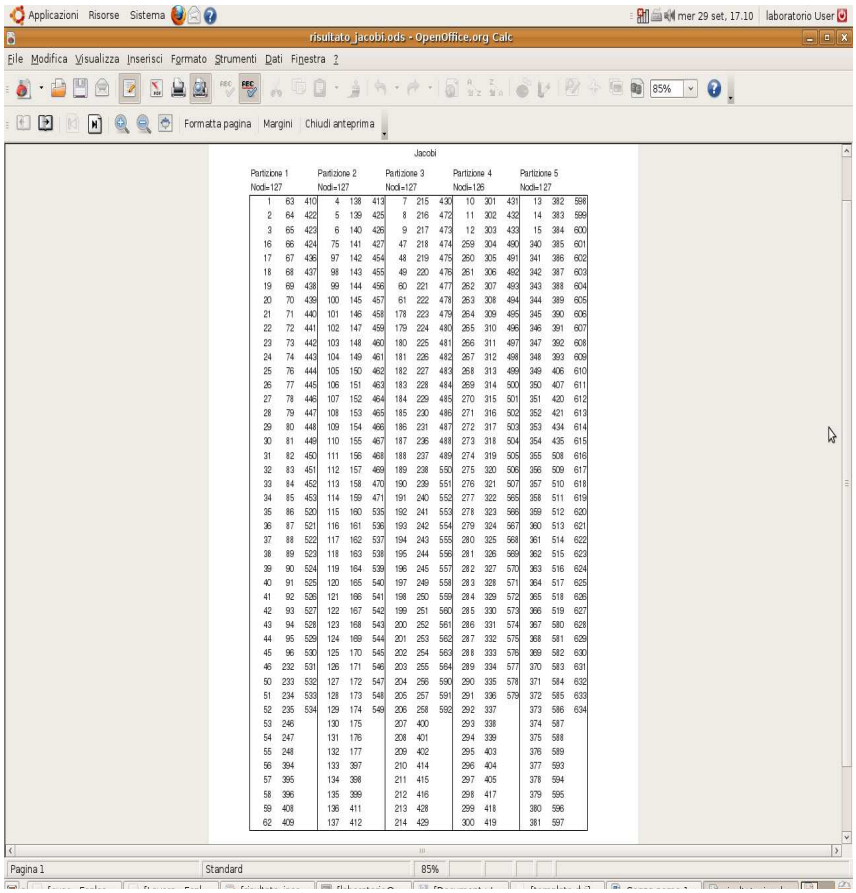

Figura 19: Partizioni prodotte da Chaco.

attori che la compongono.
Comparando i risultati ottenuti con l'obiettivo desiderato, le partizioni fornite da Chaco corrispondono quasi alla perfezione a quelle di Figura 14. Infatti, su un totale di 634 nodi solo 20 di essi non si trovano nelle partizioni preventivate, ma la soluzione ottenuta assume un'importanza diversa se si considera la generalità delle scelte adottate prima e durante il processo di partizionamento. Poichè il risultato di un partizionamento, nel caso di grafi con funzione peso associata ai link, dipende fortemente dal settaggio del peso degli archi, una prima scelta che si è dovuta affrontare è stata appunto la giusta associazione di valori agli archi in modo da fornire delle direttive su come partizionare il grafo a Chaco. A tale proposito si è assegnato il valore '1' agli archi che dovrebbero connettere attori di partizioni diverse e il valore di '300'(quasi la metà del numero di nodi) agli archi che dovrebbero connettere nodi appartenenti alla stessa partizione.
Un altro aspetto importante, che si è cercato di rendere generale, è stato l'individuazione del valore appropriato per il numero di vertici del grafo grossolano (Number of vertices to coarsen down), richiesto da Chaco, prima dell'esecuzione del metodo **KL-multilivello**. Infatti, questo valore ha la caratteristica di modificare, per uno stesso grafo in ingresso, il risultato di un partizionamento. Per questo partizionamento è stato utilizzato il valore '634'(numero di nodi del grafo).
L'ultima scelta che si è operata è stata la *non modifica* dei parametri modificabili dall'utente che Chaco offre ai suoi utilizzattori per modificare la normale esecuzione degli algoritmi implementati. Infatti, modificando opportunamente questi parametri si possono ottenere partizioni che corrispondono al 100% a quelle preventivate di Figura 14, ma se si aumenta o si diminuisce la dimensione del sistema di equazioni lineari, tale modifica produce risultati diversi. In pratica, un settaggio appropriato per un problema di una data dimensione non può essere esteso allo stesso problema di differente dimensione.
Infine, in Figura 20 è riportato il tempo di esecuzione:

- di ciascun blocco di Figura 9;
- di una sequenza di blocchi (colore rosso più colore blu più colore giallo di Figura 9);
- complessivo dell'algoritmo (colore rosso più colore blu più colore giallo più colore verde di Figura 9).

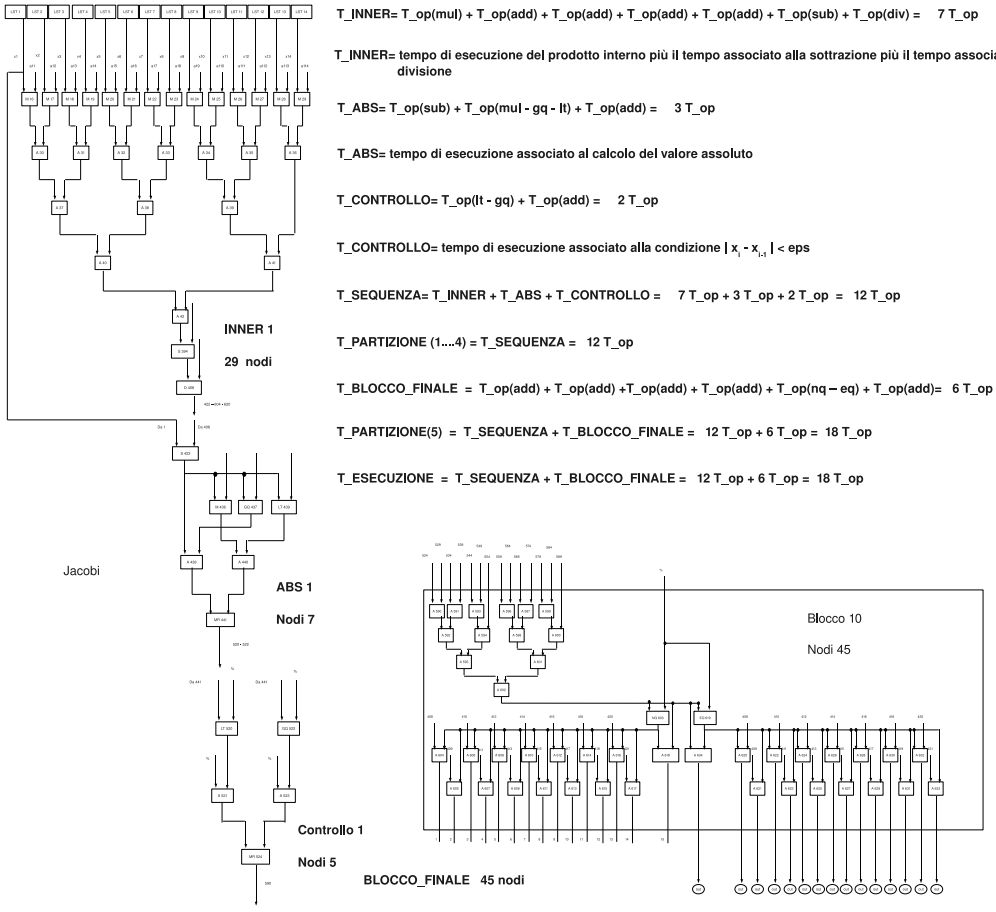

Figura 20: Tempo di esecuzione.

6 Conclusioni

In questo lavoro sono state analizzate le fasi di generazione e partizionamento del grafo dataflow rappresentante il metodo di Jacobi. A tal proposito, è stato scritto il programma 'jacobi.chr' che implementa l'algoritmo di Jacobi per un sistema di quattordici equazioni in quattordici incognite. Inoltre, la natura generale delle funzioni che definiscono l'algoritmo e la possibililità di un'analisi futura del metodo per sistemi di dimensione maggiore hanno suggerito la scrittura dell file 'genera_sorgente.c'.
Dopo aver scritto e compilato il file 'jacobi.chr' il passo successivo è stato il partizionamento del grafo mediante il software Chaco. Poichè il partizionamento di un grafo appartiene alla classe di problemi NP-completo, non esistenza della soluzione universalmente ottima, in questo lavoro si sono prima fissati degli obiettivi di partizionamento e poi si è individuato una metodica per raggiungerli mediante delle direttive di carattere generale da fornire a Chaco per ottenere una soluzione finale molto prossima a quella desiderata. Sebbene sia possibile, settare ad hoc manualmente i parametri di Chaco per ottenere delle partizioni identiche a quelle fissate negli obiettivi, si è osservato che lo stesso settaggio non sempre fornisce i risultati attesi se la dimensione del sistema di equazioni lineari varia. Ciò è dovuto al fatto che il blocco che esegue il prodotto vettore-vettor cambia di dimensione.

Riferimenti bibliografici

[1] B. Hendrickson and R. Leland. The CHACO user's guide – version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.

[2] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28, New York, NY, USA, 1995. ACM.

[3] L. Verdoscia. CODACS project: A demand-data driven reconfigurable architecture. In *Euro-Par 2002*, volume 2400 of *LNCS*, pages 547–550, Paderborn, Germany, August 27–30, 2002. Euro-Par Conference Series, Springer-Verlag.

[4] L. Verdoscia, M. Danelutto, and R. Esposito. CODACS prototype: CHIARA language and its compiler. In *Proceedings of the First International Workshop on Embedded Computing*, Tokyo University of Technology, Hachioji, Tokyo, Japan, March 23–26, 2004. IEEE Computer Society Press.

[5] L. Verdoscia and R. Esposito. Introduction to CHIARA language. Technical Report 20, CNR Research Center on Parallel Computing and Supercomputers, Via Castellino, 111 - 80131 Napoli - Italy, September 2001.

[6] L. Verdoscia and R. Vaccaro. An asynchronous many-core dataflow processor for D3AS prototype. In *Workshop on Highly Parallel Processing on a Chip (EuroPar07-HPPC)*, IRISA, Rennes, France, August 28, 2007. IEEE Computer Society Press.